BARTSTATION

A Homebrew 8-bit Video Game System with VGA Output

Bart Trzynadlowski

January 8, 2006

Table of Contents

0. Introduction **1**

1. Z80 Circuit **2**

1.1 Z80 2
1.2 Oscillator 2
1.3 Reset Circuit 3
1.4 Memory 3
1.5 Address Decoding 5
1.6 I/O Registers 7
1.7 Interrupts 8
1.8 Power Supply 8

2. Joypad **9**

2.1 Pin-Out 9
2.2 Interface 10
2.3 6-Button Joypad 10

3. FPGA-to-Z80 Interface **12**

4. Video Controller **13**

4.1 VGA Interface and Timing 13
4.2 Frame Buffer 13
4.3 Input Port 14
4.4 Asynchronous Operation of the Input Port 15
4.5 Interrupts 17

5. Programming Guide **18**

5.1 Memory Map 18
5.2 Memory Space, Stack Space, and Interrupts 18
5.3 Video Registers 19
5.4 A Simple Image Blitter 20
5.5 Joypad 21

6. Conclusion **23**

7. References **24**

Appendix A: Schematic of the Z80 Circuit **25**

Appendix B: Parts **27**

Appendix C: Video Controller VHDL Code and Constraints File **28**

Appendix D: Sample Programs **36**

Appendix E: Pictures **50**

0. Introduction

The BartStation is intended to demonstrate how to construct a simple video game system using an 8-bit microprocessor, ROM storage, and a field programmable gate array (FPGA) for the video output. It's something that I've always wanted to do. Towards the end of the fall 2005 semester, I decided to implement it for both my EE 426 (Microprocessor Applications) final project and my EE 493F independent study. Different parts of this project applied to different requirements of these courses.

For simplicity, the video output is from a frame buffer which is actually a handicap for low-performance systems. Typically, dedicated tile and sprite hardware is used to spare the CPU from having to draw the screen. Frame buffers also occupy much more space but, luckily, memory is cheap these days. Since this was my first FPGA project involving video, it was a good decision. A Sega Genesis joypad was used for input because of its standard DB-9 connector and the ease with which it can be interfaced. Audio capabilities were not implemented.

An 8 MHz ZiLOG Z80 was chosen as the CPU for two reasons. I wanted to use something other than an 8051 microcontroller which we typically use in class. I was bored with it and do not particularly like its pin configuration – dedicated ports are unnecessary for this project and the multiplexed data and address bus means an extra latch must be used for addressing.

Figure 0.1 Pac-Man, 1980.

Figure 0.2 Nintendo Game Boy, 1989.

Secondly, and more importantly, I'm interested in older video game systems where the Z80 happens to appear very often as either the main CPU or sound co-processor. Two iconic machines, among countless others, used the Z80 as their CPU: Pac-Man, the arcade game (figure 0.1), and the Nintendo Game Boy handheld system (figure 0.2.) Therefore, the Z80 is an obvious choice for a "homebrew" project.

A Xilinx Spartan-3 FPGA on Digilent's Spartan-3 Starter Kit Board was used to implement the video hardware. The board includes a 200,000-gate Spartan-3 XC3S200, a VGA connector and 1MB of SRAM which is used as a frame buffer.

The rest of this document describes the design of the system in detail. The appendices include schematics and code. **Appendix E** contains pictures of my implementation.

1. Z80 Circuit

The CPU, memory, joypad, and the LVCMOS interface for the FPGA are all implemented with discrete components on a breadboard. It is collectively referred to as the "Z80 circuit." The schematic is available in **Appendix A**. A list of parts and materials is provided in **Appendix B**. Here, we will discuss the design of the Z80 circuit.

1.1 Z80

The Z80 is the CPU of the BartStation. It initiates, controls, and coordinates all activity of the system by executing game code. Figure 1.1 shows a logical diagram of the Z80 with all signals grouped by function.

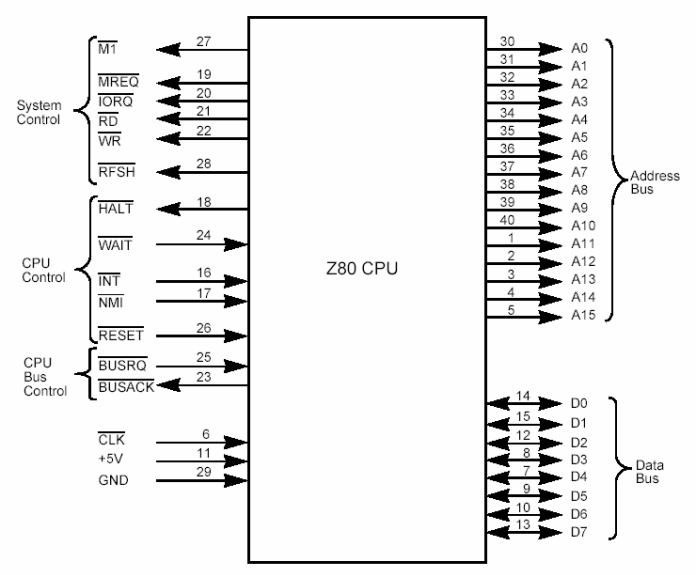


Figure 1.1 Z80 signals.

Unused output signals are left unconnected. These are M1, MREQ, IORQ, RFSH, HALT, and BUSACK. The input signals BUSRQ, INT, and WAIT are also unused and are tied to Vcc to permanently de-assert them. The rest of the signals are discussed throughout this section.

1.2 Oscillator

A square wave oscillator is required by the Z80 to generate its clock signal. This clock signal effectively synchronizes all activity in the circuit because each device responds to signals from the CPU. The video controller uses its own clock and is therefore the only component that is asynchronous with respect to the CPU.

Figure 1.2 8.000 MHz half can TTL oscillator.

Some microprocessors provide two oscillator inputs which internally lead to an inverter to create feedback (and thus establishing an oscillation.) The designer will typically attach a crystal oscillator and a couple of capacitors. The Z80 has only a single oscillator input, CLK, and expects a TTL waveform. This is accomplished by using a TTL oscillator which has three pins: Vcc, Gnd, and the output signal. The BartStation's CPU uses an 8.000 MHz oscillator (figure 1.2.)

1.3 Reset Circuit

To reset the Z80, the RST pin must be pulled low for some small amount of time and then set high again to allow the processor to operate. The circuit in figure 1.3 does just that. In its normal state, with the button released, the capacitor is charged through the 10 K-ohm resistor which brings the voltage at the RST node to Vcc. Pressing the button discharges the capacitor through the 1 K-ohm resistor with a time constant of 10 ms. When the capacitor is discharged, the equivalent circuit appears as a resistor divider and the voltage should be 0.45 V (in reality, it is about 0.5), which is interpreted by the Z80 as a logical low.

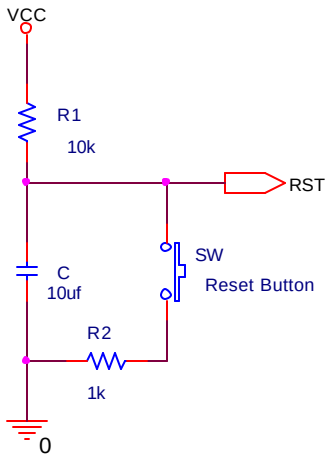


Figure 1.3 Reset circuit.

The resistors and capacitor are necessary to allow the voltage to switch from high to low over a short time constant. They also serve to protect the reset pin by limiting the amount of current that will flow when the button is pressed. Although it's likely that the Z80 has a field-effect transistor connected to its reset pin which does not draw current, an in-rush current charging the gate capacitance can still damage the device. Changing the capacitor will only change the time constant. A 1 µF or 0.1 µF capacitor might also work.

1.4 Memory

A key component of any computer system is memory, both run-time and permanent. The BartStation stores its programs (its games, effectively) in read-only memory (ROM.) In many commercial video game systems, especially those from the early 90's and older, the game ROMs were located on separate cartridges allowing different games to be played on the same base system. In our case, the ROM chip is simply located in the game system circuit itself. After all, this isn't a production machine!

Random access memory (RAM) is only available at run-time and is volatile; that is, it does not retain its contents when the power is off. It is used by programs for storing data that must be updated and manipulated while the software is running. Static RAM (SRAM) is the simplest form of memory to interface to. As long as power is supplied to it, it will remember what was written and will not lose its contents over time as dynamic memory does. This means it does not need to be refreshed – hence the term "static."

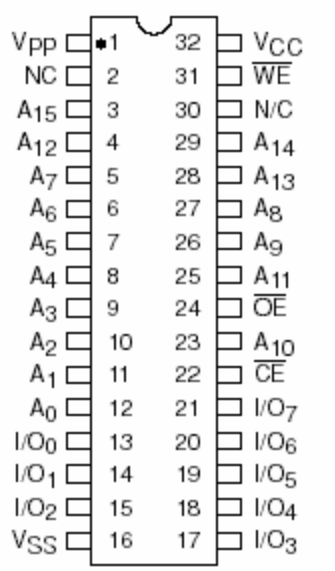


Figure 1.4 CAT28F512 64KBx8 Flash Memory.

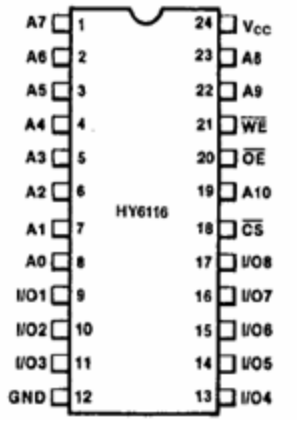


Figure 1.5 6116 2KBx8 Static RAM.

In the interest of making life easier for us, the BartStation uses flash memory as its ROM medium. Typically, mask ROMs (which can be burned in once and cannot be changed) or EPROMs (electrically programmable ROMs which can be written multiple times and are erased with ultraviolet light) are used in video game systems. These have major disadvantages for impatient and poor students such as myself. Mask ROMs are intended for high-volume production systems and cannot be re-written. EPROMs are often used for prototyping but they require a UV eraser and must be exposed for several minutes before being reused. This can slow down testing.

Flash memory does not have these limitations. Despite being more expensive per unit than EPROMs, flash memory chips are well worth the modest price premium when their ease of use is considered. Flash is electrically programmable *and* erasable. Many flash parts are available with pin-outs and interfaces virtually identical to traditional ROM chips.

One such part is the CSI CAT28F512 (figure 1.4.) It is pin-compatible with similar devices such as the Atmel AT29C512 and the SST 29EE512. These devices can be written with most standard device programmers or one can choose to build a custom programmer. Writing is done in a similar fashion as to SRAM with the exception that a programming voltage must be applied at the Vpp pin (unless it is a 5V-programmable part) and data must be written to addresses that lie in the same "sectors." The details of this procedure are beyond the scope of this document. We are only concerned with reading the device.

To do this, the procedure is quite simple: An address must be placed on the address bus (pins A15-A0), the chip enable (CE) must be asserted by pulling it low, and the output enable (OE) must then be pulled low to allow the data to appear on the data bus (I/O7 – I/O0) after a small access time delay. In the case of the BartStation, timing constraints are a non-issue because the memory is much faster than the 8 MHz Z80 whose read timing is shown in figure 1.6. By the time the Z80 samples the bus for data after asserting the address, the data is certain to be available.

The Z80 data and address buses are connected directly to the CAT28F512, except for pins A14 and A15. These are tied to ground. This provides for 16 KB of ROM space and was done because the entire address space of the Z80 is 64KB – the same size as this chip provides! In order to leave room for other memory regions and I/O areas, 16 KB was chosen as the size of the ROM space. The OE pin is connected to the Z80's RD signal.

SRAM is similarly simple. The BartStation uses the Hyundai 6116 (figure 1.5) which provides 2KB of memory. Reading is exactly the same as with the flash chip. The Z80 is connected to the SRAM in the same manner as the flash except for two differences: Only pins A9-A0 are connected (for 1KB of memory, an arbitrary design decision) and the Z80's WR signal is connected to the WE pin (write enable, which allows data to be written.)

How these memories are mapped into the Z80 address space is discussed in section **1.5**.

Note that when the output enables are not selected, the data buses are put into a high-impedance (high-Z) state which does not drive the bus. This is an important point because it allows for the creation of buses shared by many devices. If two devices were to access the bus at the same time, they could be damaged due to a short circuit. The high-Z state effectively makes it appear as though the device is no longer even connected to the bus.

