Defender Recreated, the classic arcade game in a FPGA

Jesús Arias

1 Introduction

In the early eighties Williams Electronics launched it's best known arcade game: Defender. When compared to other alien-shooters of its time, like Space Invaders or Galaxian, Defender was simply mind blowing. It was colorful, with fast controls, a breathtaking pace, and a sound out of this World. Yet, its hardware was relatively simple and later Williams games incorporated many other hardware improvements like bit-blitters, but in my opinion these later games where unremarkable, so the only arcade machine from Williams I wanted to recreate in a FPGA was the Defender.

And for that purpose we already got a board designed around an ICE40HX4K FPGA and a VGA video output. Named SIMRETRO by the student who designed it, it has enough resources to recreate all the arcade hardware in a board with only three chips (not counting voltage regulators). Namely:

1. A 7680 logic-cell, 128Kbit internal RAM (16KBytes), 144-pin, FPGA.
2. An Static RAM with 128KBytes of memory and a 16-bit data bus.
3. An SPI Flash with 8MBytes of memory. Used for FPGA configuration and also for storage.

The Defender electronics includes 4 boards with lots of ICs. It is really two separate computers, with the main board including a 6809 CPU, 48KB of RAM, most of it used for video, and the video logic built using TTLs. The ROMs are installed in a separate board totaling 26kB of memory. This board also includes a PIA (MC6821) responsible for communications with the sound board, for the reading of machine control inputs, like configuration buttons and coin switches, and for the generation of interrupts. The game buttons are read from another PIA in a separate board (Interface board). And Finally there is the sound board, built around a 6802 CPU with 128 bytes of RAM, 2KB of ROM, a PIA, and a digital to analog converter. This later board its basically a microcontroller that runs in parallel with the main CPU, generating the sounds being ordered on its PIA input.

This project really started with the search of suitable CPU cores on Internet. These cores had to be free, written in Verilog, and cycle accurate. After some searching I found the mc6809 core of Greg Miller and the ac68 6800 core of Hideyuki Abe. This last core was a bit difficult to work with due to the almost non existing documentation. Fortunately the included testbench was enough to figure out its synchronous memory design (the data read from current address is available on next clock cycle). With the 6800 core working it was easy to recreate the sound board by following its schematic. The main difference with the original board is the use of Pulse Width Modulation instead of Digital to Analog conversion. The Defender recreated project started as a lot of strange sounds in a speaker.

Also, the two CPU cores where synthesized in order to have an estimate of the number of logic cells required, that resulted in about 3400 cells for the 6809 and about 2100 cells for the 6800 of the sound board. The CPUs account for most of the logic, so it was clear the FPGA had enough room for the two cores and still we had many logic cells free for the rest of the peripherals.

It was thus the time for the video board design, starting with the VGA interface. The main document followed here was the "Defender early pcb theory" where the board schematics were dissected. Also, it was

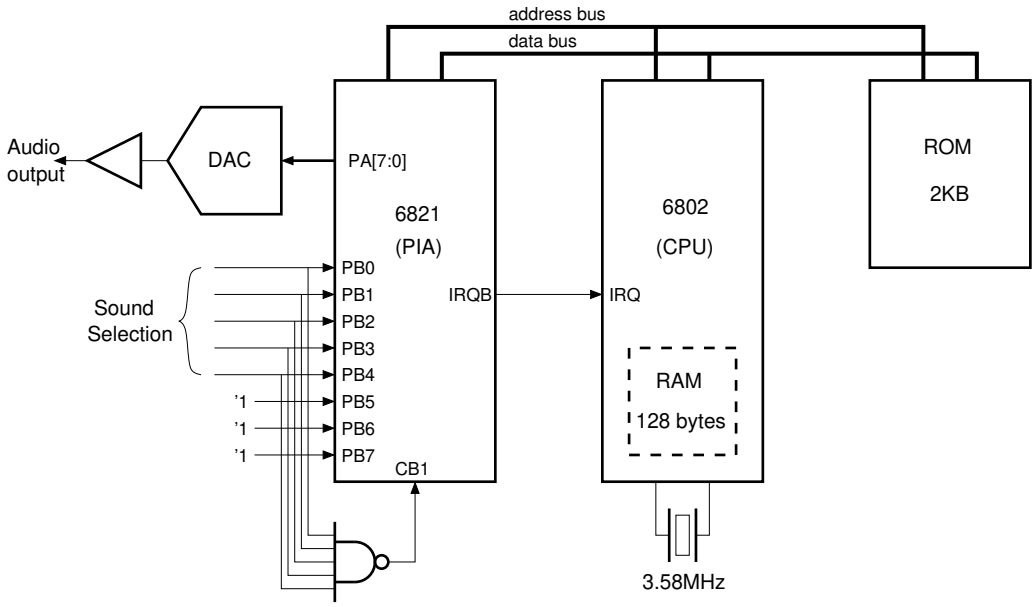


Figure 1: Simplified diagram of the sound board

Element		Address	Actual decoding	Comments
RAM		$00 to $7F	A[15:8]=8'b00000000	Internal RAM
PIA	Port A (dir A)	$400	A[15:12]=4'b0000 && A10=1	Output to DAC
	Control A	$401		Not used after reset
	Port B (dir B)	$402		Input from video game
	Control B	$403		Not used after reset
ROM		$F800 to $FFFF	A[15:12]=4'b1111	Code

Table 1: Memory map of the sound board

very clarifying a memory dump provided by Sean Riddle on his web page (https://seanriddle.com/willy.html) that showed the way pixels are really stored in RAM. The video board was soon running, but the game still had issues with the palette RAM. It took a time to fix them because the problem was related to interrupts instead.

Even with a wrong palette the video board and the sound board were interconnected and the game was played on a VGA screen showing it was as challenging as in the eighties. Finally, with the palette issues solved, the player's ship was able to explode properly with its pieces fading to red as they fly apart after just a few seconds of playing. A genuine Defender frustration valued as 8.33 pesetas (today 0.05€, not accounting for inflation) that now comes for free with Defender Recreated ;)

2 The sound board

The Defender sound board was actually designed for Pinball machines and reused for the game. Its block diagram is shown in Figure 1. It includes a 6802 CPU with 128 bytes of internal RAM and a 3.58MHz crystal. This clock is divided by 4, so, the actual CPU frequency is 895kHz. Alternatively, the board can host a 6808 CPU and a separate RAM chip, resulting in a totally equivalent circuit. The code of the sound board is stored in a 2KB ROM chip, and all the I/O is handled by a 6821 Parallel Interface Adapter (PIA). In the PIA the port A is programmed as output and connected to a digital to analog converter that drives the cabinet speaker after an amplification stage. Port B is used as input, with only 5 bits actually used in the Defender game. The PIA can also request interrupts to the CPU. This happens when the sound selection changes from all ones (value 31) to any other value. In this event a rising edge reach the CB1 input on the PIA, resulting in the IRQ line being asserted. A read of port B will clear the interrupt request.

