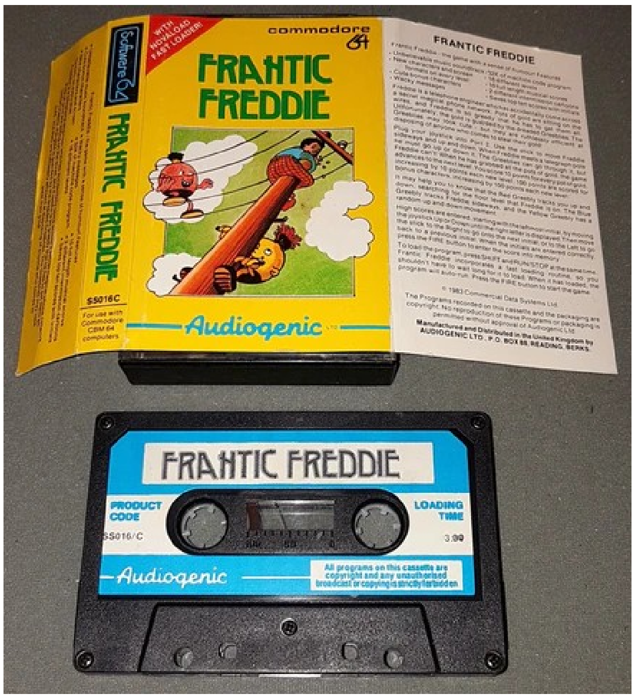
Using Greenpaks for retro gaming

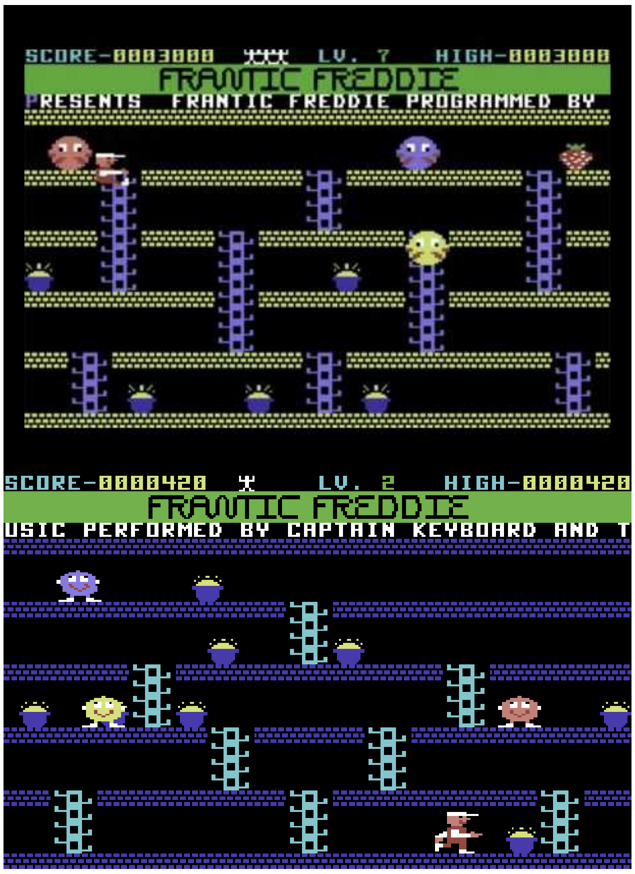
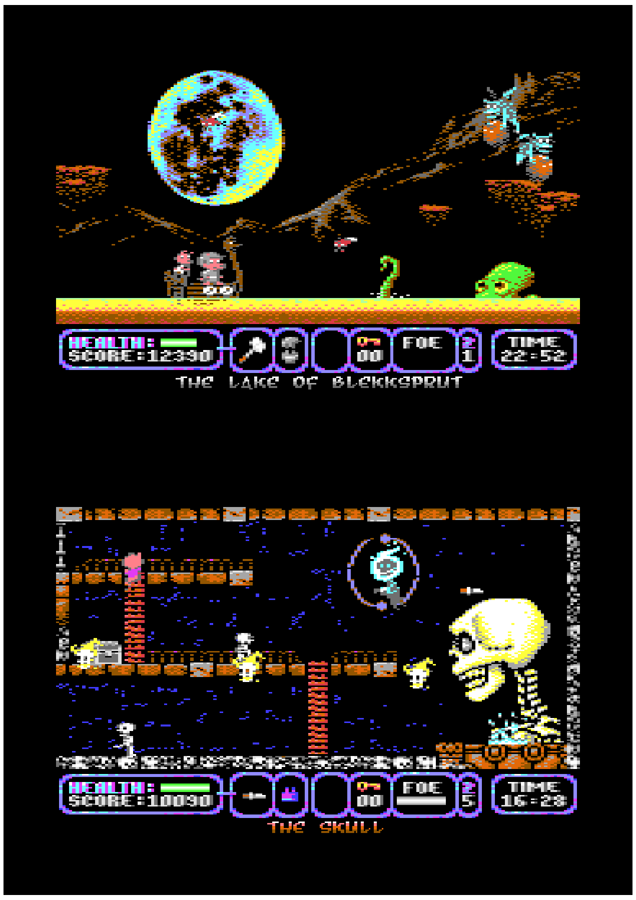
Introduction