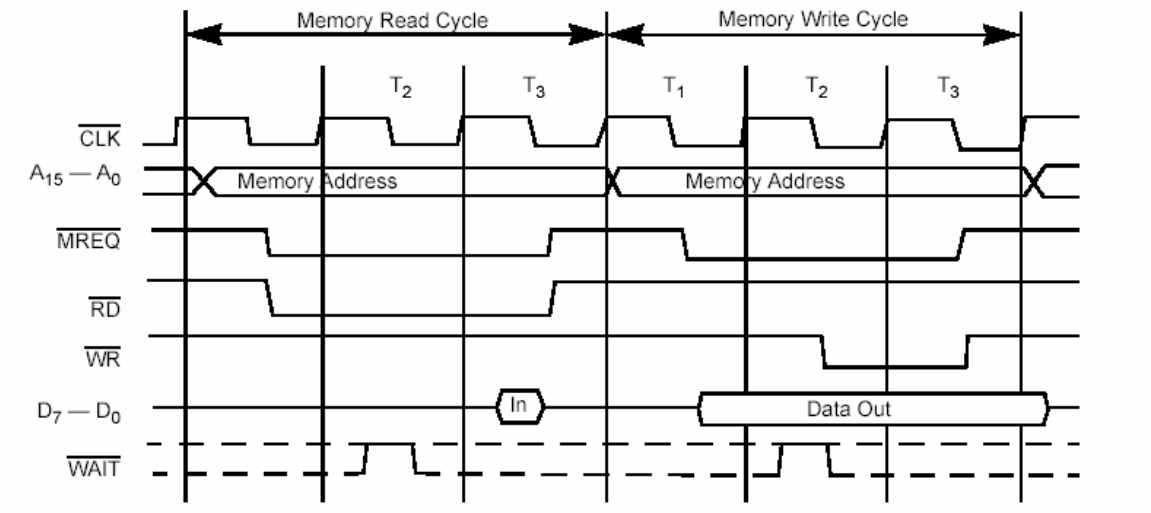


Figure 1.6 Z80 memory access timing.

The Z80 memory timing is shown in figure 1.6. Reads and writes take 6 clock cycles each and the time between address set-up and data read/write is on the order of at least a cycle or two which is plenty of time for our particular ROM and RAM chips (to verify this, see their respective datasheets.) The MREQ and WAIT signals are unnecessary for our purposes.

1.5 Address Decoding

Address decoding is the process of selecting devices according to their memory address. We want particular ranges of the Z80's address space to correspond to particular memory chips and I/O registers. This is accomplished by generating chip enable signals using the address as an input so that only the requested device responds to a read or write. In the BartStation, the 74HC145 (figure 1.7), a 4-to-16 decoder, is used along with a 74HC08 quad AND gate (figure 1.8.)

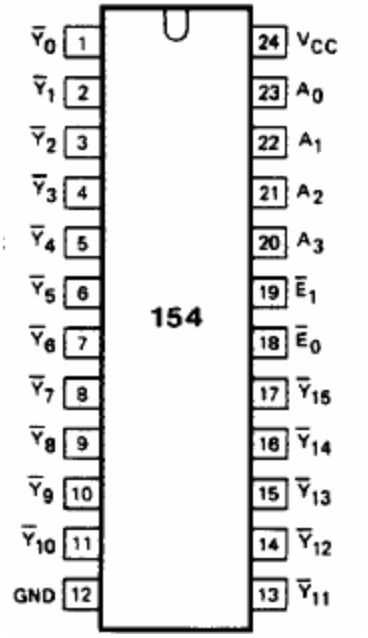


Figure 1.7 74HC154 4-to-16 decoder.

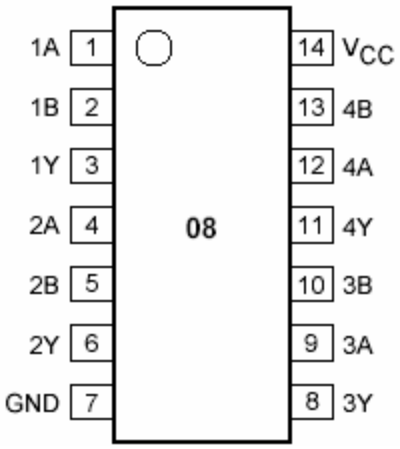


Figure 1.8 74HC08 quad 2-input AND gate.

The decoder works by mapping each possible 4-bit number presented on its input pins, A3-A0, to one of the output lines, Y15-Y0. There are 16 unique outputs which is also the number of total possible combinations on the inputs. Address decoding is accomplished by connecting some of the higher order address bits to the decoder inputs. Different address ranges will cause unique outputs to be selected. The decoder's outputs happen to be active low, meaning they are 0 when selected and 1 otherwise. Our memory chips have active low chip enable pins and can therefore be connected directly to the decoder.

The upper 4 bits of the Z80 address bus are used for decoding which allows for a total of 16 possible unique devices to be selectable. In our case, the device mapping is shown in table 1.1.

Table 1.1 Mapping of address ranges to devices in the BartStation.

A15-A12	Device
$0	ROM
$1	ROM
$2	ROM
$3	ROM
$4-$7	*Unused*
$8	Video Data Register
$9	Control Register
$A	Joypad Status Register
$B-$E	*Unused*
$F	RAM

From the table, it can be seen that the ROM is selected by 4 distinct decoder outputs. When an output is not selected, it is set to 1, not high-Z, so we cannot wire all of these directly to the ROM's CE pin. This would cause the various outputs to fight for control of the node connecting them and could possibly result in a short circuit between two or more of the pins. The solution is to AND all the ROM decoder outputs together. That way, when any one of them is selected (pulled to a logic 0, indicating the ROM should be enabled), the AND function will return 0 thereby asserting CE. Otherwise, they will all be 1 and the result will be 1, disabling the ROM. Figure 1.9 shows how the 3 of the 4 2-input AND gates are strung together to perform this function; pin numbers are included.

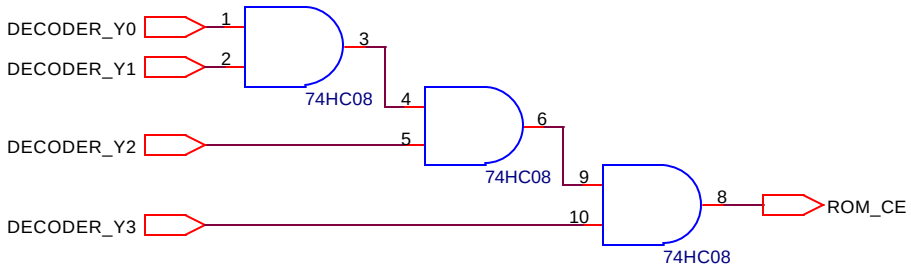


Figure 1.9 Selecting the ROM with any of the lower 4 decoder outputs.

The decoder is always kept enabled in this design by tying both enable bits, E1 and E0, directly to ground. Some applications may require decoders to be disabled (forcing all output pins to 1) but in the case of the BartStation, all activity on the system address and data buses is caused by memory and I/O accesses.

1.6 I/O Registers

There are 3 registers mapped to the address space; two are write-only and one is read-only. The first of these is the Video Data Register which appears at $8000 and is write-only. It is connected directly to the FPGA board and consists of 8 data bits (the Z80 data bus), an enable signal from the decoder, and the Z80 WR signal. The FPGA interface is described in more detail in section **3**.

The other write-only register is the Control Register. Part of it consists of the data bits routed to the FPGA and its own enable signal. Additionally, one data bit is also passed through a latch (figure 1.10) to the joypad to control button multiple xing. The joypad is described in more detail in section **2**. Because the latch enable bit is active high and the decoder is active low, an inverter (figure 1.11) is necessary. The output enable is always active (pulled to ground) because the latched multiple x bit must always be available to the joypad. The 74HC14 was chosen because it was readily available. It happens to be a Schmitt trigger as well.

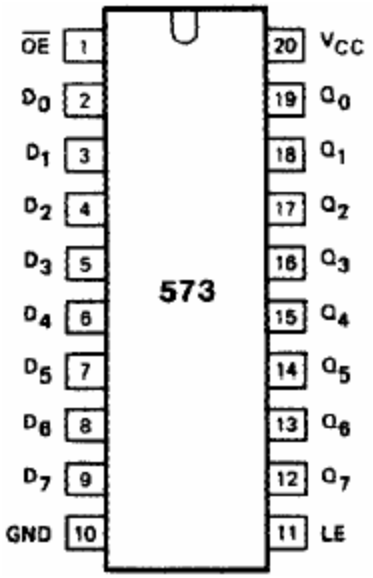


Figure 1.10 74HC573 8-bit latch with 3-state outputs.

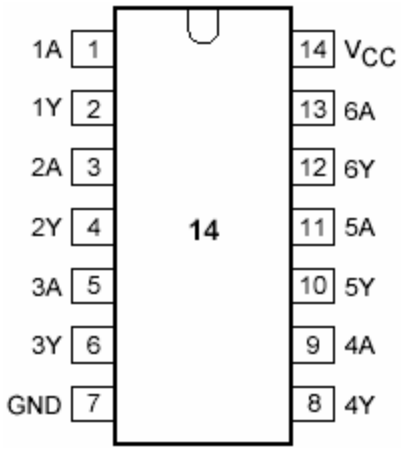


Figure 1.11 74HC14 Schmitt trigger inverter.

1.7 Interrupts

IRQs, or normal, mask-able interrupts, are not supported by the system. The IRQ line on the Z80 is always deactivated. Interrupts are only available in the form of non-maskeable interrupts (NMI) which cannot be disabled by software except by a video controller setting. The FPGA's VBL interrupt line is connected directly to the Z80 and is triggered once per frame at the beginning of the vertical blanking (or retrace) period. This occurs at the same rate as the display refresh: 60 Hz. The Z80's CMOS logic levels are such that a 3.3V signal from the FPGA will be correctly interrupted as '1', so no level shifting electronics are necessary.

1.8 Power Supply

The Z80 circuit runs off of a 5V power supply. In order to achieve this, a 7805 linear regulator (figure 1.12) is used to generate this from a higher DC voltage provided by the AC/DC wall-plug adapter (approximately 14-18V.) Refer to the schematic for details.

Figure 1.12 7805 voltage regulator.

A major advantage of a 7805-based power supply design over others is that it is extremely simple, requiring only a couple of capacitors on its input and output. The drawback is that it is a fairly inefficient power supply which dissipates power in the form of leakage current. The BartStation's 7805 becomes very hot during operation despite drawing under 100 mA. A heat sink is recommended. A superior design would involve a switching power supply chip. Due to time constraints and part availability, this was not attempted.

All ICs in the circuit include a decoupling capacitor of 0.1 µF between their Vcc and Gnd pins which is placed as close as possible to the chip. The purpose of a decoupling capacitor is to prevent the voltage supply on its corresponding chip to drop if the device suddenly switches a number of pins. Inductance and other losses in the pins and connecting wires can cause a transient spike in current which can pull Vcc lower than normal. The capacitor will begin to discharge while this is occurring and serves to decouple the chip from the circuit's Vcc rail.

2. Joypad

A Sega Genesis joypad (figure 2.1) is used by the BartStation because it uses a standard DB-9 connector and is easy to interface (hardware and software-wise.)

Figure 2.1 Standard Sega Genesis 3-button joypad.

The joypad has a directional pad (up, down, left, and right buttons), a Start button, and 3 additional buttons: A, B, and C.

2.1 Pin-Out

The joypad uses a female DB-9 connector (figure 2.2) and the game system has a matching male connector (figure 2.3.)

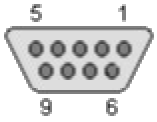


Figure 2.2 Joypad's female DB-9 connector.

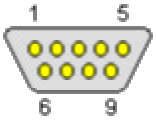


Figure 2.3 BartStation's male DB-9 connector.

The pin mapping is shown in table 2.1. There are two columns per pin because of the multiplex bit (MUX) which is used to select which button to map to a given pin. There are too many buttons to allow for a direct mapping so, internally, the joypad uses a 74HC157 quad 2-to-1 multiplexer.

Table 2.1 Joypad pin assignments.

Pin	**Function** (MUX = 0)	**Function** (MUX = 1)
1	Up	Up
2	Down	Down
3	*None (grounded)*	Left
4	*None (grounded)*	Right
5	Vcc (5V)	Vcc (5V)
6	Button A	Button B
7	MUX input	MUX input
8	Gnd	Gnd
9	Start Button	Button C

The joypad is active low; when a button is pressed, the corresponding pin is 0, otherwise, it is 1.

2.2 Interfacing

The joypad is connected to two 74HC573 latches in the BartStation. One is for output and only uses a single bit (MUX.) It is selected by the same address as the Control Register. The other is used for input and has pins 1, 2, 3, 4, 6, and 9 connected.

Reading the complete button status is a 4-step process:

1. Set MUX to 0.
2. Read buttons.
3. Set MUX to 1.
4. Read buttons.

Of course, the order of the MUX settings is interchangeable. Tables 2.2 and 2.3 show the format of the Control Register (address $9000) and Joypad Status Register (address $A000), respectively.