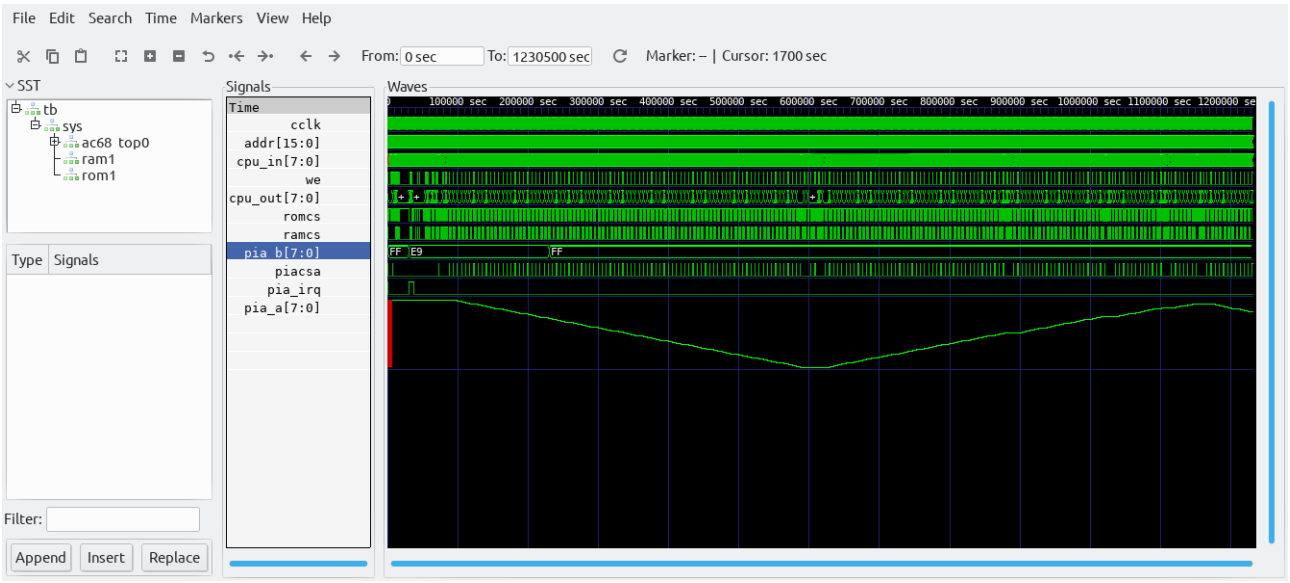


Figure 2: Testbench of the sound board playing sound #22 (engine thrust)

The only other thing we need to know to design a Verilog system equivalent to this board is the way the memories and the PIA are selected, or in other words the memory map of the board, that is listed in table 1.

The PIA is an unfriendly device for an FPGA recreation due to its bidirectional ports, but looking at the schematic we can see that port A is always output and port B always input, and therefore, a complete PIA implementation isn't required. In fact, in the Verilog code the PIA is replaced by a simple output register for port A, a multiplexer for input data for port B, and a flip-flop for interrupts. The direction and control registers for ports A and B aren't implemented at all, and the same happens with the interrupt flag of port B. Because CB1 is the only interrupt source in this system the interrupt flag (bit 7 of Control B) is not read in the code. Interrupts are thus requested when the output of the NAND gate shown in figure 1 goes high and cleared on port B reads.

This extremely simple PIA recreation is enough to get a working sound board. Other thing we still have to address is the clock generation. Our master clock is a 25MHz signal due to VGA requirements. By dividing this clock by 28 we get a 892.86kHz signal that is only a 0.24% below its nominal value of 895kHz. The difference in frequencies is so small that it is not noticeable by human ear.

Well, the only problem remaining is the lack of an digital to analog converter in the FPGA board. But we can use a fast PWM instead. The pulse-width modulator is an 8-bit counter whose value is compared to a buffered copy of the port A output of the PIA. When the counter overflows the PWM output is set to 1, and when a matching to the buffer happens the PWM output is cleared to 0. If a set and a clear happens in the came cycle (when port A value is zero) the set is ignored. Port A is copied to the buffer register on counter overflows. With a clock frequency of 25MHz and a total count of 256 the resulting PWM frequency is 97.65kHz, high enough to drive a power CMOS inverter built with discrete MOSFETs that is directly connected to an 8Ω speaker. The resulting audio is clearly a Defender soundtrack.

The design of the sound board was an easy one. It also served as a training exercise for the main Defender computer. Here the PIA was replaced with simple I/O ports and we are also planning to do the same in the big board. A simulation is shown in figure 2, where a background noise waveform is displayed at the port A of the PIA as an analog signal. As we can see, the wave is a sequence of ramps with random slopes. In the game this sound is generated when the player's ship is moving and there are no other sounds around. The FPGA real state used by the sound board recreation is listed next. It requires 2201 logic cells, most of them for the 6800 CPU, and 5 memory blocks (each memory block is a 512 byte memory, either RAM or ROM)

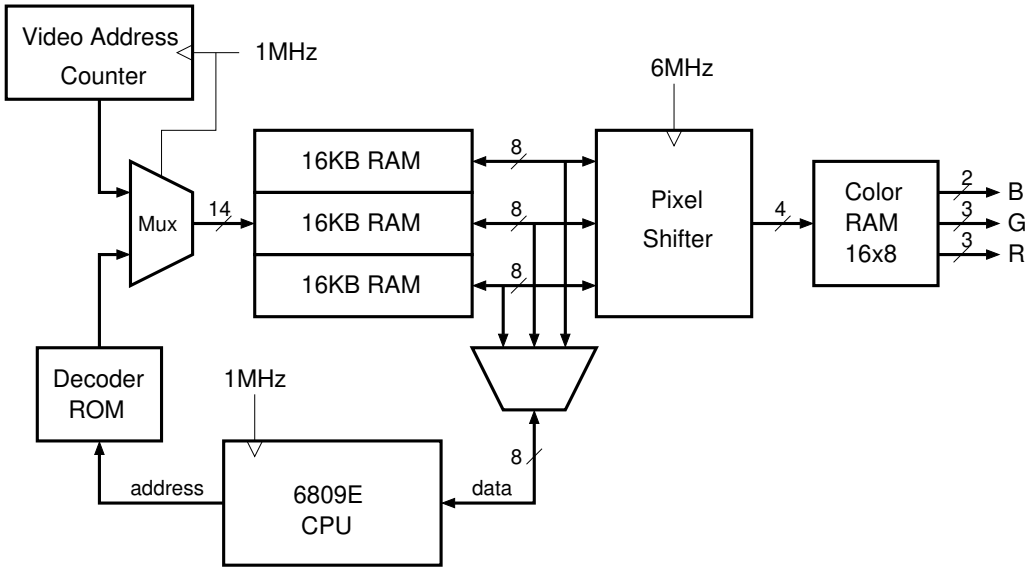


Figure 3: Simplified diagram of the video generation logic

```
Info: Device utilisation:
Info:            ICESTORM_LC:  2201/ 7680    28%
Info:           ICESTORM_RAM:     5/   32    15%
```

3 The main Defender computer

The Defender hardware comprises 4 boards. One of them is the sound board, while the other three are all for the main computer of the game. They are:

1. The Video board, that includes the 6809E CPU, the RAM memory, and all the logic related to video generation.
2. The ROM board, that includes all the ROMs, the paging logic, and one PIA. The PIA provides a communication channel with the sound board, generates interrupts, and reads the cabinet management inputs, like setup buttons and coin switches. It also drives 4 diagnostic LEDs.
3. The Interface board with one additional PIA for the reading of the 9 game buttons: Up, Down, Thrust, Reverse, Fire, Bomb, Hyperspace, One player, and Two players.

The schematics of these boards are well detailed in the "Defender Theory of Operation" manual. In overall this is a quite complex computer and not all its features are going to be recreated in our FPGA version. In particular, the video generation logic is going to have a completely different design, mostly due to the different video output (VGA instead of NTSC) but also because we have a much faster RAM than the original board and this feature can be exploited to simplify the design. But lets first talk a little about the original video board.

3.1 Video generation in the original Defender board

The video generation logic follows the diagram of figure 3. Here, three DRAM banks are used for the frame-buffer, totaling 48KB of RAM. The memory is accessed twice each microsecond: one read for video refresh and one read or write for CPU operations. Memory accesses are therefore interleaved and the CPU isn't slowed at all due to video refresh. But the video logic has to output 6 pixels each microsecond, and we have 4 bits per pixel. In order to get this required bandwidth the three memory banks are read in parallel during video refresh and the resulting 24 bits are stored in shift registers. These shifters runs with a 6MHz clock, and generate a 4 bit

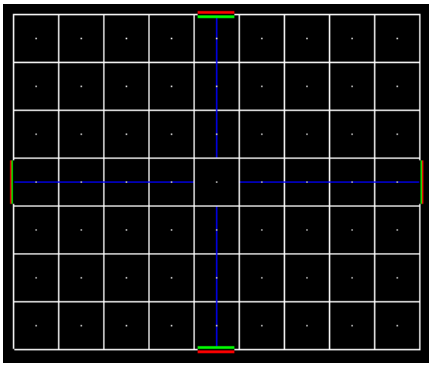


Figure 4: Test pattern for the monitor (302x252 pixels)

output for each screen pixel. But this isn't the final video value, this 4 bit data is used as an address for a color RAM that outputs an 8 bit data with the final BGR233 video values (these RAMs were later called "palettes"). These BGR values are translated into video levels using three, maybe too weird, resistor-transistor DACs and applied to the CRT inputs.