Retro Gaming is very popular nowadays. Not only do people play old games for nostalgic reasons, there also a lot of software and hardware development and innovation being done on old computers.

One of the old computers that is still receives a lot of love in the 21th century is the Commodore 64. I am myself involved in the Commodore 64 community. The amount of people using a Commodore 64 today is still large enough to publish games on a commercial basis for the platform. Although the computer has a long and glorious history in games, over and over again programmers prove that the computer end of its capabilities has not been reached. In fact, during recent years the computer has received some of the most impressive and most polished games ever.

Some things have changed during the last 40 years…

Frantic Freddie (1983)	**A pig quest (2023)**
 commodore 64 FRANTIC FREDDIE Audiogenic SS016C PRODUCT CODE LOADING TIME	
Size: 31KB **Soundtrack:** 8 minutes **Graphics:** Text mode with custom charset, sprites barely have animation **Storage medium:** Cassette	**Size:** 1 MB **Soundtrack:** 94 minutes **Graphics:** Bitmap mode with background animations + fully animated sprites 1 bitmap = 9KB. Game contains 200 screens = 9x200=1.8MB of bitmap data (compressed into cartridge) This is just the screen backgrounds. **Storage medium:** EasyFlash cartridge

Graphics reused between levels:	No graphics reuse, not between levels, not even between different screens:
 SCORE-0003000 LV. 7 HIGH-0003000 FRANTIC FREDDIE PRESENTS FRANTIC FREDDIE PROGRAMMED BY SCORE-0000420 LV. 2 HIGH-0000420 FRANTIC FREDDIE USIC PERFORMED BY CAPTAIN KEYBOARD AND T	 HEALTH: SCORE:12330 00 FOE 1 TIME 22:52 THE LAKE OF BLEKKSPRUT HEALTH: SCORE:10090 00 FOE 5 TIME 16:28 THE SKULL
	Hardware: EasyFlash cartridge

During the last years several Commodore 64 titles have appeared that fill up an 1MB cartridge up to the very last byte that is available:

- Briley Witch Chronicles (2021) (homebrew)
- Lykia the Lost Island (2022) (commercial)
- Eye of the Beholder (2022) (homebrew)
- A pig quest (2023) (commercial)

More large games are in development and discussion with those developers learns that they are constrained by cartridge size in what they can do.

Conclusion:

We need to design even bigger cartridges!

A bigger flash ROM

Idea: Bigger parallel flash ROM

The most basic idea to get a bigger cartridge is to simply increase the size of the ROM. Most C64 cartridges currently use 512KB parallel ROMs and it is not diffucult to see why: They are relatively low-cost, can run on 5V and the PLCC32 package is friendly for both hand-soldering as well as automated SMD assembly. What if we want a bigger parallel flash ROM?