Table 2.2 Control Register.

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Name					JP	M2	M1	M0

Table 2.3 Joypad Status Register.

Bit	Multiplex	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Button	JP=0			Up	Down	0	0	A	Start
	JP=1			Up	Down	Left	Right	B	C

The JP bit in the Control Register is mapped to MUX. Sample code is provided in section **5**.

2.3 6-Button Joypad

Sega released a 6-button joypad (figure 2.4) for the Genesis which was backwards compatible with the 3-button model. The additional buttons (X, Y, Z, and Mode) are accessible by toggling the multiplex bit between 0 and 1 in a particular sequence (table 2.4.) The multiplex bit must be written in the indicated order or the joypad will reset to the beginning of the sequence. After a very short timeout, the joypad automatically resets to maintain compatibility with 3-button pads.

Figure 2.4 6-button joypad.

Some old Genesis games read 3-button pads using more than the 4-step sequence described earlier. This could cause problems when a 6-button pad was used because the wrong button status would be returned after a few reads. Sega included a workaround in the form of the Mode button which, if held down while powering up the system, would disable the additional buttons and emulate a 3-button pad.

It may be possible to detect a 6-button pad by checking that step 4 returns all 0's in bits 5 to 2 and step 6 returns all 1's in those same bits.

It is recommended that the 4-step sequence (for 3-button pads) or the 6-step sequence (for 6-button pads) be performed only *once* per frame to avoid any problems.

Table 2.4 Six-step sequence for 6-button joypads.

			Bit							
Step	**Multiplex**		*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
1	JP=1	**Button**			Up	Down	Left	Right	B	C
2	JP=0				Up	Down	0	0	A	Start
3	JP=1				Up	Down	Left	Right	B	C
4	JP=0				0	0	0	0	A	Start
5	JP=1				Z	Y	X	Mode	B	C
6	JP=0				1	1	1	1	A	Start

3. FPGA-to-Z80 Interface

The Z80 operates with 5V CMOS logic whereas the Spartan-3 uses LVCMOS 3.3V logic and is not 5V-tolerant. There are a number of ways to perform voltage level shifting, including a resistor divider network and a transistor switch. However, the simplest, and the one suggested by Xilinx, is to use a series resistor. To understand how this works consider figure 3.1 which is a schematic of the ESD protection diodes immediately inside a Spartan-3 I/O pin.

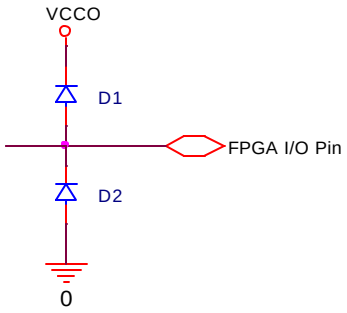


Figure 3.1 Structure of the ESD protection diodes on a Spartan-3 I/O pin.

Vcco is the output driver voltage and is 3.45V. The maximum voltage allowable at the input pin without causing oxide stress is 4.05V. This gives a diode voltage drop of 4.05-3.45=0.6V. The diode happens to be forward biased at 0.5V, so we are now operating under the current constraints of the diode. Xilinx gives the maximum current at 0.6V to be 5.51 mA. Finally, the input voltage from the driving device must be considered. To be safe, a 10% tolerance is used yielding a maximum input voltage of 5V+10%=5.5V. Our series resistor will be placed between the driver and the FPGA as in figure 3.2.

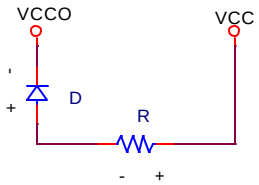


Figure 3.2 Equivalent circuit of the FPGA -Z80 interface.

Determining the resistor value is a simple matter of solving the loop equation for the equivalent circuit.

$$V_{CC} - IR - V_D = V_{CCO}$$

$$R = \frac{V_{CC} - V_D - V_{CCO}}{I} = \frac{5.5\,V - 0.6\,V - 3.45\,V}{5.51 \times 10^{-3}\,A} = 263 \text{ ohms}$$

The closest standard resistor sizes are 270 and 300 ohms.

4. Video Controller

Video is output via a VGA port to a standard computer monitor at a resolution of 320x240 pixels. The actual VGA timing is for a 640x480 display but each pixel is doubled in both the vertical and horizontal directions to halve the resolution.

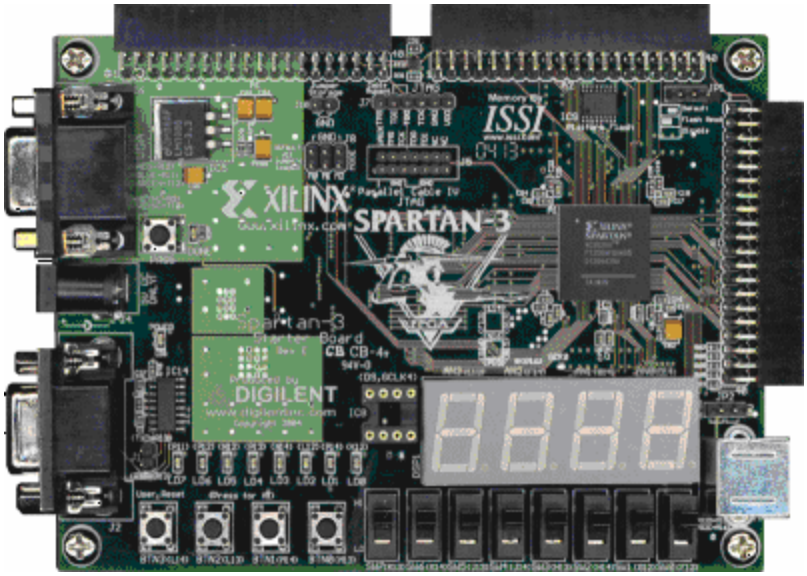


Figure 4.1 Spartan-3 Starter Kit Board. The VGA port is located on the top-left.

The Spartan-3 Starter Kit Board (figure 4.1) conveniently provides a VGA port connected to the FPGA. There is also 1MB of high-speed SRAM which is used by the BartStation to hold the frame buffer.

Appendix C contains the video controller VHDL code.

4.1 VGA Interface and Timing

There are 5 signals that compose a VGA output: Red, Green, Blue, Vertical Sync (VS), and Horizontal Sync (HS.) HS and VS control the horizontal and vertical scan rates, respectively, and therefore the resolution. The standard VGA resolution is 640x480 pixels at 60 Hz and the timing for HS and VS is derived from a 25 MHz clock. Information on VGA timing can be easily found on the web or in the "Spartan-3 Starter Kit Board User Guide."

To implement the correct timing for the HS and VS pulses, a series of counters were used based on the 25 MHz pixel clock (which in turn is derived from the board's 50 MHz clock.) When the counters reach certain values, the sync lines are toggled.

Colors are represented in terms of their primary components: Red, green, and blue. Each corresponding signal is analog and can range from 0V to 0.7V. The Starter Kit Board only allows for one bit of resolution for each of the components giving a total of 8 possible colors.

4.2 Frame Buffer

A frame buffer is kept in the Starter Kit Board's SRAM. It is linear and each pixel occupies one byte for a total of 76,800 bytes (320x240 pixels.) During the refresh period, pixels are read from the frame buffer and sent directly to the VGA R, G, and B lines.

Frame buffers are highly disadvantageous for use in low-performance video game systems and no early systems used them. The bandwidth is simply not sufficient to send a screen full of pixels every frame (or even at half or a quarter of the refresh rate.) A better solution is to use tiled background layers and hardware sprites. In such schemes, the pixel data for all tiles is written once and updated infrequently. Tiles are positioned by changing pointers (often less than 16 bits in typical implementations) in a table. Hardware scrolling is almost always provided. Sprites can be positioned and oriented according to parameters in a table as well.

Nonetheless, frame buffers are very simple to implement and the decision was warranted for that reason alone.

4.3 Input Port

The Z80 cannot address a 76,800-byte frame buffer with its 16-bit address bus. It would also be impractical to implement a proper bank switching scheme to map part of the frame buffer into the Z80's memory space because of the number of lines that would have to be routed from the FPGA board. Instead, a simpler approach was chosen, one which is common in actual video game systems: Input "ports" or registers that provide a gateway to the frame buffer.

There are two write-only registers provided by the video controller which are collectively referred to as the "input port." The first is Video Data Register (table 4.1), mapped to address $8000, and the second is the Control Register (table 4.2), mapped to $9000 and shared with the joypad multiplex bit. The Control Register is primarily used to load an address into an internal frame buffer write pointer and the Video Data Register is used to actually send data to the frame buffer, automatically incrementing the pointer for each pixel. In order to relieve the CPU of the burden of having to calculate an offset into the frame buffer and pass it along, the video controller can handle that directly and the CPU must only send X and Y coordinates.

Table 4.1 Video Data Register. The purpose of the bits depends on the current Control Register mode.

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Name	D7	D6	D5	D4	D3	D2	D1	D0

Table 4.2 Control Register.

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Name					JP	M2	M1	M0

How the Video Data Register is interpreted depends on the Control Register Mode. The procedure for sending data to the input port is to write to the Control Register, setting the mode (which is latched), and then writing to the Video Data Register. Table 4.3 shows the bit settings for the different modes.

Table 4.3 Data register modes.

M2-M0	**Mode**
000	Data mode.
001	X coordinate low.
010	X coordinate high.
011	Y coordinate.
100	Settings.
…	Reserved.

The following is a description of the various modes:

Mode 0: Data Mode
In this mode, only the lower 3 bits of the Video Data Register matter. D2-D0 are red, green, and blue, respectively. This color data forms a single pixel which is written to the frame buffer at the current internal write pointer address. The pointer is automatically incremented after each write.

Mode 1: X Coordinate Low
Because the screen is 320 pixels wide, 9 bits are required to represent an X coordinate. This mode accepts the lower 8 bits of that coordinate which is in turn used to calculate a new write pointer address. If the X coordinate written is greater than 319, the results are undefined.

Mode 2: X Coordinate High
This is the most significant bit (the 9^{th}) of the X coordinate.

Mode 3: Y Coordinate
This is the Y coordinate. If it exceeds 239, the results are undefined.

Mode 4: Settings
This mode is used to configure the video controller. Bit D0 controls the display (1 turns it on, 0 turns it off causing black pixels to be output) and D1 controls interrupts (1 enables them, 0 disables them.) The display and interrupt settings do not affect the operation of the other modes.

Addresses and data may be set while the display is on or off. The X and Y coordinates are latched internally and the write pointer is updated each time they are changed. However, the latched coordinate are never updated internally, even when the write pointer auto-increments. The next time any part of the coordinate is changed, the write pointer will be calculated using the new bits and the old coordinates from the last time they were written.

An example of where this behavior is useful is in a sprite renderer. The X coordinate can be set once and then, to advance to the next line, only the Y coordinate needs to be updated. Each time this occurs, the pointer will be reset to the original X coordinate, which is the desired behavior.

The frame buffer can be written at any time, including the active display period when the screen is being scanned and the frame buffer is being read out. Writes take priority over reads so undefined pixels will be output during such times, but these will quickly be refreshed during the next frame. Transferring extensive amounts of data to the frame buffer during the display period will cause noticeable flicker and artifacts.

4.4 Asynchronous Operation of the Input Port

The input port hardware interface is implemented as 8 data bus lines, 2 enable lines, and an input clock. The enable lines are taken off of the address decoder and are used to select which register to write. The input clock is the Z80 WR signal. Examining the timing in figure 1.6, we can see that WR goes low for a while during a write. This falling edge transition is used to detect when a write is occurring and is what allows the internal write pointer to be automatically incremented.

There was a long-standing bug in the initial VHDL code which caused memory writes to randomly fail dozens of times per frame. Pixels from the image would sometimes be misplaced or duplicated around random parts of the screen and would remain static, indicating that they had been committed to memory in the wrong locations. At first, the quality of the input signal was suspected, but was quickly ruled out after some testing. It was also observed that the misplaced pixels were always of the same colors used in the image and the write pointer never failed to be updated.

It turns out that the problem was caused by the asynchronous input signals interfering with the SRAM state machine. This caused data to sometimes be written to the wrong location (probably to where the refresh process was currently reading from.) The FPGA clock has no frequency or phase relation to the Z80 clock,

therefore, transitions on the input lines (including the input clock line) appear to be completely asynchronous to the FPGA logic. The presence of such signals in a synchronous circuit can yield unpredictable results because changes can occur at any point in the logic due to timing constraints not being obeyed. The solution is to latch the signals and synchronize them.