From the video address counter are derived not only the memory addresses for the video refresh, but also the horizontal and vertical sync pulses for the monitor and an horizontal blanking signal that turns the video output to black during retraces. Interestingly, no vertical blank signal is generated.

An NTSC line is 64μs wide, and this corresponds to an horizontal total of 384 pixels with 300 pixels visible. The vertical total is 260 lines, resulting in a 60Hz refresh rate. The NTSC standard has 525 lines in two interlaced fields of 262 and 263 lines, so the Defender video is a little out of specs. Also, all of the 260 lines are potentially visible, with the lines 252 to 255 being repeated after line 255. These lines had to be black on the video memory because no vertical blank signal exist. In fact, looking at memory dumps, it seems that no image is present after line 240, with the exception of a test pattern for the monitor (figure 4). Also, an interrupt is generated after reaching line 240, suggesting this is the start of the vertical blanking interval (lines 240 to 259). The vertical sync pulse also starts at line 248. From all this data its difficult to state a given resolution for the Defender video. 300$\times$240 is the minimum, but from memory dumps we can see up to 312 horizontal pixels in the image, and the test pattern of figure 4 has 252 vertical lines. It seems that Defender designers were pushing the resolution of monitors a bit too much, and probably the image stored in the memory was bigger that the actual image displayed in the screen.

The palette RAM is a write-only memory from the CPU point of view, but writes to this memory interferes with video refresh and at least three pixels are corrupted for every write. For this reason the game only changes the palette contents while in the vertical retrace. But the lack of a true vertical blank also means that some color dashes could be seen on the screen if the monitor isn't properly adjusted (an issue pointed out by Sean Riddle in his web page: https://seanriddle.com/lines.html)

And from the CPU point of view the video RAM is arranged in a completely different way, with the board using a ROM to translate the 8 MSB bits of the CPU address into a 6-bit pseudo address along with the corresponding bank select signal. This is so weird that schematics were almost useless to figure out the correspondence between video addresses and CPU addresses. At the end I resorted to try to get an screen capture out of a memory dump of the game (also from Sean's web), and I found a quite simple, yet unusual, pixel arrangement in the memory that is shown in figure 5, starting at the upper-left corner of the screen. Talking only about CPU addresses, an increment of one is translated as a whole line of video, and an increment of 256 results in a displacement 2 pixels to the right (every byte of memory holds two pixels). Using this pixel order I was able to get the image of figure 5 using an approximate palette because the actual palette memory wasn't included in the dump (the palette is a write only memory). The image is 312 pixels wide and this is interesting because the horizontal blank is asserted on pixel #300. But some of the extra 12 pixels could actually

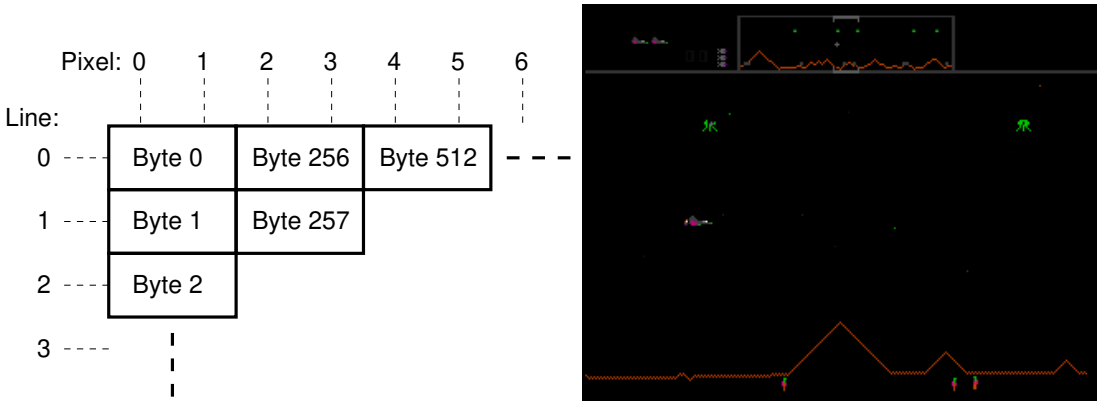


Figure 5: Pixel arrangement in memory and the corresponding image extracted from a memory dump (312×240 pixels, shown using an arbitrary palette)

be displayed due to the turn-off delay of the blanking transistor (Q10, color RAM circuit schematic). Again, another uncertainty regarding video resolution.

In summary, the video generation logic is weird and its recreation is going to take a much simpler approach while maintaining the pixel arrangement of figure 5 for a 312×240 resolution and a VGA timing.

3.2 The ROM paging

The video image fills the first 39KB of RAM, with the remaining 9KB being used for system variables. There are also 26KB of ROM, totaling 74KB of memory. This exceeds the 64KB address range of the CPU and, consequently, memory banking is used. The idea is to have a 4-bit page register whose value is used in the selection of the banked ROM ICs when a read in the memory range $C000 to $CFFF is performed. Up to 16 different 4KB chips could be selected, adding 60KB to the address space, but in the Defender board there is provision for only 4 ROM pages (pages #1, #2, #3, and #7, with only 2KB ROM selected at page #7). Page #0 is special because it selects peripherals instead of ROM. The peripherals are:

- The color RAM (palette). A 16-byte, write only memory, at address $C000 to $C00F
- A watchdog reset register. A write-only register at address $C010 (actually written at $C3FF). A value $38 written to this register will reset the watchdog for another 133ms.
- The NVRAM. A 256×4 battery backed RAM at address $C400 to $C4FF. Only the 4 LSBs of every byte are actually implemented.
- The Vertical Count Buffer. A read-only register with the 6 MSBs of the line number being displayed, at address $C800.
- The ROM board PIA, at address $CC00 to $CC03.
- The Interface board PIA, at address $CC04 to $CC07.

The page register is a write-only one, and it is mapped to the $D000 to $DFFF range. Notice that the $D000 to $FFFF range will select the unbanked ROMs (12KB), but there is no problem in mapping the page register inside this range because write-only devices can use the same address space as read-only ones.

3.3 Interrupts.

All interrupts are of the IRQ type (the NMI and FIRQ inputs of the CPU are always inactive) and all are generated by the ROM board PIA. These are:

- The CA1 interrupt. This pin is driven by a rectangular wave that goes high when the line being displayed is in the range 240 to 259 (vertical retrace).
- The CB1 interrupt. This pin is driven by a square wave with a 4.096ms period generated by the video address counter.
- The CA2 interrupt. This pin is driven high when the coin door opens.

4 The FPGA redesign of the main Defender computer

My aim here wasn't the building of an exact Defender replica in a FPGA, but just a compatible computer capable of playing the game. The design targeted a particular FPGA board (SIMRETRO) that was specifically designed for these kind of computers and includes a VGA interface with 4-bit DACs in its RGB signals. VGA runs at double the speed than NTSC (a line takes 32μs instead of 64μs) and is not interlaced, with a total of 525 lines. I also wanted to get a good image on LCD monitors, without sampling artifacts, and to achieve this it is a good idea to have the same pixel rate than the standard VGA mode (640x480). VGA uses a 25 MHz pixel clock (well, 25.175MHz, but 25MHz is also fine and gives a much easier math), and dividing this frequency by two we get 320 horizontal pixels, a little more than the 312 pixels desired. So, the main idea here is to use a 12.5MHz clock for pixel shifting and to duplicate lines to obtain 480 visible lines instead of 240.