Example: SST39VF6401

- 8MB parallel flash ROM
- cannot run on 5V and thus needs voltage conversion on a huge number of pins
- TSOP48 or even ball-grid-array not fun to solder by hand, okay for automated assembly
- Price: €8.18 excl. VAT

The price of the chip alone already shows that the big parallel flash ROM isn't going to help producing a game at a friendly price. We still need to add voltage regulators and voltage level converters to the cartridge. We also need extra bank switch registers to deal with the larger ROM size.

Idea: Big serial flash ROM

Example: W25Q64

- 8MB QPI flash ROM
- cannot run on 5V but we only need voltage conversion on a few pins
- Price: €0.41 excl. VAT

There is no escape from the economic reality: The price shows, that if we want to go for a bigger cartridge, serial flash ROMs are the way to go. If we only pay €0.41 for the flash, we have budget left for additional components and there is still opportunity to be able to produce the cartridge at a friendly price.

The problem is that the bus of the Commodore 64 natively supports parallel flash ROMs while for serial flash ROMs interface circuitry will be needed. It is possible to design some very basic circuitry with 74xx logic chips that allows software to do bitbanging. This will work, but the flash memory cannot be directly accessed from the computer, bitbanging routines will have to retrieve the data. This makes flash access slow, but also removes the ability to execute code directly from cartridge memory, a technique regularily used by C64 programmers to save RAM.

There have been attempts before to construct cartridges that convert serial to parallel with the help of a small FPGA: The Gmod3 cartridge. The maximum 104MHz clock speed of the W25Q64 certainly allows for a lot of flash clock cycles within a Commodore 64 clock cycle, so the idea sounds feasible. However, the chip shortage has caused FPGA prices to skyrocket and it looks like the Gmod3 cartridge will not be economically feasible anytime soon.

My idea: QPI to OPI conversion

The Commodore 64 has an 8-bit data bus. A QPI flash ROM has a 4-bit bus to communicate with the outside world. What if we don't do a full parallel to serial conversion (where we use both the address and data busses), but just convert 4-bit to 8-bit and then connect the serial ROM just to the 8-bit data bus?

I'm calling it "octal pheripheral interface", or OPI.

Suppose we map the flash rom to a register in the IO1 memory region of the Commodore 64, for example $DE00. The CS line of the flash is automatically activated on access to $DE00 and will stay active. The register is also mirrored at memory address $DE01 but any access to $DE01 will make the flash CS line deactivate at the end of the clock cycle.

Suppose we want to read 256 bytes from flash memory at flash memory address $012345. In that case the Commodore 64 can do this:

- Write "fast read quad I/O" command $EB to $DE00
- Write the high address byte $01 to $DE00
- Write the mid address byte $23 to $DE00
- Write the low address byte $45 to $DE00
- Write the M byte $00 to $DE00
- Read 255 times a byte from $DE00
- Read a byte from $DE01

Note that we can just repeatedly read bytes from $DE00, without any need to increase a pointer, without worrying about page boundaries, without worrying about cartridge bank boundaries. These things would have to be worried about in a traditional bank switched parallel flash ROM.

This means that a serial flash ROM, using QPI to OPI conversion, can actually be accessed faster than a parallel flash ROM! This should help developers push the boundaries of the Commodore 64 even further.

Sequential access memory

There is still a problem: The Commodore 64 won't do a cartridge boot with just a register in memory. The Commodore 64 detects from a cartridge signature in the ROML memory region, if found it will boot from cartridge.

To solve this, we will also map the flash ROM register to the ROML memory region at $8000..$9FFF. The address bus is ignored, so it doesn't matter at which address you read, the flash ROM will always return the next byte from flash memory.

In other words, the flash ROM will become sequential access memory: Each access returns the next byte.