At first, I tried something similar to listing 4.1. The idea was to change a synchronized signal (or so I thought), `write_trigger`, each time data was detected on the input port. Then, a process operating on the 50 MHz clock would check to see if a change occurred and, if so, would trigger an SRAM write procedure.

Listing 4.1 Incorrect synchronization example.

```
process(clk_50mhz, input_clk)
begin
   if input_clk'event and input_clk = '0' then  -- falling edge
      -- Latch input data (not shown)
      write_ptr <= write_ptr + 1;
      write_trigger <= not write_trigger;       -- change this bit
   end if;

   if clk_50mhz'event and clk_50mhz = '1' then  -- 50 MHz process
      write_trigger_prev <= write_trigger;      -- save previous value
      if write_trigger /= write_trigger_prev then
         sram_do_write <= '1';                  -- change detected, begin write
      end if;
   end if;
end process;
```

This code fixed the problem of writes occurring at random locations but it introduced a new problem: Writes would sometimes be “dropped” – that is, the write procedure would never begin but the write pointer would be updated, indicating that the request should have been seen. A pixel would have to be written several times to guarantee that it had been set.

The reason for this is that the synchronization is not actually occurring. The part of the process dependent on `clk_50mhz` is again dealing with an asynchronous signal: `write_trigger`, the very signal that was introduced to solve the problem in the first place! Note that it is modified asynchronously according to the input clock; so we are not much better off. The solution is to latch (buffer) `write_trigger` and *then* use it, as in listing 4.2.

Listing 4.2 Correct synchronization example.

```
process(clk_50mhz, input_clk)
begin
   if input_clk'event and input_clk = '0' then  -- falling edge
      -- Latch input data (not shown)
      write_ptr <= write_ptr + 1;
      write_trigger <= not write_trigger;       -- change this bit
   end if;

   if clk_50mhz'event and clk_50mhz = '1' then  -- 50 MHz process
      write_trigger_buf <= write_trigger;       -- latch it
      write_trigger_prev <= write_trigger_buf;  -- save previous value
      if write_trigger_buf /= write_trigger_prev then
         sram_do_write <= '1';                  -- change detected, begin write
      end if;
   end if;
end process;
```

Now, the problem is solved.

In the BartStation, it is `input_clk` itself that is buffered and then the synchronized process monitors the status of that signal until a transition from 1 to 0 is observed. This allowed me to remove `input_clk` from the process sensitivity list altogether.

4.5 Interrupts

As soon as the last line of the display has been refreshed, the video controller enters the vertical blanking period during which the electron beam in a CRT monitor is being repositioned to the top of the screen. An interrupt is generated via a high-to-low transition on the Z80's NMI line. This occurs at 60 Hz and is useful for timing and updating graphics.

Interrupts can be enabled or disabled using the settings mode of the video controller. Initially, they ought to be disabled but it is always best to explicitly configure them.

5. Programming Guide

An overview of the BartStation's software environment is provided here along with examples of how to program the system. Complete demonstration programs can be found in **Appendix D**.

5.1 Memory Map

The complete memory map of the BartStation as seen by the Z80 is shown in table 5.1. All unspecified address ranges are undefined and should not be accessed. Furthermore, the directional constraints should be obeyed. Do not attempt to write to read-only addresses!

Table 5.1 Memory map.

Start Address	End	Size	Function	Direction
$0000	$3FFF	16 Kbytes	ROM	Read only
$8000	$8FFF	1 byte (mirrored)	Video Data Register	Write only
$9000	$9FFF	1 byte (mirrored)	Control Register	Write only
$A000	$AFFF	1 byte (mirrored)	Joypad Status Register	Read only
$F000	$F3FF	1 Kbyte	RAM	Read/Write
$F400	$FFFF	1 Kbyte (mirrored)	RAM (mirrored every 1KB)	Read/Write

The layout of the registers is described in the previous sections of this document in more detail.

Due to minimal decoding, some devices in the address space are "mirrored" repeatedly throughout different memory locations. In other words, the same device may be accessible through several different addresses. RAM, for example, is intended to be accessed between $F000 and $F3FF but it is also appears from $F400 onwards in 1KB intervals.

5.2 Memory Space, Stack, and Interrupts

This section describes the memory space, stack, and interrupts from the software's perspective.

Programs are stored in ROM, which is mapped from $0000 to $3FFF. Upon reset, the Z80 begins execution at $0000. The IRQ vector is at $0038 and the NMI is at $0066. For more details about the Z80, refer to its documentation.

At a minimum, we must define all of the vectors. It is also important to immediately set up a stack pointer so that NMIs may be processed without crashing the system. Because the stack grows down, a logical location to place the stack pointer would be at the top of the RAM space. Listing 5.1 shows a sample skeleton program.

Listing 5.1 Sample skeleton program demonstrating the bare minimum requirements for set-up.

```
;
; ResetVector:
;
; Execution begins here when the Z80 powers up.
;
.ORG $0
ResetVector:
   ld sp,$F400 ; set SP to the top of the RAM area
   di          ; disable IRQs just to be safe
   jp Start    ; jump to program entry point

;
; IRQVector:
;
```

```
; Does nothing. IRQs should never occur.
;
.ORG $38
IRQVector:
   reti

;
; NMIVector:
;
; Triggered at the start of each VBL period.
;
.ORG $66
NMIVector:
   reti        ; do nothing for now

;
; Start:
;
; Program entry point.
;
Start:
   jp Start    ; loop infinitely for now
```

An additional important step not shown here is to wait for at least 2 seconds for the FPGA to configure itself. None of my example programs in **Appendix D** do this, but they should. The FPGA loads its configuration from an on-board EEPROM and this takes some time. If the Z80 program accesses the video controller while this is occurring, the commands will have no effect. It is best to wait until the system is known to be ready.

Another way of detecting readiness is to continuously attempt to enable interrupts (described in the next section) and check that 2 or 3 interrupts have been generated. This should ensure the system is up and running.

5.3 Video Registers

The video registers are located at $8000 and $9000. They are described in more detail in section **4**. On power-up, the state of the video hardware is largely undefined. The display may be on or off. It is best to begin by turning the display off, clearing the random contents of the frame buffer, and then re-enabling the display. Listing 5.2 demonstrates how this may be accomplished.

Listing 5.2 Clearing the frame buffer with the display off.

```
ld    hl,$9000    ; address of Control Register
ld    (hl),4      ; settings mode
ld    a,0         ; disable interrupts and turn the display off
ld    ($8000),a
call  ClearScreen ; your subroutine to clear the screen; assume it saves HL
ld    (hl),4
ld    a,3         ; turn interrupts and the display on
ld    ($8000),a
```

Writing to the frame buffer is a multi-step process which is composed of setting up the internal write pointer and then transferring data to the Video Data Register. The pointer is automatically incremented and does not need to be specified again until a new location in the buffer is sought. Listing 5.3 demonstrates a routine for setting up the address.

Listing 5.3 Frame buffer address set up.

```
;
; SetFrameBufferAddress:
;
; Sets the frame buffer address to the desired X,Y coordinate.
```

```
;
; Inputs:
;  BC = X coordinate (9 bits)
;  D  = Y coordinate
;
SetFrameBufferAddress:
   ld hl,$9000    ; Control Register
   ld (hl),1      ; mode 1: X coordinate low
   ld ($8000),c   ; low 8 bits of X coordinate
   ld (hl),2      ; mode 2: X coordinate high
   ld ($8000),b   ; high bit of X coordinate
   ld (hl),3      ; mode 3: Y coordinate
   ld ($8000),d   ; Y coordinate
   ret
```

Writing pixels is then simply a matter of storing them to address $8000. Each time a write occurs, the internal pointer is incremented. If it exceeds the limits of the current line, it wraps to the next. If the lower border is exceeded, the results are undefined. Care should be taken to avoid writing outside of the 76,800 pixel region.

5.4 A Simple Image Blitter

Listing 5.4 demonstrates a simple but complete and usable image blitter which copies a rectangular image (stored as a linear 1-dimensional array of pixels, 1 byte each) to an arbitrary position in the frame buffer. Notice that only the Y component is updated after the address has been set up for the first time. The code takes advantage of the fact that when the Y position is re-loaded, the previously loaded X position will be used to calculate the frame buffer address.

Listing 5.4 Blitter routine.

```
;
; BlitImage:
;
; Copies an image from memory to the frame buffer.
;
; Input:
;  B  = X size.
;  C  = Y size.
;  D  = Destination X position.
;  E  = Destination Y position.
;  HL = Image source address.
;
; Saves AF, BC, DE, HL.
;
BlitImage:
   push  af
   push  bc
   push  de
   push  hl

   push  hl
   push  bc
   ld    bc,$8000 ; data register
   ld    hl,$9000 ; command register
   ld    (hl),2   ; set X position high to 0
   ld    a,0
   ld    (bc),a
   ld    (hl),1   ; set X position low
   ld    a,d
   ld    (bc),a
   pop   bc
   pop   hl

yloop:
   ld    a,3      ; set Y position
```

```
   ld    ($9000),a
   ld    a,e
   ld    ($8000),a

   ld    a,0   ; data write mode
   ld    ($9000),a
   push  bc    ; save X size

xloop:
   ld    a,(hl)      ; fetch pixel
   inc   hl
   ld    ($8000),a   ; write to framebuffer
   dec   b           ; decrement X count and loop
   ld    a,b
   cp    0
   jr    nz,xloop

   pop   bc
   inc   e  ; next line in framebuffer
   dec   c  ; decrement Y count and loop
   ld    a,c
   cp    0
   jr    nz,yloop

   pop   hl
   pop   de
   pop   bc
   pop   af
   ret
```

5.5 Joypad

Information on how the joypad works can be found in section **2**. Included there are procedures for how to poll both 3- and 6-button joypads. Listing 5.5 is a routine to read a 6-button joypad. The code for reading a 3-button pad would only include the first two reads from the status register as the rest are 6-button-specific.

Listing 5.5 Routine to read a 6-button joypad.

```
;
; ReadJoypad:
;
; Reads all 6 buttons.
;
; Output:
;  DE = 0000 ZYXM UDLR BCAS (M=Mode, S=Start)
;
; Preserves all registers. Video mode ends up as 0.
;
ReadJoypad:
   push  af ; save used registers
   push  bc
   push  hl

   ld    hl,$9000 ; Control Register
   ld    bc,$A000 ; Joypad Status Register

   ld    (hl),8   ; MUX=1
   ld    a,(bc)   ; A=--UDLRBC
   sla   a
   sla   a
   ld    e,a      ; E=UDLRBC00
   ld    (hl),0   ; MUX=0
   ld    a,(bc)   ; A=--UD00AS
   and   3        ; A=000000AS
   or    e        ; A=UDLRBCAS
   ld    e,a      ; E=UDLRBCAS
```

```
ld    (hl),8   ; MUX=1
nop
ld    (hl),0   ; MUX=0
nop

ld    (hl),8   ; MUX=1
ld    a,(bc)   ; A=--ZYXMBC
srl   a
srl   a
and   $F       ; A=0000ZYXM
ld    d,a      ; D=0000ZYXM

ld    (hl),0   ; MUX=0

pop   hl       ; restore registers
pop   bc
pop   af
ret
```

An important precaution is to avoid writing the joypad multiplex bit, located in the Control Register, during an interrupt. This is because the mode bits for the video controller's input port will also be altered which can disrupt video code that may have been executing when the interrupt occurred. It is best to poll the joypads outside of interrupt routines.

6. Conclusion

This project was a success and has taught me a lot about the practical issues involved in the design of simple computer systems using 8-bit microprocessors and FPGAs. Three particular lessons learned were how to deal with asynchronous signals in a synchronous design, an issue I expect to face again in future projects, how to interface 5V logic to an LVCMOS FPGA, and how to design digital circuits in VHDL which can output a VGA signal.

I hope that readers of this documentation will find it helpful and will be encouraged to pursue similar projects. The most expensive components of this project were the FPGA board and the device programmer used to program the flash memory chips. Suitable FPGA boards can be found for as low as $100 at the time of this writing, which is what the Spartan-3 Starter Kit Board costs, but device programmers are typically more expensive. It is feasible to construct one for the particular chips used in this design, however. In my case, the university's EE department had one available for me to use.

Should you attempt a similar project: Good luck and have fun!

7. References

The following documents were used in writing this report and designing the project.

"Z80 Family CPU User Manual"; UM008005-0205; Feb. 5 2005; ZiLOG Inc.

"CAT28F512 512K-Bit CMOS Flash Memory"; Doc. No. 1084, Rev. H; July 2, 2004; Catalyst Semiconductor, Inc.

"HY6116 2048x8-Bit CMOS Static RAM"; February 1986; Hyundai Semiconductor

"74HC/HCT154 4-to-16 line decoder/demultiplexer"; September 1993; Koninklijke Philips Electronics N.V.