The SIMRETRO board also includes 128KB of RAM outside the FPGA. This RAM has a 16 bit data bus, but also byte-select signals, so it is easy to use it as an 8-bit memory. Also the FPGA has 16KB of memory that can have a known initial content, and therefore it can be used as ROM. But this is not enough ROM for all the Defender game while we still have a lot of RAM unused. The idea here is to store the game ROMs in the upper RAM, and to inhibit writes to this memory area after the initial uploading of data. We still need a boot ROM for the upload, but it can be an small one and it can be placed inside the FPGA. We will also need some I/O ports for accessing the SPI flash and to read the Defender ROM images from there.

There is also the problem of the user interface. The Defender arcade has lots of inputs: 9 buttons for the player and 6 more inputs for coin slots and setup. My idea was to use a NES game controller clone for playing, but it has only 8 buttons, so, the button mapping will require some thinking. Also, I would like to translate the Thrust and Reverse buttons of the Defender game into the Left and Right arrows of the game controller.

And finally, I want to remark that some features of the game never were planned to be incorporated into the replica, like the watchdog circuit or the coin door interrupt, and others are only half functional, like the NVRAM.

And now, lets go into the details.

4.1 Clocking

We already found that an adequate pixel clock is 12.5MHz. The external memory is more than capable to perform a read or write during one of such clock cycles. Moreover, a byte contains two pixels and, consequently, video reads should be performed every two pixel clock cycles, for instance on even pixels. Therefore half of the clock cycles are available for CPU accesses, but this would result in a CPU frequency of 6.25MHz, a lot faster than the original Defender rate (1MHz).

We want a CPU frequency of 1MHz, but this isn't possible using integer dividers. So, what can we do? My solution was to generate a non periodic clock waveform in the following way:

- A 5-bit counter is incremented using the 25MHz clock. When the count reaches 24 the counter is reset to zero and a CPU clock is requested by asserting the rE signal (E is the main CPU clock)

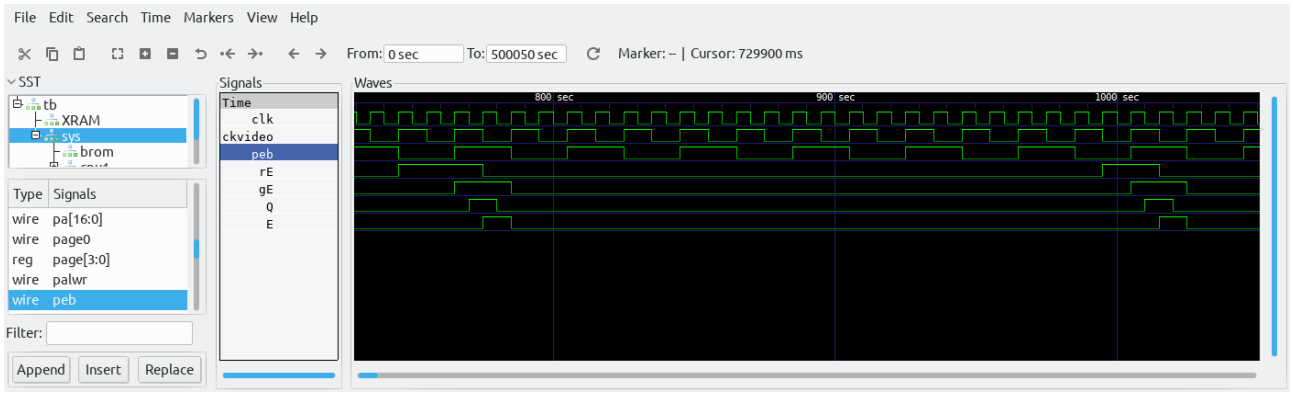


Figure 6: Clock generation simulation. "clk" is a 25MHz signal. "peb" is the LSB of the horizontal pixel counter.

- If rE is active, and the time for an odd pixel comes, the CPU clock pulse is granted by activating gE. When this happens rE is also reset. gE remains active for a whole pixel clock cycle.
- When gE is active the CPU clocks E, and Q, toggle, providing a clock pulse for the CPU.

In this way the time between E pulses can be more or less, but on average we get a pulse every microsecond. A simulation is shown in figure 6 where we can see the variable delay between rE and gE assertions.

The gE signal is also used to select the address that is presented to the external memory: If gE is zero the address comes from the video generator, and if gE is one the address comes from the CPU and paging logic. Addresses are 17 bit wide.

4.1.1 Slow clocks

Apart from the video and CPU clocks we still need some other clock signals of a much lower frequency. For instance, there is an interrupt input that is driven by a square wave with a 4.096ms period (244Hz). In the original Defender this signal was a video address bit. I tried to do the same and the game failed to update the palette RAM. It took me some time, silly debugging the palette logic, until I realized the signal I used for the interrupt were not a square one because in my case the video has an odd number of lines (525). At the end I resorted to use an independent counter driven by the horizontal sync pulse of the VGA. This pulse happens every 32μs and the counter multiplies this period by a power of two, obtaining the correct 4.096ms period at the interrupt input. This solved the strange palette issues.

Also another clock is required for the NES controller. This controller is basically a CD4021, parallel to serial shift register, and the reading of its inputs involves a load pulse followed by seven clock pulses. I chose a 1.024ms period for this clock, that is translated to 8.192ms between button samples. This low sampling rate avoids any problem related to switch bouncing. The NES clock comes from the same counter as the 4.096ms interrupt.

4.2 Memory map and booting.

The Defender recreated has two operaing modes: Boot and Play. The current mode depends on a bit in an I/O port and after reset this bit select Boot mode, enabling some aditional logic that isn't present during Play mode. Namelly:

- A 512 byte ROM can be read from adresses $FE00 to $FFFF. This memory contains the code for booting.
- An output register can be written at address $BFxx (see figure 8). This register contains the mode bit (BOOT) along with the SPI output signals.

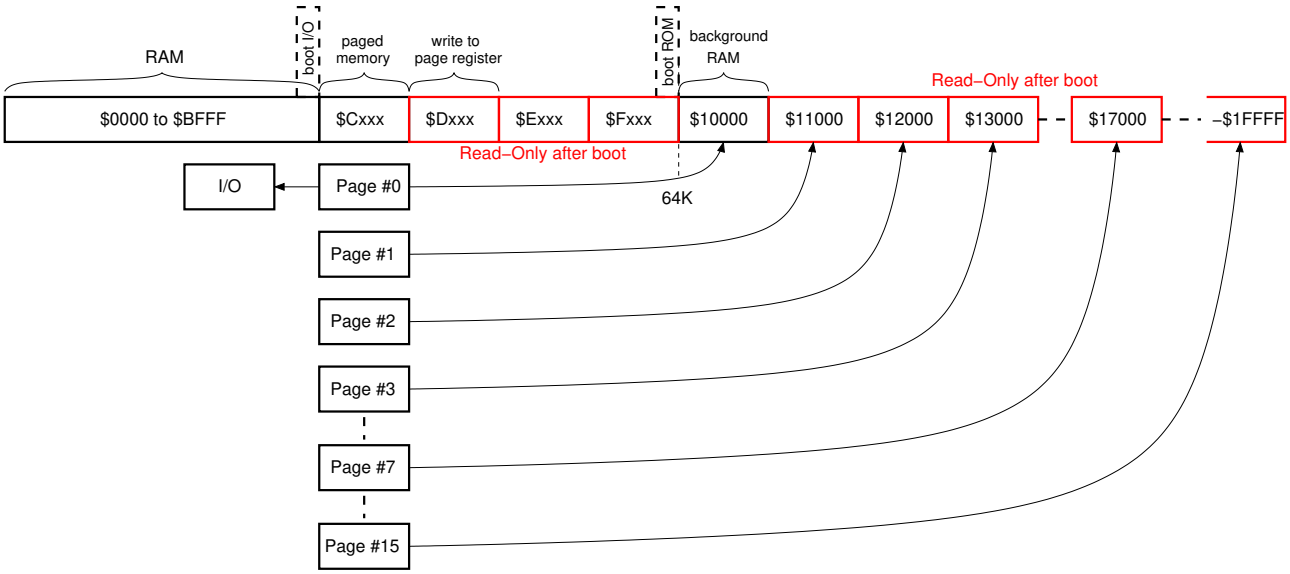


Figure 7: Memory map of the Defender recreated.