Mapping the register this way allows us to make the C64 detect the cartridge signature and it will do the cartridge boot. We can also execute code in the ROML region, the C64 CPU will increase the program counter and read the next instructiom from flash. Just jumps will be problematic because the address is ignored.

This technique doesn't just allow us to boot, but is also useful for speedcode, a technique regularily used by C64 programmers to get more performance out of the CPU.

A challenge here is that the cartridge logic will have to send the initial command and address to the flash ROM on boot (for example $EB $00 $00 $00 $00 to start execution from the beginning of flash memory).

A little bit of SRAM

A believe that a cartridge with nothing more than sequential access memory will be too revolutionary and unconventional for people. I want to include a traditional parallel component, so programmers still can rely on what they have done for decades.

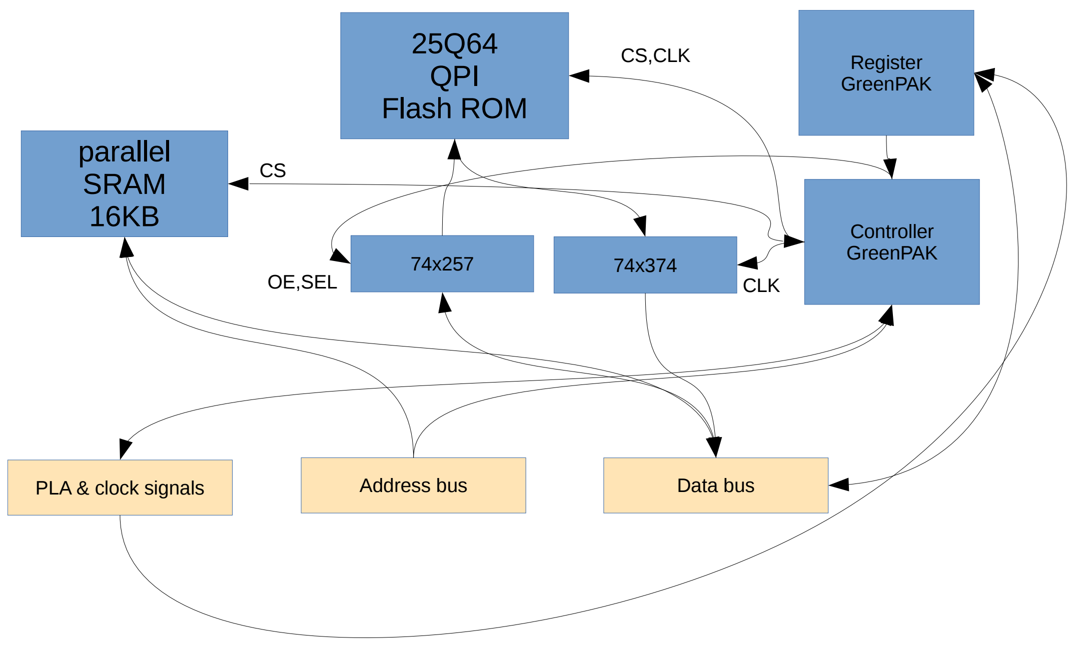
One possibility is to add a small parallel ROM to the cartridge. This also would solve the boot issue in a very traditional way.

However, it would be much more powerful to add a small SRAM to the cartridge. After all, when you have 8MB of flash memory, 16KB of additional ROM won't make big difference in capacity. However, when adding RAM, programmers can fill of the SRAM with data from the serial flash ROM and use them as traditional cartridge ROM, for example to emulate traditional 16KB non-bank switched cartridges. In addition, they can use it as a RAM expansion. I'm sure C64 developers that try to push the limits of the machine will appreciate having 16KB of additional RAM available.

Therefore the cartridge will also have 16KB of SRAM available.