"74HC08; 74HCT08 Quad 2-input AND gate"; July 25, 2003; Koninklijke Philips Electronics N.V.

"74HC/HCT573 Octal D-type transparent latch; 3-state"; December 1990; Koninklijke Philips Electronics N.V.

"74H14; 74HCT14 Hex inverting Schmitt trigger"; October 30, 2003; Koninklijke Philips Electronics N.V.

"Sega Genesis Controller (joystick) pinout"; June 25, 2005; GET WIRED! at pinouts.ru; http://pinouts.ru/data/genesiscontroller_pinout.shtml

"Sega Genesis hardware notes"; Version 0.8; February 1, 2003; Charles MacDonald; http://cgfm2.emuviews.com

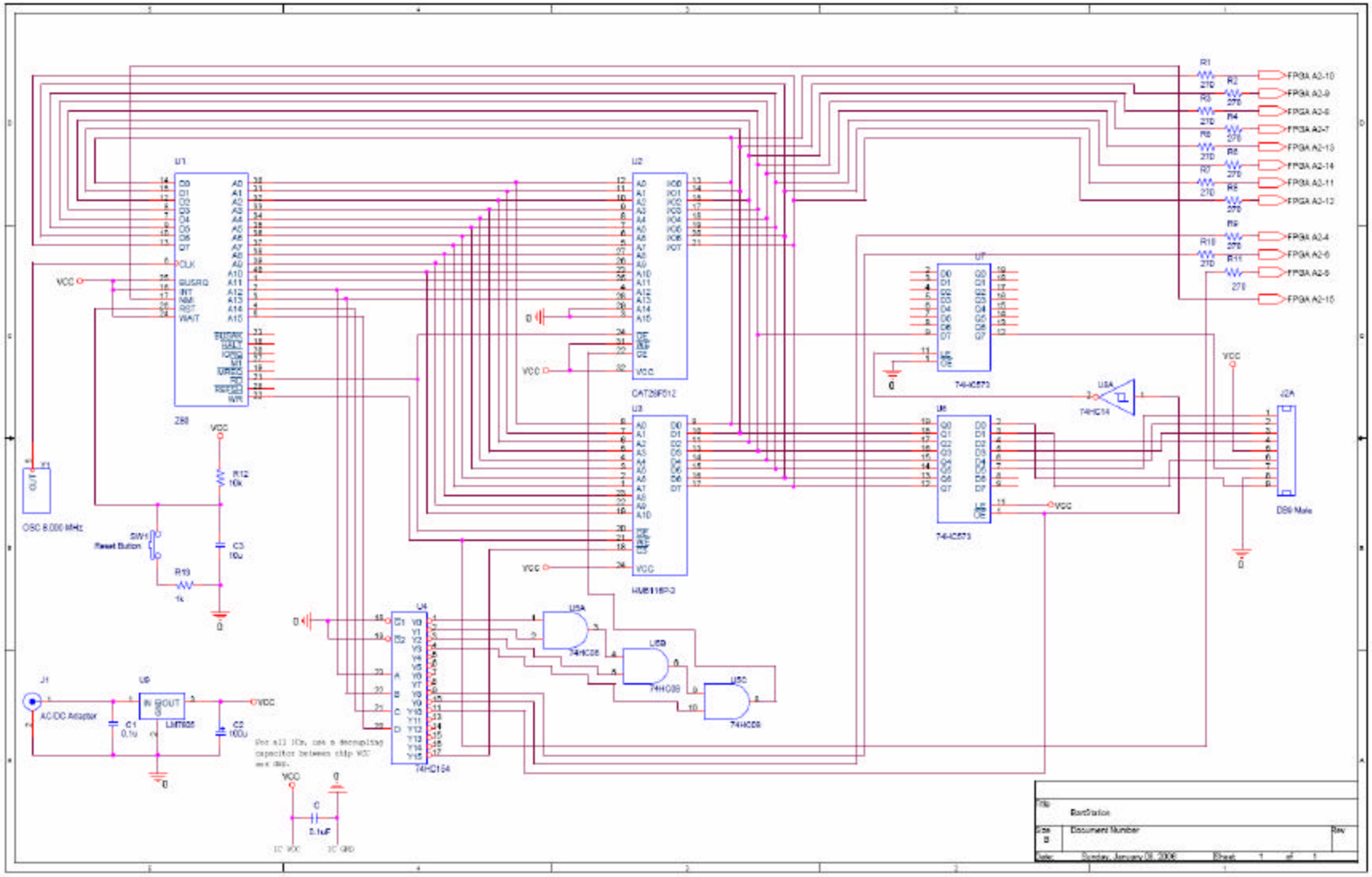
"Are the Spartan-3/-3E I/Os 5V-tolerant? Can I drive I/O with a higher voltage?"; Answer Record #19146; July 14, 2005; Xilinx, Inc.; http://www.xilinx.com/xlnx/xil_ans_display.jsp?BV_UseBVCookie=yes&getPagePath=19146

"Spartan-3 FPGA Family: Complete Data Sheet"; DS099; August 19, 2005; Xilinx, Inc.

"Spartan-3 Starter Kit Board User Guide"; UG130 (v1.1); May 13, 2005; Xilinx, Inc.

Appendix A: Schematic of Z80 Circuit

The following page contains a schematic of the Z80 circuit. Unfortunately, the schematic resolution is to high to fit in this document, so the result is not completely readable. Please refer to a separate schematic.


U1
Z80
U2
CAT28F512
U3
HM6116P-3
U4
74HC154
U5A
74HC08
U5B
74HC08
U5C
74HC08
U6
74HC573
U7
74HC573
U8A
74HC04
J2A
DB9 Male
J1
AC/DC Adaptor
U9
LM7805
OSC 8.000 MHz
SW1
Reset Button
R12
10k
R13
1k
C1
0.1u
C2
100u
C3
10u
VCC
0
FPGA A2-10
FPGA A2-9
FPGA A2-8
FPGA A2-7
FPGA A2-13
FPGA A2-14
FPGA A2-11
FPGA A2-12
FPGA A2-4
FPGA A2-6
FPGA A2-5
FPGA A2-15
BusStation
Document Number
Sunday, January 08, 2006

Appendix B: Parts

The parts I used to construct my prototype are listed in table B.1 along with approximate cost, when known.

Table B.1 Part list.

Quantity	Part	Total Price	Supplier
1	Z80 CPU, 8MHz (Z84C0008PEC)	$3.05	Jameco
1	Flash memory, 64KB (CAT28F512)	$5.25	Jameco
1	SRAM, 2KB (6116P-3)	$1.49	Jameco
1[1]	74HC154	$0.75	Jameco
1[2]	74HC08	$1.58	Jameco
2	74HC573	$0.62	Jameco
1	74HC14	$0.22	Jameco
1	8.000 MHz TTL oscillator, half can	$1.32	Jameco
1	7805T, TO-220 package	$0.35	Jameco
12 (100 pc. bundle)	270 ohm or 300 ohm ¼W resistor	$0.99	Jameco
1 (100 pc. bundle)	10 K-ohm resistor	$0.99	Jameco
1 (100 pc. bundle)	1 K-ohm resistor	$0.99	Jameco
9[2]	0.1 µF ceramic disc capacitor, 25V, 20%	$1.40	Jameco
1[2]	100 µF radial electrolytic capacitor, 50V, 20%	$0.70	Jameco
1[2]	10 µF radial electrolytic capacitor, 50V, 20%	$0.60	Jameco
1[2]	Tactile switch[3], two-pin, 5.0mm	$5.20	Jameco
1	Male DC power jack, 2.1mm	$0.49	Jameco
1	Male DB-9 connector, solder cup	$0.56	Jameco
1	26 AWG solid wire[4]	?	?
1	MPJA 3220 T/P 4B/P breadboard (stock no. 4447 TE)	$22.95	MPJA
1	AC-DC wall transformer, 12VDC, 2.1mm female	$10.95	Jameco
1	Spartan-3 Starter Kit Board	$99.00	Xilinx

The total price is approximately $159.45 and does not include the 26 AWG wire, taxes, or shipping.

The vendor web sites are:

Jameco: http://www.jameco.com
MPJA: http://www.mpja.com
Xilinx: http://www.xilinx.com

The Spartan-3 Starter Kit Board may also be obtained from Digilent (http://www.digilent.com)

[1] The 74HC154 comes in both 0.3''-wide and 0.6'' DIP configurations. The 0.3'' model is recommended because it occupies less space (it is the same width as the other 74HC parts used.)

[2] Indicates this part is only available in some minimum quantity. The price reflects this.

[3] I already had tactile switches available to me. I have not purchased this listed switch from Jameco, so take care to ensure it will work with the design. It ought to be mountable on a breadboard (if you choose to implement the circuit that way) and in its default state should be off (the connection will be broken.)

[4] I obtained 26 AWG solid wire from work. It's very useful for wiring the various buses and signals because it is so thin and flexible. If you decide to purchase it, make sure the insulation is not so thick as to make it the same diameter as breadboard hook-up wire. It should be noticeably thinner.