	7	6	5	4	3	2	1	0
Write to $BFxx	MOSI	SCK	/SS	/CSSD	BOOT	---	---	---
Read from $BFxx	MOSI	SCK	/SS	/CSSD	BOOT	---	---	MISO

Figure 8: I/O register for boot. After reset all output bits are '1', including BOOT.

- An input register can be read at address $BFxx (see figure 8). This register contains the same bits as the output register (for read back) and also the input signal from the SPI bus.
- All the external RAM is writable, including the address range of the boot ROM.

In Play mode the boot ROM and the extra I/O register are no longer accesible for programs, and also the writing to ROM addresses is inhibited.

After this explanation about boot mode we can present the memory map of figure 7. Here the 128KB of the external RAM are shown along with its correspondence with ROM pages. The RAM and the unpaged ROMs are mapped to the lower 64KB of the memory, while the paged ROMs are stored in the upper half of the external RAM. Up to 15 pages exist, even if Defender uses only 4. Also, page #0, that is reserved for peripherals, is mapped to the upper RAM, providing a "background" memory for the areas of the page where there are no peripherals selected. In this way the NVRAM gets implemented for free ;) Also, this background memory allows the read back of palette values.

The upper RAM is accesible through the mapped 4KB pages at address $Cxxx. The 17-bit address for the external RAM is obtained using the following Verilog code:

```
assign selC000=ca[15]&ca[14]&(~ca[13])&(~ca[12]);
assign cam = selC000 ? {1'b1,page,ca[11:0]} : {1'b0,ca};
```

Where "ca" is the CPU address (16-bit) and "cam" is the 17-bit RAM address. (Notice that the address range $0C000 to $0CFFF of the external memory is never read nor written)

4.2.1 Boot procedure

After reset the 6809 PC gets loaded with the two last bytes of the boot ROM and the execution jumps to the boot code. The two main functions of this code are the filling of the memory with the ROM images stored in

the SPI flash, and the switching to play mode by clearing the BOOT bit and jumping to the code of the loaded ROMs. Going into further detail we can list the following actions:

- Setting the DP register to $A0 for local variable storage at $A0xx, and the stack pointer to $BEFF, just below the boot I/O register.
- Selecting the SPI flash by setting the /SS bit low in boot I/O register.
- Sending a read command to the SPI flash with the address of the ROM images in that flash ($7E0000. Low addresses are used by FPGA configuration images and $7F0000 also contains a ROM image for a ZX spectum ;) All SPI data is transferred to the flash via bitbanging, and the read command is the sending of the bytes: $03, $7E, $00, and $00.
- Write $1 to $D000. This selects the ROM page #1. Then read 4096 bytes from the SPI bus and store them in the range $C000 to $CFFF.
- Repeat the reading for pages #2 and #3. Store the 4KB data in the same $C000 to $CFFF range.
- Next select page #7 and read 2048 bytes to the range $C000 to $C7FF.
- Now read 12288 bytes and store then in the range $D000 to $FFFF. Bytes written to the range $FE00 to $FFFF can't be read back because reads will select the boot ROM instead of the external RAM, but writes are always directed to the external RAM. Also, the data writen to the $D000 to $DFFF range has changed the value of the page register. We want to return this register to a value of zero, but we have to be carefull because that zero will also be written to RAM. Fortunatelly the byte at address $D130 is already zero, so we can use this particular address to write the page register.
- Now we have all the ROM images availabe in RAM. We can deselect the SPI flash and prepare for starting the game. This implies two things: writting the BOOT bit with zero and an indirect jump to the contents of the reset vector. But this code can't be executed in the boot ROM because this ROM is replaced with RAM as soon as the BOOT bit is written with zero. We have to copy the relevant instructions to RAM and to jump there. Any RAM location is good but I choose the address $8000 that is visible on the screen.
- After the trampoline code is copied to RAM we can jump to it. Its instructions are:

```
lda    #$37  ; Disable BOOT (bit 3). It can't be no longer enabled
sta    $BF00
ldx    $FFFE ; Load reset vector to X
jmp    ,x    ; and do an indirect jump
```

With the last jump instruction the control is transferred to the Defender game. It all goes well the screen gets filled with random data and then the following message is displayed:

INITIAL TEST INDICATE

UNIT OK

Then the game enters a setup menu because the contents on the NVRAM were wrong. By reseting the FPGA again without removing the power the NVRAM now holds valid data (default values) and the game proceeds to the Williams Defender Star-Wars-like presentation screen.

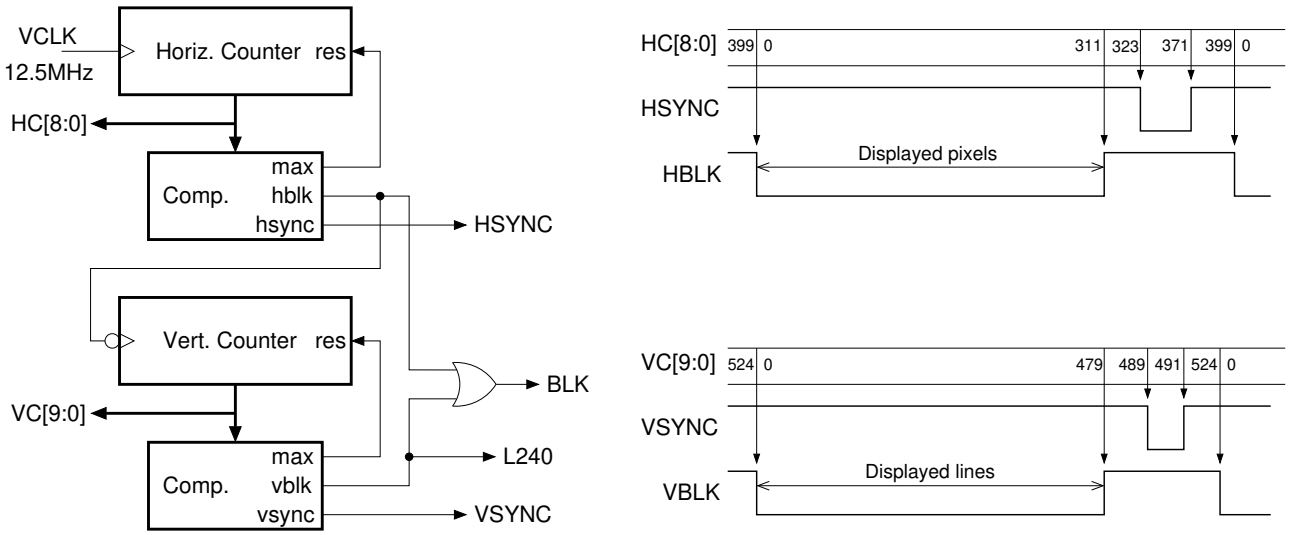


Figure 9: Video timing generator and its waveforms.

4.3 VGA Video

Lets present now the design of the video logic. Some of its characteristics were already discussed, like the way the memory is shared with the CPU, so lets talk about its internal details next.