Cartridge design - schematic overview

The components of the cartridge are shown below:



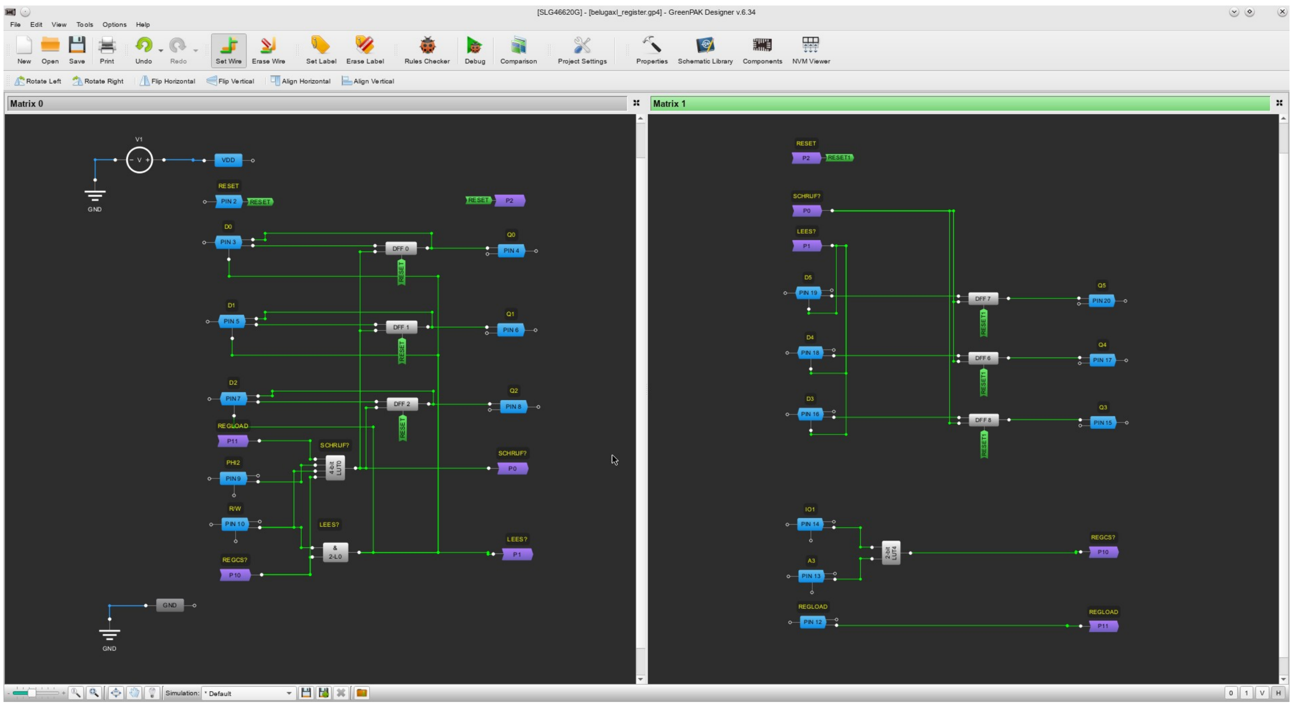
The conversion between QPI and OPI is performend by a 74x257 (from C64 to flash) and 74x374 (from flash to C64). We will need a register to control to memory mapping. The 74x174 and 74x273 are popular components to implement cartridge registers, but let's use a GreenPAK to implement the register. We can afford the Greenpak and Greenpak will have two advantages:

- Besides writing to the register, we will also be able to read from the register. The inability to read from cartridge registers is an annoyance amongst C64 programmers, sometimes you just want to change a single bit after all.
- 74x174 and 74x273 need additional chips to generate the control signals of these chips. The GreenPAK will be able to generate its own control signals, so this saves components.

We will still need control signals for the 74x257 and 74x374. We need to generate the clock signal and chip select signal for the flash memory, we need to generate the chip select signal for the SRAM and we need to generate the Commodore 64 PLA control signals. This will be done inside a second chip, the Controller GreenPAK.

Register GreenPAK

The register is easily implemented in GreenPAK. I am using an SLG46620. I was able to implement a 6-bit register that generates its own control signals. It makes itself visible in C64 memory at memory address $DE08. The current implementation uses the PHI2 clock signal to determine the moment to read the C64 data bus. It is still possible to free up the PHI2 pin by using delay components inside the GreenPAK to delay the IO1 signal until the bus is stable and then read it.