Appendix C: Video Controller VHDL Code and Constraints File

The subsequent pages contain the two files which define the video controller. The first, vga.vhd, is the video controller VHDL code. The second, vga.ucf, is a Xilinx constraint file which defines pin assignments.

```
--
-- BartStation Video Controller
-- by Bart Trzynadlowski, 2005-2006
--
-- A video controller designed for use with the Spartan-3 Starter Kit Board for
-- the BartStation homebrew video game system project.
--

--
-- vga.vhd
--
-- Implementation of the VGA video controller.
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;   -- Xilinx primitives
use UNISIM.VComponents.all;

--
-- VGA
--
-- Defines the VGA controller interface.
--
entity VGA is
   port
   (
      --
      -- 50MHz Clock
      --
      clk_50mhz:  in std_logic;

      --
      -- VGA Signals
      --
      vga_rgb: out std_logic_vector(2 downto 0);
      vga_hs:  out std_logic;
      vga_vs:  out std_logic;
      vga_vbl: out std_logic;

      --
      -- SRAM Pins
      --
      -- These are mapped to the actual FPGA pins connected to the SRAM IC.
      --
      sram_pins_addr:out std_logic_vector(17 downto 0);
      sram_pins_io:  inout std_logic_vector(7 downto 0);
      sram_pin_oe:   out std_logic;
      sram_pin_we:   out std_logic;
      sram_pin_ce:   out std_logic;
      sram_pin_ce2:  out std_logic;
      sram_pin_ub:   out std_logic;
      sram_pin_lb:   out std_logic;

      --
      -- Input Port
      --
      -- input(10):  Control Register Enable. Active low.
      -- input(9):   Video Data Register Enable. Active low.
      -- input(8):   Input Clock. Writes are processed on falling edge.
      -- input(7):   D7
      -- input(6):   D6
      -- input(5):   D5
      -- input(4):   D4
      -- input(3):   D3
      -- input(2):   D2 (R)
      -- input(1):   D1 (G)
      -- input(0):   D0 (B)
```

```
      --
      input_pins: in std_logic_vector(10 downto 0)
   );
end VGA;

--
-- VGA: Behavioral Architecture
--
-- Implementation of the VGA controller.
--
architecture Behavioral of VGA is
   --
   -- VGA Signals
   --
   -- A 25MHz clock is generated for VGA timing.
   --
   signal clk_25mhz:          std_logic;
   signal horizontal_counter: std_logic_vector(9 downto 0);
   signal vertical_counter:   std_logic_vector(9 downto 0);
   signal framebuf_addr:      std_logic_vector(18 downto 0);
   signal framebuf_line_addr: std_logic_vector(18 downto 0);
   signal display_enable:     std_logic;
   signal interrupt_enable:   std_logic := '0';

   --
   -- Inport Port Signals
   --
   signal input:              std_logic_vector(10 downto 0);
   signal input_clk:          std_logic := '0';
   signal input_clk_prev:     std_logic := '0';

   signal mode:               std_logic_vector(2 downto 0);

   signal addr_x:             std_logic_vector(8 downto 0);
   signal addr_y:             std_logic_vector(7 downto 0);
   signal write_addr:         std_logic_vector(17 downto 0);

   --
   -- SRAM Signals and State Management
   --
   signal sram_addr:    std_logic_vector(17 downto 0);
   signal sram_write_addr: std_logic_vector(17 downto 0);
   signal sram_read_addr:  std_logic_vector(17 downto 0);
   signal sram_data_write: std_logic_vector(7 downto 0);
   signal sram_data_read:  std_logic_vector(7 downto 0);
   signal sram_ce:      std_logic;
   signal sram_ce2:     std_logic;
   signal sram_oe:      std_logic;
   signal sram_we:      std_logic;
   signal sram_ub:      std_logic;
   signal sram_lb:      std_logic;
   signal sram_io_t:    std_logic;

   type sram_state_t is
   (
      SRAM_WAITING,  -- wait for command, place data on bus
      SRAM_WRITE,    -- complete a write transaction
      SRAM_READ      -- complete a read transaction
   );
   signal sram_state:      sram_state_t := SRAM_WAITING;

   signal sram_do_read:    std_logic := '0';
   signal sram_do_write:   std_logic := '0';

   signal sram_data_read_buf:    std_logic_vector(7 downto 0);
   signal sram_data_write_buf:   std_logic_vector(7 downto 0);
begin
   --
   -- Map SRAM pins to Xilinx IO buffers
   --
   g1:
```

```
   for i in sram_pins_io'range generate
      iod0: iobuf port map
      (
         i => sram_data_write(i),
         io => sram_pins_io(i),
         o => sram_data_read(i),
         t => sram_io_t   -- 0 read?
      );
   end generate g1;
g2:
   for i in sram_pins_addr'range generate
      oa0: obuf port map
      (
         o => sram_pins_addr(i),
         i => sram_addr(i)
      );
   end generate g2;
we: obuf port map
   (
      o => sram_pin_we,
      i => sram_we
   );

oe: obuf port map
   (
      o => sram_pin_oe,
      i => sram_oe
   );
ce: obuf port map
   (
      o => sram_pin_ce,
      i => sram_ce
   );
ce2: obuf port map
(
   o => sram_pin_ce2,
   i => sram_ce2
);
ub: obuf port map
   (
      o => sram_pin_ub,
      i => sram_ub
   );
lb: obuf port map
   (
      o => sram_pin_lb,
      i => sram_lb
   );

inp:
   for i in input_pins'range generate
   np0: ibuf port map
      (
         i => input_pins(i),
         o => input(i)
      );
   end generate inp;

--
-- Fixed signals
--
sram_ce2 <= '1';  -- make sure second SRAM chip stays off the bus
sram_lb <= '0';
sram_ub <= '1';

--
-- Generate the 25MHz clock
--
process(clk_50mhz)
begin
   if clk_50mhz'event and clk_50mhz = '1' then
```

```
         clk_25mhz <= (clk_25mhz xor '1');
      end if;
   end process;

   --
   -- Input Port State Machine
   --
   process(clk_50mhz)
   begin
      if clk_50mhz'event and clk_50mhz = '1' then
         input_clk <= input(8);        -- latch the asynchronous input clock
         input_clk_prev <= input_clk;  -- save the previous latched clock value


         if input_clk_prev = '1' and input_clk = '0' then   -- falling edge
            if input(10) = '0' then                -- Control Register enabled
               mode <= input(2 downto 0);          -- mode bits
            elsif input(9) = '0' then              -- Video Data Register enabled
               case mode is
               when "000" =>  -- data write mode
                  sram_data_write_buf(2 downto 0) <= input(2 downto 0);
                  sram_write_addr <= write_addr;
                  write_addr <= write_addr + 1;
                  sram_do_write <= '1';
               when "001" =>  -- address X low
                  addr_x <= addr_x(8) & input(7 downto 0);
                  write_addr <= ("00" & addr_y & "00000000") + ("0000" & addr_y &
"000000") + (addr_x(8) & input(7 downto 0));
               when "010" =>  -- address X high
                  addr_x(8) <= input(0);
                  write_addr <= ("00" & addr_y & "00000000") + ("0000" & addr_y &
"000000") + (input(0) & addr_x(7 downto 0));
               when "011" =>  -- address Y
                  addr_y <= input(7 downto 0);
                  write_addr <= ("00" & input(7 downto 0) & "00000000") + ("0000" &
input(7 downto 0) & "000000") + addr_x;
               when "100" =>  -- settings
                  display_enable <= input(0);
                  interrupt_enable <= input(1);
               when others =>
               end case;
            end if;
         end if;

         --
         -- When a write has completed, we must lower the write signal
         --
         if sram_state = SRAM_WRITE then
            sram_do_write <= '0';
         end if;
      end if;
   end process;


   --
   -- SRAM State Machine
   --
   process(clk_50mhz)
   begin
      sram_oe <= '0';

      if clk_50mhz'event and clk_50mhz = '1' then
         case sram_state is
         when SRAM_WAITING =>
            if sram_do_write = '1' then   -- write requested
               sram_addr <= sram_write_addr;
               sram_data_write <= sram_data_write_buf;
               sram_ce <= '0';
               sram_we <= '0';
               sram_io_t <= '0';
               sram_state <= SRAM_WRITE;
```

```
            elsif sram_do_read = '1' then -- read requested
               sram_addr <= sram_read_addr;
               sram_ce <= '0';
               sram_we <= '1';
               sram_io_t <= '1';
               sram_state <= SRAM_READ;
            end if;
         when SRAM_READ =>
            sram_ce <= '1';
            sram_we <= '1';
            sram_io_t <= '0';
            sram_data_read_buf <= sram_data_read;
            sram_state <= SRAM_WAITING;
         when SRAM_WRITE =>
            sram_ce <= '1';
            sram_we <= '1';
            sram_io_t <= '0';
            sram_state <= SRAM_WAITING;
         end case;
      end if;
   end process;

   --
   -- VGA Output
   --
   process(clk_25mhz)
   begin
      if clk_25mhz'event and clk_25mhz = '1' then
         --
         -- When starting a new line, save the frame buffer address and only
         -- update it every other line to halve the vertical resolution. Note
         -- that the display begins at V = 39 so therefore, every ODD line
         -- will have the frame buffer updated and saved and the even line will
         -- restart there
         --
         if (horizontal_counter = "0000000000") then
            if (vertical_counter >= "0000000000") and (vertical_counter <= "0000100111")
then  -- 39
               framebuf_addr <= "0000000000000000000";
               framebuf_line_addr <= "0000000000000000000";
            else
               --
               -- On odd lines, advance frame buffer by 320 pixels
               --
               if vertical_counter(0) = '1' then   -- odd lines
                  framebuf_line_addr <= framebuf_line_addr + "0000000000101000000";
                  framebuf_addr <= framebuf_line_addr + "0000000000101000000";
               else
                  framebuf_addr <= framebuf_line_addr;
               end if;
            end if;
         end if;


         --
         -- If the horizontal counter is between 144 and 783, we are within the
         -- 640 pixel range of the visible part of the scanline. If the
         -- vertical counter is between 39 and 518, we are in the visible part
         -- of the screen (480 lines.)
         --
         if (horizontal_counter >= "0010010000")      -- 144
            and (horizontal_counter < "1100010000")      -- 784
            and (vertical_counter >= "0000100111")       -- 39
            and (vertical_counter < "1000000111") then   -- 519

            --
            -- Fetch the current pixel from SRAM (2 50MHz clock cycles) and
            -- display it.
            --

            if display_enable = '1' then
```

```
            sram_read_addr <= framebuf_addr(17 downto 0);
            sram_do_read <= '1';
            vga_rgb <= sram_data_read_buf(2 downto 0);
         else
            sram_do_read <= '0';
            vga_rgb <= "000";
         end if;

         --
         -- Only advance every other pixel to halve the resolution
         --
         if horizontal_counter(0) = '1' then
            framebuf_addr <= framebuf_addr + 1;
         end if;
      else
         --
         -- We are outside of the visible region, output black
         --
         vga_rgb <= "000";
         sram_do_read <= '0';
      end if;

      if (vertical_counter < "0000100111") or (vertical_counter >= "1000000111") then
         if interrupt_enable = '1' then
            vga_vbl <= '0';   -- active low
         else
            vga_vbl <= '1';
         end if;
      else
         vga_vbl <= '1';
      end if;

      --
      -- Within this horizontal range, the H-sync signal must be low
      --
      if (horizontal_counter > "0000000000")
         and (horizontal_counter < "0001100001") then -- 96+1
         vga_hs <= '0';
      else
         vga_hs <= '1';
      end if;

      --
      -- Generate the V-sync signal in the appropriate range
      --
      if (vertical_counter > "0000000000")
         and (vertical_counter < "0000000011") then   -- 2+1
         vga_vs <= '0';
      else
         vga_vs <= '1';
      end if;

      --
      -- Increment the horizontal and vertical counters and reset them when
      -- starting a new line.
      --
      horizontal_counter <= horizontal_counter+1;
      if (horizontal_counter = "1100100000") then  -- 800
         vertical_counter <= vertical_counter+1;
         horizontal_counter <= "0000000000";
      end if;

      if (vertical_counter="1000001001") then      -- 521
         vertical_counter <= "0000000000";
      end if;

   end if;
 end process;

end Behavioral;
```

```
#
# BartStation Video Controller
# by Bart Trzynadlowski, 2005-2006
#
# A video controller designed for use with the Spartan-3 Starter Kit Board for
# the BartStation homebrew video game system project.
#

#
# vga.ucf
#
# Constraints for Spartan-3 Starter Kit Board (pin assignments.)
#

NET "clk_50mhz"  LOC = "t9"  ;
NET "vga_rgb<2>"  LOC = "r12"  ;
NET "vga_rgb<1>"  LOC = "t12"  ;
NET "vga_rgb<0>"  LOC = "r11"  ;
NET "vga_hs"  LOC = "r9"  ;
NET "vga_vs"  LOC = "t10"  ;
NET "vga_vbl" LOC = "d10";          # 15
NET "sram_pins_addr<0>"  LOC = "l5"  ;
NET "sram_pins_addr<10>"  LOC = "g5"  ;
NET "sram_pins_addr<11>"  LOC = "h3"  ;
NET "sram_pins_addr<12>"  LOC = "h4"  ;
NET "sram_pins_addr<13>"  LOC = "j4"  ;
NET "sram_pins_addr<14>"  LOC = "j3"  ;
NET "sram_pins_addr<15>"  LOC = "k3"  ;
NET "sram_pins_addr<16>"  LOC = "k5"  ;
NET "sram_pins_addr<17>"  LOC = "l3"  ;
NET "sram_pins_addr<1>"  LOC = "n3"  ;
NET "sram_pins_addr<2>"  LOC = "m4"  ;
NET "sram_pins_addr<3>"  LOC = "m3"  ;
NET "sram_pins_addr<4>"  LOC = "l4"  ;
NET "sram_pins_addr<5>"  LOC = "g4"  ;
NET "sram_pins_addr<6>"  LOC = "f3"  ;
NET "sram_pins_addr<7>"  LOC = "f4"  ;
NET "sram_pins_addr<8>"  LOC = "e3"  ;
NET "sram_pins_addr<9>"  LOC = "e4"  ;
NET "sram_pin_ce"  LOC = "p7"  ;
NET "sram_pin_ce2"  LOC = "n5";
NET "sram_pins_io<0>"  LOC = "n7"  ;
NET "sram_pins_io<1>"  LOC = "t8"  ;
NET "sram_pins_io<2>"  LOC = "r6"  ;
NET "sram_pins_io<3>"  LOC = "t5"  ;
NET "sram_pin_lb"  LOC = "p6"  ;
NET "sram_pin_oe"  LOC = "k4"  ;
NET "sram_pin_ub"  LOC = "t4"  ;
NET "sram_pin_we"  LOC = "g3"  ;
NET "input_pins<0>"  LOC = "c7"; # 10
NET "input_pins<1>"  LOC = "e7"; # 9
NET "input_pins<2>"  LOC = "c6"; # 8
NET "input_pins<3>"  LOC = "d6"; # 7
NET "input_pins<4>"  LOC = "d8"; # 13
NET "input_pins<5>"  LOC = "c9"; # 14
NET "input_pins<6>"  LOC = "d7"; # 11
NET "input_pins<7>"  LOC = "c8"; # 12
NET "input_pins<8>"  LOC = "d5"; # 5 -- WR CLK
NET "input_pins<9>"  LOC = "c5";    # 6 -- REG2 (data) Enable
NET "input_pins<10>"    LOC = "e6"; # 4 -- REG1 (command) Enable
```

Appendix D: Sample Programs

Two complete sample programs are included here: A 6-button joypad tester and an animated sprite demo which allows the player to move a character left and right on the screen. The sprite consists of 8 frames (figure D.1) of animation and was ripped from id Software's classic PC game, "Commander Keen: Invasion of the Vorticons."

Figure D.1 "Garg" sprites for the playable "Garg Demo."