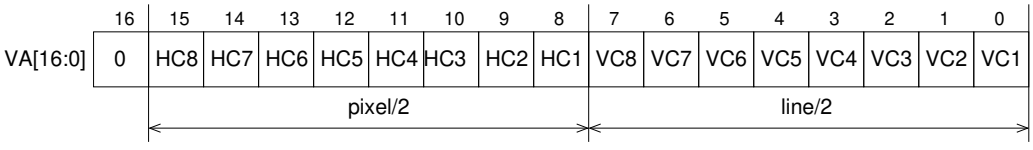
4.3.1 Video timing

The video timing circuit, whose block diagram is shown in figure 9, is built around two binary counters and some comparators. The 9 bit horizontal counter runs at 12.5MHz and gets incremented with every pixel. A video line takes 32μs and this means a total of 400 pixels/line. Therefore the counter counts up to 399 and then it get reset. But the visible part of the line is only 312 pixels, so, an horizontal blanking signal is generated by being set when the counter reaches the value 311 and reset when the counter reaches 399. The horizontal sync pulse is generated in a similar way. It is being reset on count 323 and set on count 371 (hsync is active low). The sync pulse has to be placed inside the blank portion of the line, and by moving it early or late the image on the screen is shifted right or left (Well, at least on CRT screens. On modern LCD monitors with auto-adjust buttons this makes little sense).

The vertical counter follows the same design ideas, but it is clocked by the horizontal blank signal instead of VCLK. Also it is a 10 bit counter because its maximum value is 524. Contrary to the video logic of the Defender, here we also have a vertical blank that is active from lines 480 to 524. The image has double the lines than the game (480 instead of 240) and, therefore, each line of the game has to be displayed twice. The vertical blank signal is also labeled as Line240 and it is used as an interrupt input (CA1 in the ROM PIA).

4.3.2 Video addresses

The video address is a 17 bit value obtained by the concatenation of several bits of the horizontal and vertical counters. By looking at the pixel order in figure 5 we can conclude that the horizontal counter value has to be placed starting at bit #8, but every memory byte stores two pixels, so the count has to be divided by two. The same happens with the vertical count because lines have to be displayed twice. The resulting address is then:



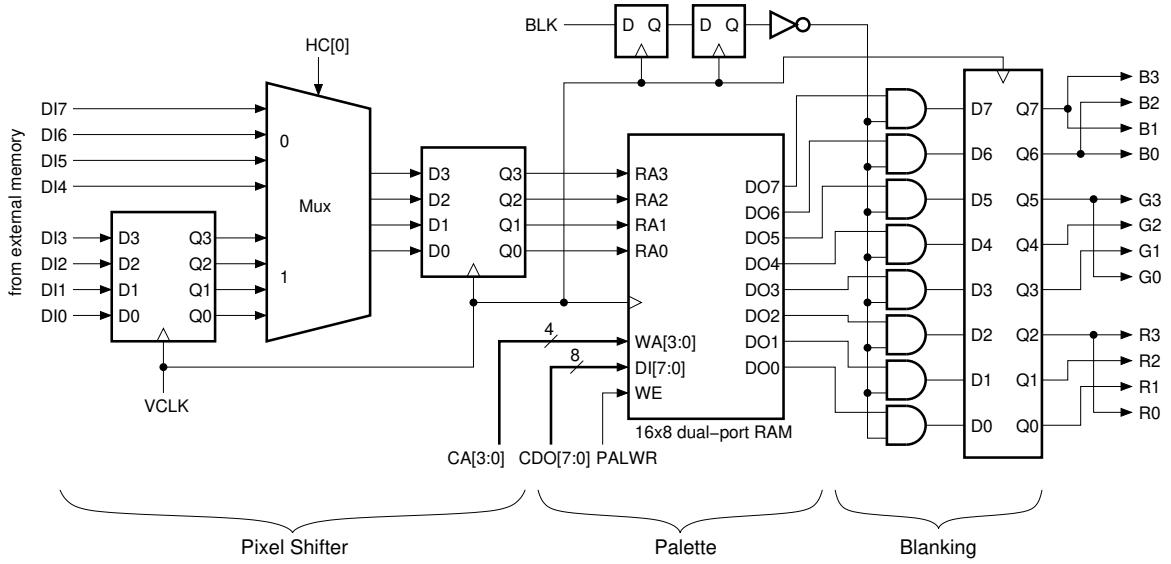


Figure 10: Block diagram of the video pipeline.

Video memory is read all the time, even during blanking time, and the vertical counter can roll over after line 511 resulting in incorrect addresses. But this isn't a problem at all because the video is going to be black during blanking time.

4.3.3 Video pipeline

The video section of the design ends with the pixel pipeline shown in figure 10, which has three different sub-blocks. The first one is the pixel shifter, that is a quite simple circuit. The data from the memory contains video information on even pixels (on odd pixels we can get CPU data instead). Thus, on even pixels the 4 MSBs of the data are copied to the output register and the 4 LSB are stored on another temporary 4-bit register. Then, in the next cycle the 4-bit data of the temporary register is copied to the output register.

Next follows the palette memory. A memory block of the FPGA is used here and its dual port feature is put to full use. In contrast with the original Defender hardware, here we can write to the palette RAM at any time without random dashes being displayed on the screen.

The palette converts a 4-bit pixel data into an 8-bit BRG signal. The only remaining processing is the blanking of the output during retraces. This is done by anding the outputs with an inverted an delayed blank signal. The two cycle delay is required because both the pixel shifter and the palette are synchronous devices with one cycle delay each. The final 8-bit register was included to avoid combinatorial glitches at the output (these glitches went unnoticed in an old monitor but were visible on a modern one). The video output has 3 bits for the red and green components and only 2 bits for blue. This signal has to be converted to 4 bit per component before driving the R-2R DACs of the SIMRETRO board. But this requires no logic, we only have to connect the MSB bits of the color components to the additional LSB ones, as is shown in figure 10.

4.4 Inputs and the NES controller

In adition to all the memories and video logic we must also recreate the Defender peripherals. These are only the vertical count buffer and two PIAs. The former is a 6-bit input port (at address $C8xx) where the MSBs of the vertical counter of the video timing can be read. In the recreation this input is saturated. This means that if the vertical counter is over line 511 the value presented at the input is 8'b1111_11xx.

Then comes the PIAs.

ROM board PIA ($CC00 to $CC03)		
PIN	DIR	FUNCTION
PA0	In	Auto UP switch (*)
PA1	In	Advance button (*)
PA2	In	Coin Right
PA3	In	High Score Reset switch (*)
PA4	In	Coin Left
PA5	In	Coin Center
PA7,PA6	Out	Diagnostic LEDs (ignored)
PB5-PB0	Out	Sound selection
PB7,PB6	Out	Diagnostic LEDs (ignored)
CA1	In	Line 240 interrupt
CB1	In	4ms interrupt
CA2	In	Coin door interrupt (ignored)
CB2	In	Sound board ACK (unused)

Interface board PIA ($CC04 to $CC07)		
PIN	DIR	FUNCTION
PA0	In	Fire
PA1	In	Thrust
PA2	In	Bomb
PA3	In	Hyperspace
PA4	In	2 Players
PA5	In	1 Player
PA6	In	Reverse
PA7	In	Down
PB0	In	Up
PB7-PB1	In	(unused)
CAx, CBx	nc	(unused)

(*) Management switches placed inside the Defender cabinet.

Table 2: Pin assignment of the Defender PIAs. All inputs are active high. Sound selection is active low (bits inverted).

Bit	Field	Function	Comments
0	CA1 control	CA1 IRQ Enable if 1	
1		CA1 EDGE (0=falling, 1=rising)	
2		Port/Dir (0=direction reg, 1=port reg)	Port and Dir share the same address
3	CA2 control	CA2 IRQ Enable if 1	If bit #5 is 0, different meaning if 1
4		CA2 EDGE (1=rising)	
5		CA2 dir (0=input, 1=output)	
6	IRQ Flags	CA2 IRQ Flag	Cleared on port read
7		CA1 IRQ Flag	

Similar control register for port B (replace CA for CB in the previous table)

Table 3: Bits of the control registers of the PIA

4.4.1 The PIAs

The 6821 PIA was already found in the sound board and in that case it was replaced with simple input and output registers. Lets try to do the same here, but before we have to known what functions the PIA pins have. In that case the "Theory of operation" manual wasn't enough. Thankfully, the comments in the source code of MAME came to the rescue. The Functions of the PIA pins are listed on table 2.