To assemble these programs, use TASM, the table assembler by Speech Technologies, Inc. It can be found freely on the web. The "Garg Demo" requires that the sprites be converted to raw binary format and appended to the binary output that TASM produces.

```
;
; Joypad Tester
; by Bart Trzynadlowski
; December 20, 2005 and January 7, 2006
;

;
; joypad.asm
;
; Reads the 6-button joypad and displays the button status on the screen.
;


;*******************************************************************************
; Vector Table
;*******************************************************************************

;
; ResetVector:
;
; Execution begins here when the Z80 powers up.
;

.ORG $0
ResetVector:
   di             ; disable interrupts
   ld sp,$F400    ; set SP to the top of the RAM area
   jp Start

;
; InterruptVector:
; NMIVector:
;
; Do nothing.
;

.ORG $38
InterruptVector:
   reti

.ORG $66
NMIVector:
   reti

;*******************************************************************************
; Main Program
;*******************************************************************************

;
; Start:
;
; Program entry point.
;

Start:
   ;
   ; Clear the screen
   ;
   ld    hl,$9000    ; control reg
   ld    (hl),4      ; settings mode
   ld    a,0
   ld    ($8000),a   ; disable interrupts and turn screen off
   call  ClearScreen
   ld    (hl),4
   ld    a,1
   ld    ($8000),a   ; turn display on

   ;
   ; Main Loop:
   ;
   ; - Read joypad
```

```
   ; - Draw appropriate sprites
   ; - Loop forever.
   ;
Loop:
   ;
   ; Read joypad buttons into DE
   ;
   call  ReadJoypad

   ;
   ; Draw the correct sprites. Sprite placement is (relative to some base
   ; X,Y coordinates):
   ;
   ; Up     5,0
   ; Down   5,10
   ; Left   0,5
   ; Right  10,5
   ; Start  20,5
   ; Mode   25,5
   ; A      35,10
   ; B      40,10
   ; C      45,10
   ; X      35,0
   ; Y      40,0
   ; Z      45,0
   ;
   push  de       ; save joypad buttons stored in D
   ld    a,e      ; load UDLRBCAS into A

   ld    b,5      ; set up size of sprite -- same for all!
   ld    c,5

   ld    d,100+20
   ld    e,100+5
   ld    hl,spr_start
   call  DrawButtonStatus
   ld    d,100+35
   ld    e,100+10
   ld    hl,spr_a
   call  DrawButtonStatus
   ld    d,100+45
   ld    e,100+10
   ld    hl,spr_c
   call  DrawButtonStatus
   ld    d,100+40
   ld    e,100+10
   ld    hl,spr_b
   call  DrawButtonStatus
   ld    d,100+10
   ld    e,100+5
   ld    hl,spr_right
   call  DrawButtonStatus
   ld    d,100+0
   ld    e,100+5
   ld    hl,spr_left
   call  DrawButtonStatus
   ld    d,100+5
   ld    e,100+10
   ld    hl,spr_down
   call  DrawButtonStatus
   ld    d,100+5
   ld    e,100+0
   ld    hl,spr_up
   call  DrawButtonStatus

   pop   de          ; retrieve remaining buttons in D (ZYXM)
   ld    a,d         ; transfer them to A
   ld    b,5         ; size of sprite
   ld    c,5
   ld    d,100+25
   ld    e,100+5
```

```
   ld    hl,spr_mode
   call  DrawButtonStatus
   ld    d,100+35
   ld    e,100+0
   ld    hl,spr_x
   call  DrawButtonStatus
   ld    d,100+40
   ld    e,100+0
   ld    hl,spr_y
   call  DrawButtonStatus
   ld    d,100+45
   ld    e,100+0
   ld    hl,spr_z
   call  DrawButtonStatus

   jp    Loop

;
; ReadJoypad:
;
; Reads all 6 buttons.
;
; Output:
;  DE = 0000 ZYXM UDLR BCAS (M=Mode, S=Start)
;
; Preserves all registers. Video mode ends up as 0.
;
ReadJoypad:
   push  af          ; save used registers
   push  bc
   push  hl

   ld    hl,$9000    ; Control Register
   ld    bc,$A000    ; Joypad Status Register

   ld    (hl),8      ; MUX=1
   ld    a,(bc)      ; A=--UDLRBC
   sla   a
   sla   a
   ld    e,a         ; E=UDLRBC00
   ld    (hl),0      ; MUX=0
   ld    a,(bc)      ; A=--UD00AS
   and   3           ; A=000000AS
   or    e           ; A=UDLRBCAS
   ld    e,a         ; E=UDLRBCAS

   ld    (hl),8      ; MUX=1
   nop
   ld    (hl),0      ; MUX=0
   nop

   ld    (hl),8      ; MUX=1
   ld    a,(bc)      ; A=--ZYXMBC
   srl   a
   srl   a
   and   $F          ; A=0000ZYXM
   ld    d,a         ; D=0000ZYXM

   ld    (hl),0      ; MUX=0

   pop   hl          ; restore registers
   pop   bc
   pop   af
   ret

;
; DrawButtonStatus:
;
; Shifts in the next button from A. Assumes that B, C, D, and E have been set
; up for DrawSprite() appropriately. HL must be set to the button sprite.
;
```

```
DrawButtonStatus:
   rrca                    ; test button
   jr    nc,button_pressed ; if pressed, use supplied sprite in HL
   ld    hl,spr_blank      ; otherwise, blank it out
button_pressed:
   call  DrawSprite
   ret

;
; Sprites (5x5)
;

spr_blank:
   .db 0,0,0,0,0
   .db 0,0,0,0,0
   .db 0,0,0,0,0
   .db 0,0,0,0,0
   .db 0,0,0,0,0
spr_left:
   .db 0,0,7,0,0
   .db 0,7,0,0,0
   .db 7,7,7,7,0
   .db 0,7,0,0,0
   .db 0,0,7,0,0
spr_right:
   .db 0,0,7,0,0
   .db 0,0,0,7,0
   .db 0,7,7,7,7
   .db 0,0,0,7,0
   .db 0,0,7,0,0
spr_up:
   .db 0,0,7,0,0
   .db 0,7,7,7,0
   .db 7,0,7,0,7
   .db 0,0,7,0,0
   .db 0,0,0,0,0
spr_down:
   .db 0,0,0,0,0
   .db 0,0,7,0,0
   .db 7,0,7,0,7
   .db 0,7,7,7,0
   .db 0,0,7,0,0
spr_a:
   .db 0,0,0,0,0
   .db 0,7,7,0,0
   .db 7,0,0,7,0
   .db 7,0,0,7,0
   .db 0,7,7,0,7
spr_b:
   .db 7,0,0,0,0
   .db 7,7,7,0,0
   .db 7,0,0,7,0
   .db 7,0,0,7,0
   .db 7,7,7,0,0
spr_c:
   .db 0,0,0,0,0
   .db 0,7,7,7,0
   .db 7,0,0,0,0
   .db 7,0,0,0,0
   .db 0,7,7,7,0
spr_start:
   .db 0,0,7,7,0
   .db 0,7,0,0,0
   .db 0,7,7,7,0
   .db 0,0,0,7,0
   .db 0,7,7,7,0
spr_x:
   .db 7,0,0,0,7
   .db 0,7,0,7,0
   .db 0,0,7,0,0
   .db 0,7,0,7,0
```

```
   .db 7,0,0,0,7
spr_y:
   .db 7,0,0,0,7
   .db 0,7,0,7,0
   .db 0,0,7,0,0
   .db 0,0,7,0,0
   .db 0,0,7,0,0
spr_z:
   .db 7,7,7,7,7
   .db 0,0,0,7,0
   .db 0,0,7,0,0
   .db 0,7,0,0,0
   .db 7,7,7,7,0
spr_mode:
   .db 7,7,0,7,7
   .db 7,0,7,0,7
   .db 7,0,7,0,7
   .db 7,0,0,0,7
   .db 7,0,0,0,7


;*****************************************************************************
; Video Functions
;*****************************************************************************

;
; DrawSprite:
;
; Draws a sprite on the screen.
;
; Input:
;  B  = X size.
;  C  = Y size.
;  D  = X position.
;  E  = Y position.
;  HL = Sprite address.
;
; Saves AF, BC, DE, HL.
;
DrawSprite:
   push  af
   push  bc
   push  de
   push  hl

   push  hl
   push  bc
   ld    bc,$8000 ; data register
   ld    hl,$9000 ; command register
   ld    (hl),2   ; set X position high to 0
   ld    a,0
   ld    (bc),a
   ld    (hl),1   ; set X position low
   ld    a,d
   ld    (bc),a
   pop   bc
   pop   hl

yloop:
   ld    a,3      ; set Y position
   ld    ($9000),a
   ld    a,e
   ld    ($8000),a

   ld    a,0      ; data write mode
   ld    ($9000),a
   push  bc       ; save X size

xloop:
   ld    a,(hl)   ; fetch pixel
   inc   hl
```

```
   ld    ($8000),a   ; write to framebuffer
   dec   b           ; decrement X count and loop
   ld    a,b
   cp    0
   jr    nz,xloop

   pop   bc
   inc   e           ; next line in framebuffer
   dec   c           ; decrement Y count and loop
   ld    a,c
   cp    0
   jr    nz,yloop

   pop   hl
   pop   de
   pop   bc
   pop   af
   ret

;
; ClearScreen:
;
; Clears the screen to black. Saves BC, HL, DE and destroys everything else.
;
ClearScreen:
   push  bc
   push  hl
   push  de

   ld    a,1
   ld    ($9000),a   ; set address X low
   ld    a,0
   ld    ($8000),a
   ld    a,2         ; set address X high
   ld    ($9000),a
   ld    a,0
   ld    ($8000),a

   ld    b,239       ; 240 lines, start clearing at last one
   ld    de,1
ClearScreen_yloop:
   ld    a,3
   ld    ($9000),a   ; set address Y
   ld    a,b
   ld    ($8000),a
   ld    a,0         ; data mode
   ld    ($9000),a

   ld    hl,320   ; 320 pixels
ClearScreen_xloop:
   ld    ($8000),a   ; write black pixel
   scf               ; clear carry
   ccf
   sbc   hl,de       ; X loop
   jr    nz,ClearScreen_xloop

   ld    a,b      ; if we cleared line 0, quit
   cp    0
   jr    z,ClearScreen_end
   sub      1     ; otherwise, keep going
   ld    b,a
   jp    ClearScreen_yloop
ClearScreen_end:
   pop      de
   pop      hl
   pop      bc
   ret

.END
```

```
;
; Garg Demo!
; by Bart Trzynadlowski
; December 20, 2005 and January 7, 2006
;

;
; demo.asm
;
; Let's you move a Garg around the screen. The sprites were ripped from
; Commander Keen: Invasion of the Vorticons by id Software.
;


;*****************************************************************************
; Constants
;*****************************************************************************

;
; Variables
;
; Addresses for variables stored in RAM.
;

VBL = $F001          ; VBL counter

PLAYER_X = $F002     ; X position of player
PLAYER_Y = $F003     ; Y position
PLAYER_VX = $F004    ; velocity X
PLAYER_VY = $F005    ; velocity Y
PLAYER_JUMP = $F006  ; incremented each frame jump button is held down for

PLAYER_X_PREV = $F007   ; X position during previous update cycle
PLAYER_Y_PREV = $F008   ; X position during previous update cycle

ANIM_PTR = $F00A     ; base of 2-byte pointer to animation sequence

;
; Constants
;

SPRITE_WIDTH = 23       ; width of sprite in pixels
SPRITE_HEIGHT = 31      ; height
GRAVITY = 1             ; acceleration due to gravity
TERMINAL_VELOCITY = 9   ; maximum vertical speed
WALK_SPEED = 1          ; walking speed (MUST be < SPRITE_WIDTH/2)
MAX_JUMP = 0            ; ??? maximum velocity due to jump


;*****************************************************************************
; Vector Table
;*****************************************************************************

;
; ResetVector:
;
; Execution begins here when the Z80 powers up.
;

.ORG $0
ResetVector:
   di             ; disable interrupts
   ld sp,$F400    ; set SP to the top of the RAM area
   jp Start

;
; InterruptVector:
;
; Do nothing.
;
```

```
.ORG $38
InterruptVector:
   reti

;
; NMIVector:
;
; Triggered at the start of the VBlank period. Increments VBL.
;
.ORG $66
NMIVector:
   push  af
   ld    a,(VBL)
   inc   a
   ld    (VBL),a
   pop   af
   reti


;*****************************************************************************
; Main Program
;*****************************************************************************

;
; Start:
;
; Program entry point.
;

Start:
   ;
   ; Clear screen and enable interrupts
   ;
   ld    hl,$9000    ; control reg
   ld    (hl),4      ; settings mode
   ld    a,0
   ld    ($8000),a   ; disable interrupts and turn screen off
   call  ClearScreen
   ld    (hl),4
   ld    a,3
   ld    ($8000),a   ; turn display and interrupts on

   ;
   ; Initialize player
   ;
   ld    a,0
   ld    (PLAYER_X),a
   ld    (PLAYER_Y),a
   ld    (PLAYER_VX),a
   ld    (PLAYER_VY),a
   ld    (PLAYER_JUMP),a
   ld    hl,anim_idle   ; get address of idle animation
   ld    (ANIM_PTR),hl  ; store as current animation pointer

   ;
   ; Main Loop:
   ;
   ; - Read inputs
   ; - Update player velocities
   ; - Apply acceleration
   ; - Save old coordinates
   ; - Update player coordinates
   ; - Perform out-of-bounds checks and clamp appropriately
   ; - Wait until VBL
   ; - Erase sprite
   ; - Draw sprite at updated coordinates
   ;
MainLoop:

   ;
   ; Read joypad
```

```
   ;
   ld    hl,$9000 ; control reg
   ld    de,$A000 ; joypad status
   ld    (hl),8   ; mux = 1
   ld    a,(de)   ; A = --UDLRBC
   ld    b,a      ; save copy in B

   ;
   ; Update player velocities
   ;
   Ld    hl,anim_idle      ; default animation
   ld    (ANIM_PTR),hl
   and   $08               ; if LEFT is pressed (0), move left
   ld    a,0
   jr    nz,no_left
   ld    hl,anim_walk_left ; queue animation...
   ld    (ANIM_PTR),hl
   ld    a,-WALK_SPEED
no_left:
   ld    (PLAYER_VX),a
   ld    a,b               ; get button data
   and   $04               ; if RIGHT is pressed, move right
   jr    nz,no_right
   ld    hl,anim_walk_right   ; queue animation...
   ld    (ANIM_PTR),hl
   ld    a,WALK_SPEED
   ld    (PLAYER_VX),a
no_right:

   ;
   ; TO-DO: Implement jumping...
   ;

   ;
   ; Apply acceleration (jumping and gravity)
   ;
   ld    a,(PLAYER_VY)
   add   a,GRAVITY
   cp    TERMINAL_VELOCITY    ; check for terminal velocity
   jr    nc,dont_clamp_vy
   ld    a,TERMINAL_VELOCITY  ; if reached, clamp the velocity to that
dont_clamp_vy:
   ld    (PLAYER_VY),a

   ;
   ; Save old coordinates
   ;
   ld    a,(PLAYER_X)
   ld    (PLAYER_X_PREV),a
   ld    a,(PLAYER_Y)
   ld    (PLAYER_Y_PREV),a

   ;
   ; Update player coordinates (ie., apply velocities!)
   ;
   ld    a,(PLAYER_X)
   ld    b,a
   ld    a,(PLAYER_VX)
   add   a,b
   ld    (PLAYER_X),a
   ld    a,(PLAYER_Y)
   ld    b,a
   ld    a,(PLAYER_VY)
   add   a,b
   ld    (PLAYER_Y),a

   ;
   ; Perform out-of-bounds checks.
   ;
   ;
   ; There is a trick here: We merely assume that the maximum X coordinate is
```

```
   ; 255 for the purpose of this demo. Therefore, if the player X exceeds
   ; (256 - SPRITE_WIDTH/2), we assume that the player has wandered too far
   ; left and clamp to 0. Then, we check for the coordinate exceeding
   ; (256 - SPRITE_WIDTH) and clamp there.
   ;
   ; This is all under the assumption that WALK_SPEED < SPRITE_WIDTH/2.
   ;

   ld    a,(PLAYER_X)
   cp    256-(SPRITE_WIDTH/2)
   jr    c,no_clamp_left   ; if carry, no need to clamp to X=0
   ld    a,0
   ld    (PLAYER_VX),a     ; if clamping X, also kill velocity
no_clamp_left:
   cp    (256-SPRITE_WIDTH)
   jr    c,no_clamp_right  ; if carry, we are to the left of the test pt.
   ld    a,0
   ld    (PLAYER_VX),a     ; kill velocity
   ld    a,256-SPRITE_WIDTH
no_clamp_right:
   ld    (PLAYER_X),a      ; write back the clamped X coordinate

   ld    a,(PLAYER_Y)      ; clamp Y coordinate now to bottom of screen
   cp    240-SPRITE_HEIGHT
   jr    c,no_clamp_bottom
   ld    a,0
   ld    (PLAYER_VY),a
   ld    a,240-SPRITE_HEIGHT
no_clamp_bottom:
   ld    (PLAYER_Y),a

   ;
   ; Wait until VBL
   ;
   call  WaitVBL
   call  WaitVBL
   call  WaitVBL

   ;
   ; Erase sprite at old location and draw new one
   ;
   ld    b,23
   ld    c,31
   ld    a,(PLAYER_X_PREV)
   ld    d,a
   ld    a,(PLAYER_Y_PREV)
   ld    e,a
   ld    hl,spr_blank
   call  DrawSprite

   ;
   ; Draw sprite at new location using current animation frame
   ;
   ld    a,(VBL)  ; use refresh rate / 8, and then multiply by 2 because
   srl   a        ; each pointer is 2 bytes. So only shift twice.
   srl   a
   and   $06      ; only 4 animations...
   ld    e,a
   ld    d,0
   ld    hl,(ANIM_PTR)  ; get base of animation
   add   hl,de    ; add in the offset
   ld    a,(hl)   ; get low byte of sprite address
   ld    e,a
   inc   hl
   ld    a,(hl)
   ld    d,a
   push  de
   pop   hl       ; transfer address of sprite to HL

   ld    a,(PLAYER_X)
   ld    d,a
```

```
   ld    a,(PLAYER_Y)
   ld    e,a
   call  DrawSprite

   jp    MainLoop


;****************************************************************************
; Video Functions
;****************************************************************************

;
; WaitVBL:
;
; Waits until the next VBlank period has started.
;
WaitVBL:
   push  af
   push  bc
   ld    a,(VBL)
   ld    b,a   ; old VBL counter into B
WaitVBL_loop:
   ld    a,(VBL)
   cp    b
   jr    z,WaitVBL_loop
   pop   bc
   pop   af
   ret

;
; DrawSprite:
;
; Draws a sprite on the screen.
;
; Input:
;  B  = X size.
;  C  = Y size.
;  D  = X position.
;  E  = Y position.
;  HL = Sprite address.
;
; Saves AF, BC, DE, HL.
;
DrawSprite:
   push  af
   push  bc
   push  de
   push  hl

   push  hl
   push  bc
   ld    bc,$8000 ; data register
   ld    hl,$9000 ; command register
   ld    (hl),2   ; set X position high to 0
   ld    a,0
   ld    (bc),a
   ld    (hl),1   ; set X position low
   ld    a,d
   ld    (bc),a
   pop   bc
   pop   hl

yloop:
   ld    a,3      ; set Y position
   ld    ($9000),a
   ld    a,e
   ld    ($8000),a

   ld    a,0      ; data write mode
   ld    ($9000),a
   push  bc       ; save X size
```

```
xloop:
   ld    a,(hl)      ; fetch pixel
   inc   hl
   ld    ($8000),a   ; write to framebuffer
   dec   b           ; decrement X count and loop
   ld    a,b
   cp    0
   jr    nz,xloop

   pop   bc
   inc   e           ; next line in framebuffer
   dec   c           ; decrement Y count and loop
   ld    a,c
   cp    0
   jr    nz,yloop

   pop   hl
   pop   de
   pop   bc
   pop   af
   ret

;
; ClearScreen:
;
; Clears the screen to black. Saves BC, HL, DE and destroys everything else.
;
ClearScreen:
   push  bc
   push  hl
   push  de

   ld    a,1
   ld    ($9000),a   ; set address X low
   ld    a,0
   ld    ($8000),a
   ld    a,2         ; set address X high
   ld    ($9000),a
   ld    a,0
   ld    ($8000),a

   ld    b,239       ; 240 lines, start clearing at last one
   ld    de,1
ClearScreen_yloop:
   ld    a,3
   ld    ($9000),a   ; set address Y
   ld    a,b
   ld    ($8000),a
   ld    a,0         ; data mode
   ld    ($9000),a

   ld    hl,320      ; 320 pixels
ClearScreen_xloop:
   ld    ($8000),a   ; write black pixel
   scf               ; clear carry
   ccf
   sbc   hl,de       ; X loop
   jr    nz,ClearScreen_xloop

   ld    a,b         ; if we cleared line 0, quit
   cp    0
   jr    z,ClearScreen_end
   sub   1           ; otherwise, keep going
   ld    b,a
   jp    ClearScreen_yloop
ClearScreen_end:
   pop   de
   pop   hl
   pop   bc
   ret
```

```
;****************************************************************************
; Sprite Data
;****************************************************************************

;
; Animations are hard-coded to be 4 frames. The tables consist of pointers to
; sprites and are cycled in sequence.
;

anim_idle:
   .dw   spr_garg + (0*23*31)
   .dw   spr_garg + (1*23*31)
   .dw   spr_garg + (2*23*31)
   .dw   spr_garg + (3*23*31)

anim_walk_left:
   .dw   spr_garg + (6*23*31)
   .dw   spr_garg + (7*23*31)
   .dw   spr_garg + (6*23*31)
   .dw   spr_garg + (7*23*31)


anim_walk_right:
   .dw   spr_garg + (4*23*31)
   .dw   spr_garg + (5*23*31)
   .dw   spr_garg + (4*23*31)
   .dw   spr_garg + (5*23*31)

;
; Blank sprite to erase with
;

spr_blank:
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   .db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;
; The actual Garg sprites are appended as raw binary data to the ROM file
;
spr_garg:

.END
```

Appendix E: Pictures

Here are some photographs of my implementation of the BartStation. Figure E.1 shows the whole system; the 6-button Sega joypad, the Spartan-3 board, and the Z80 circuit breadboard.

Figure E.1 Complete system.

Figure E.2 shows the Z80 board up close.

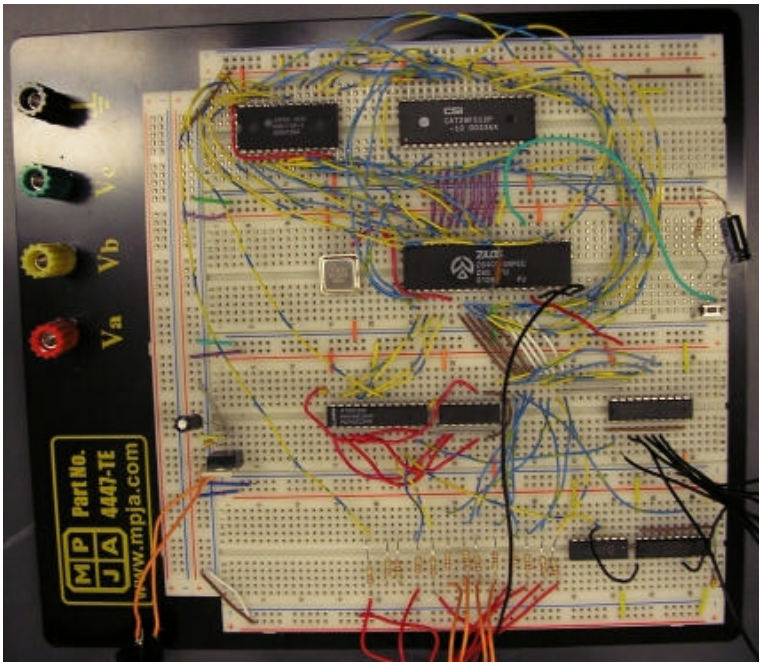


Figure E.2 The Z80 board.

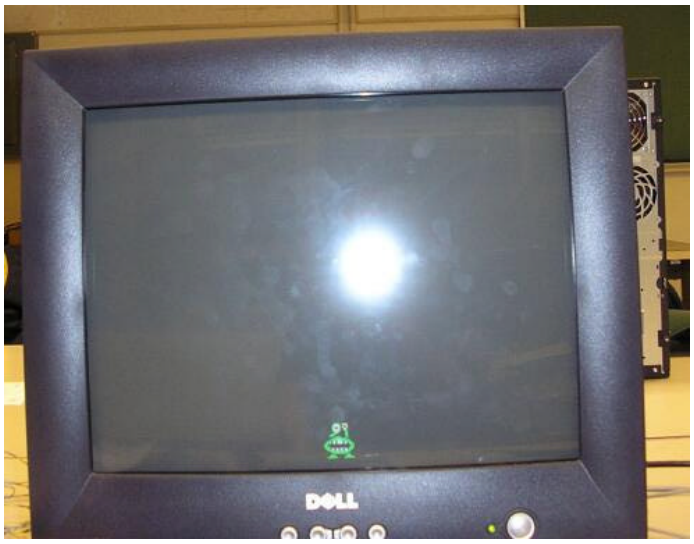


Figure E.3 The “Garg Demo” running.

And, finally, figure E.3 shows the “Garg Demo” running.