The Interface board PIA is used only as input and it can be replaced by two simple input ports at addresses $CC04 and $CC06, but the ROM board PIA is more complicated because of interrupts. There are several interrupt sources and the CPU has to read the port control registers in search for interrupt flags in order to determine the cause of interrupts. Moreover, it is possible to have some of the interrupts disabled, so, in the case of the ROM board PIA we have to do a minimally decent control register recreation, including the relevant interrupt enable and interrupt flag bits. The bits of the control register for port A (address $CC01) are listed on table 3.

The port pins for the PIA have fixed directions, and therefore the direction registers for port A and B can be ignored. The CA2 and CB2 registers are also inputs, so we don't have to worry about strobes (not detailed in table 3). The CA2 interrupt (coin door) can be left always inactive. And all interrupts are active on rising edges. This leaves us with only bits 0 and 7 of the two control registers requiring storage. Bit 7 is the interrupt flag and it is set when a rising edge is detected on CA1/CB1. This bit is cleared on port A/B read. If the flag is

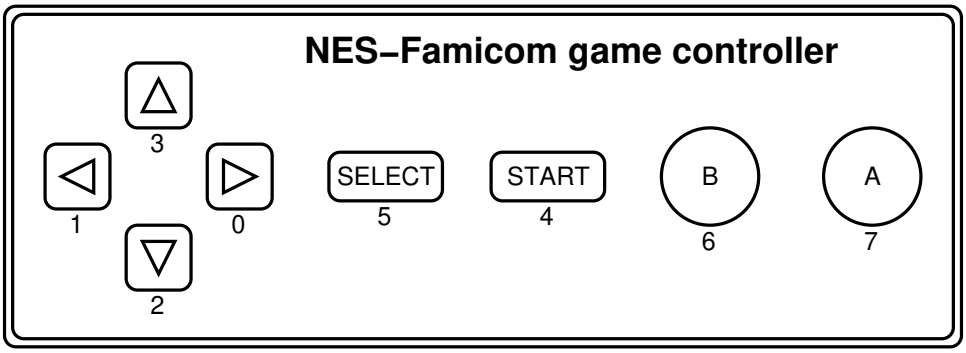


Figure 11: Button layout of the NES game controller and related bit numbers

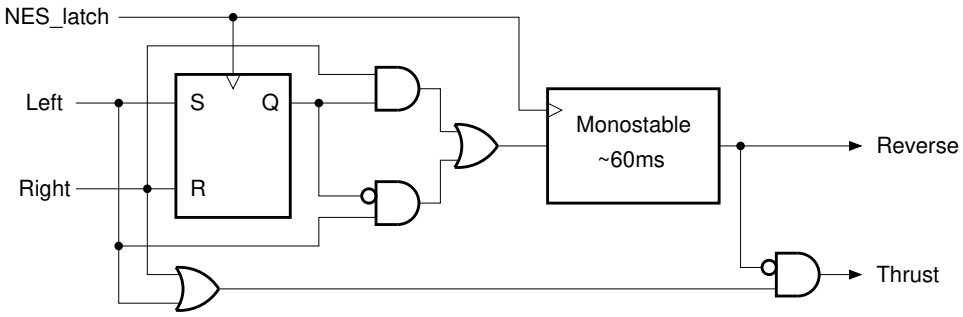


Figure 12: Logic for the Reverse input.

set, and the interrupt enable bit (bit 0) is also set, the IRQ input of the CPU is asserted.

4.4.2 The NES controller

A NES/Famicon controller was intended to be used for user input. It includes 8 buttons arranged as shown in figure 11. These buttons are read every 8.192ms by means of an 8-bit shift register, its levels inverted, so at the end they are active high, and stored. But Defender totals 15 inputs, so, something has to be done in order to play the game. The idea here was to remove as much inputs as possible, and to use combinations of buttons for the simulation of additional inputs. As an example, “Coin Center” and “Coin Left” are left inactive, and “Coin Right” is the AND of buttons “Fire (A)” and “Bomb” (B). Also, “Two Players” is inactive and “One Player” is the AND of “Hyperspace” (start) and “Fire” (A).

A very annoying thing was the “Reverse” button of defender. I would prefer to move left or right with the arrow buttons instead of redefining them as reverse and thrust. In order to use the left and right arrows as they are supposed to be, the additional circuit of figure 12 was included. Still, it isn’t a perfect solution because you have to press the right arrow just before the playing starts or you can end with the controls reversed. Yet, even with this limitation the controls are much more player friendly than the originals. The flip flop stores the last direction of the ship (0 = right), and the monostable is a 3-bit counter that gets loaded with 3’b111 when triggered and stops downcounting when it reaches zero. Both the monostable and the direction flip-flop use the latch signal of the NES controller as a clock (8.192ms period).

The final mapping of inputs are listed on table 4. With these controls the setup and test menus can be navigated and the game played.

4.5 Synthesis and FPGA real state

The complete design, also including the sound board, was synthesized using the ICESTORM public domain tools. The Lattice tools won’t be able to synthesize this design because for these tools the ICE40HX4K is limited to only 3520 logic cells while the chip actually have 7680 logic cells (The memory blocks are also

Defender input	Interface PIA pin	Game Controller	Bits
Reverse	PA6	Left and Right, see fig. 12	0 & 1
Thrust	PA1		
Up	PB0	Up	3
Down	PA7	Down	2
Fire	PA0	A & (~start)	7
Bomb	PA2	B	6
Hyperspace	PA3	Start & (~A)	4
One player	PA5	A & Start	7, 4
Two players	PA4	none	-

Defender input	ROM PIA pin	Game Controller	Bits
Advance	PA1	Select	5
Auto UP	PA0	Start	4
High Score Reset	PA3	Up	3
Coin Right	PA2	A & B	7, 6
Coin Left	PA4	none	-
Coin Center	PA5	none	-

Table 4: Correspondence between Defender inputs and game controller buttons

artificially limited to 20 instead of 32). The total usage of the FPGA resources is:

```
ICESTORM_LC:  5677/ 7680    73%

ICESTORM_RAM:    7/   32    21%
```

Again, most of the logic is used by the two CPUs of the system. The main Defender computer only uses two additional memory blocks: one for the boot ROM and another for the video palette.

5 Known problems and possible improvements

- When playing the game if the down button is pressed long enough the player's ship moves almost completely out of the screen. I don't know if this actually happens in the real game, but I'm starting to suspect that the ship is being drawn after line 240. This nasty habit of drawing things on the screen borders is starting to get on my nerves. So desperate were the Williams engineers to use up to the last line of the monitor, or they were just following the orders of some stupid management guy? The game is perfectly playable with 240 lines, anyway. No humanoid climb down below line 240. I tried to increase the number of lines of the video by means of removing the vertical blank signal but my LCD monitor refused stubbornly to draw more than 240 lines. Maybe this could have worked on a CRT screen, but on LCDs is useless. So, I have to stay with 240 lines, but not all hope is lost. The entire screen can be shifted up by means of adding the offset (in lines) to the video address, and surely, some lines can be removed from the top.

- And talking about improvements, there are still room for some of them. They can begin by adding more button combinations for the now unused inputs, like "Two players".

- Also, it would be interesting to have the coin door interrupt implemented and CA2 mapped to some combination of buttons just to see what happens if that interrupt is triggered. Does the game sound an alarm or something like that?

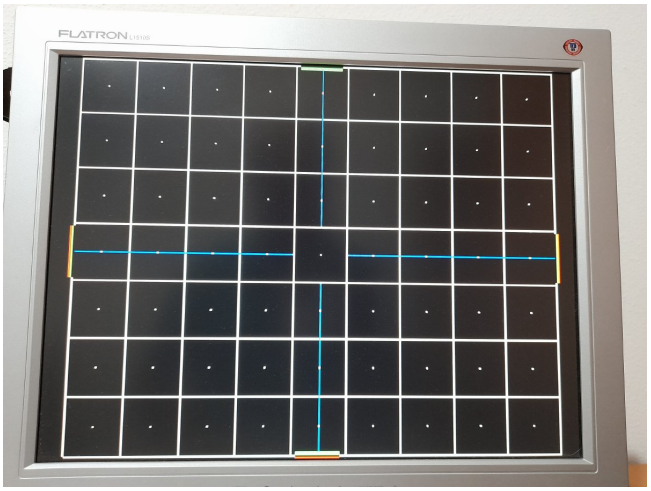


Figure 13: Test pattern for monitor after video address mod.

- Another possible improvement is the speed up of the booting code. The reading of the SPI flash is done via bitbanging and this is slow, with the reading of ROM images taking about 10 seconds. This can be much faster if the CPU clock is increased to 6.25MHz during the boot mode. I'm not sure about doing this modification. The ROM images are now beautifully displayed on the screen during the loading and the time it takes isn't so long...

- The last "to do" improvement I have in mind is the nonvolatile storage of the NVRAM contents. This has to be done in the boot code and it will involve the erasing and writing of some SPI flash sector. The boot ROM size is only 512 bytes. This size was selected because it is the size of a single memory block in the FPGA, but we still have many blocks unused, so the boot ROM can be made bigger and it could store more code than now.

5.1 Update

- The problem of the ship hidden at the bottom was finally solved by means of adding a small offset to the video address generated. Again, the memory dump was useful to find that the first 7 lines of the image are actually black. So I added 7 to the video address and now the whole game field is visible. In retrospect it makes sense to have these lines black because the vertical sync pulse ends at line #0 and some time is required for the retrace. Now, every line starts at address $xx07 and ends at address $xxF6 (instead of $xx00 to $xxEF). This little mod will cost an extra 8-bit adder in the FPGA bill (there is no carry after bit #7), but now the image is finally perfect. The test pattern for the monitor is now completely displayed (see figure 13), or at least it looks so at first glance. There is still a missing line at the top, but an offset of #6 results in a missing line at the bottom, so I'll keep the offset at #7.

- Another successful modification was the speed up of the boot code. The trick was simply to replace the request E pulse signal (rE) by the logic OR of rE and BOOT. In this way, when in boot mode, every odd pixel is a cycle for the CPU and it runs at 6.25MHz instead of 1MHz. The loading time is now less than 1/6 than before.

- The coin door interrupt was also added but it does nothing. After some debugging it is clear the interrupt flag is set but the CPU keeps the interrupt enable bit off, so, the game ignores that interrupt completely. The CA2 flag can still be read by pooling, and it probably is because it gets reset, but no action is taken by the game when the coin door switch is toggled. Not only that, the CA1 interrupt (line>=240) is also disabled. Only the CB1 interrupt (4.096ms interrupt) is on. Therefore, the CA2 interrupt, with its flag

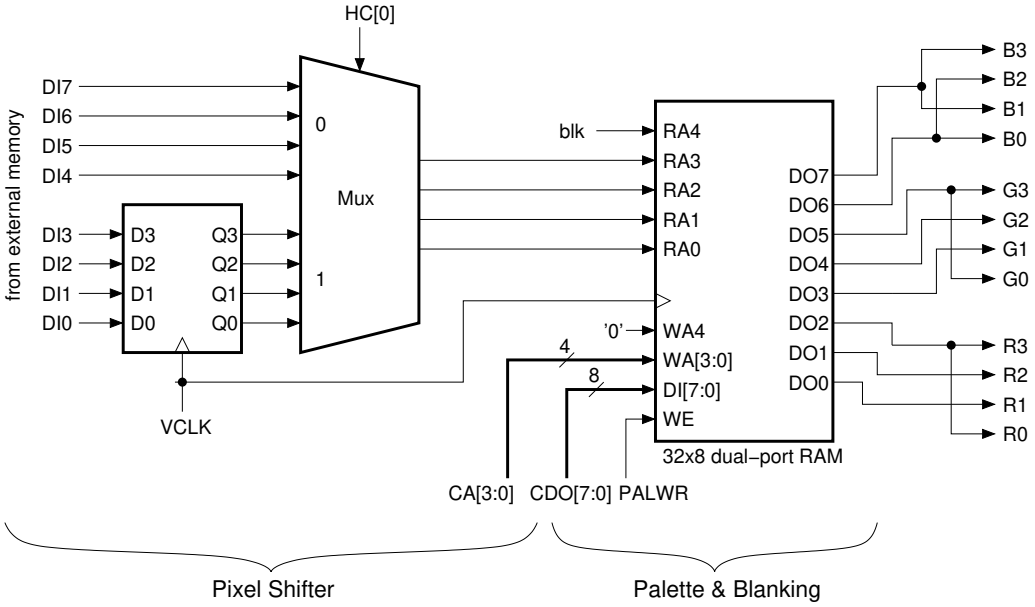


Figure 14: Simplified video pipeline with combined palette and blanking

and enable bits, is simply useless and could be removed from the design to save a few logic cells. The CA1 interrupt logic can also be simplified by removing its interrupt enable bit.

- The CA2 interrupt was generated by pressing Start and B in the game controller. In the same way, by pressing Select, Start, and B, a reset pulse is generated (and it works ;) I found it convenient to have a three-finger-salute at hand instead of removing power or using the FPGA downloading tools to perform the reset. Still, more inputs can be generated using other combinations of buttons.

- The video pipeline was also simplified. The main idea here was to move the blanking function into the palette memory (see figure 14). Now, the palette RAM has 5 bit addresses and 32 entries. This is double in size than before, but it will result in the same BRAM occupation because the smaller memory block is 512 bytes. Only entries #0 to #15 can be written by program, the high 16 bytes always remains as zero. Therefore, during blanking times the reading address is always higher than 15 and the video output is forced to zero, as expected. Also, the current pixel register and the blank delay have been removed because they weren't really needed thanks to the BRAM being a synchronous memory. The resulting circuit block is now quite simpler than before.

6 Conclusions

The Defender recreation in a FPGA was an ambitious project with very satisfactory results. The game plays flawlessly, I bet better than on any emulator, and this is mainly due to the fact that I'm building a similar hardware in the FPGA instead of dealing with an emulation code fighting against an operating system with all its software headaches (window-manager, video scaling, sound API with its buffers and latency, multitreading... and some other tons of crap).

And talking about hardware, It took me some time to dig into the details of the original Defender schematics with all their subtleties. The lack of a vertical blank signal, and also the use of a slow transistor switch for the horizontal blanking, resulted in an uncertainty in the video resolution that was difficult to figure out precisely. In my opinion the design of the video blanking and digital to analog conversion in the original Defender board is a crappy one. After expending hundreds of ICs in the boards these guys resorted to discrete transistors to save a pair of NAND gate ICs that could have made the blanking a reliable one and the D/A conversion more

predictable (now the color even depends on the temperature of transistors because of their Emitter-Base voltage drop). Not only that, there are 8 diodes in series with the palette outputs that aren't needed at all because these outputs are of the open-collector type.

In summary, I think this approach is a better one than the usual Raspberry-Pi inside the cabinet. First, it is more faithful to the original game hardware, it provides an inherent parallelism that makes possible a perfect playing using slow clocks (Defender recreated has two CPUs inside the FPGA running in parallel), it has a very fast boot (much faster than any OS booting), it consumes less power, and finally, it still has the capability of being reprogrammed for the recreation of other kind of arcade games or personal computers (A ZX spectrum was also recreated in the same FPGA board)