I have used 4 bits of the control register until now to implement memory mapping control. 2 bits remain available to implement additional features.

Memory mapping

We need to give the programmer control over how the cartridge presents itself in the memory of the Commodore 64. I am using 3 bits of the register to allow the programmer to select between 8 different modes. The decision to use 3 bits is because I want to be able to at least have a normal mode, 8K traditional cartridge mode (from SRAM) 16K traditional cartridge mode (from SRAM) and a writeable SRAM mode. As I need modes where the Sequential Access Memory appears as well, 2 bits are not enough.

For the time being, these modes are as follows:

	Description	ROML REGION	BASIC/ROMH REGION	UPPER RAM REGION	I/O REGION	KERNAL REGION
Mode / Address		$8000..$9FFF	$A000..$BFFF	$C000..$CFFF	$D000..$DFFF	$E000..$FFFF
0	Normal mode, cartridge inactive	C64 RAM	BASIC	C64 RAM	Normal I/O	KERNAL
1	8KB cartridge SRAM	Cart SRAM readonly	BASIC	C64 RAM	Normal I/O	KERNAL
2	16KB cartridge SRAM	Cart SRAM readonly	Cart SRAM readonly	C64 RAM	Normal I/O	KERNAL
3	SRAM writable	Cart SRAM R/W	BASIC	Unallocated	Ultimax I/O	Cart SRAM R/W
4	SRAM ROMH	C64 RAM	SRAM readonly	C64 RAM	Normal I/O	KERNAL
5	SAM 8K	Cart SAM	BASIC	Unallocated	Ultimax I/O	KERNAL
6	SAM 16K	Cart SAM	Cart SAM	Unallocated	Ultimax I/O	KERNAL
7	SRAM/SAM mix	SRAM R/W	Cart SAM	Unallocated	Ultimax I/O	KERNAL

- Mode 0 allows the programmer to switch of the cartridge mapping in memory. Modes 1 and 2 implement traditional cartridge modes based on SRAM (which is read-only). Mode 3 makes the SRAM writable by the programmer.
- Mode 4 makes SRAM appear in the ROMH region only. This is normally the location of the BASIC interpreter. This will allow people to design cartridges that replace the BASIC interpreter in the C64.
- Mode 5 allows sequential access memory in the ROML region. This mode will likely be used to boot the cartridge.
- Mode 6 is a mode with 16K sequential access memory. I'm not sure wether this is really needed, but I have a mode available. It could be changed if I later think of something more useful.
- Mode 7 has both SRAM and sequential access memory visible in the memory map. This mode might be useful for situations where the cartridge is normally hidden (mode 0), but has some system vectors hooken, which dynamically activate the cartridge (into this mode 7). Having the sequential access memory also visible allows these routines to access the huge flash memory as well.

The Commedore 64 PLA has 2 inputs on the cartridge port, called EXROM and GAME to select the cartridge mapping mode:

Mode	EXROM	GAME
Normal	1	1
8K cartridge	0	1
Ultimax	1	0
16K cartridge	0	0

(The upper RAM region is sometimes unallocated. The reason is that the Commodore 128 detects the presence of a Commodore 64 cartridge in the I/O region. By activating Ultimax mode in the I/O region, we make the Commodore 128 start in C64 mode rather than its native mode. However an undesired side effect is that the upper RAM region is deallocated (Ultimax mode deactivates internal RAM). This would be solvable by an extra address bus bit, but combinatorial logic will be already complex enough with the current situation.)

In order to implement the above memory map, we need to drive EXROM and GAME with the right value depending on the address bus and/or ROML/ROMH signals. Using ROML/ROMH signals where it is possible has preference, since it gives the programmer more control over the memory map via the processor port register at $0001 in the C64 memory.

Converting the above table results in:

	Description	**ROML REGION**	**BASIC/ROMH REGION**	**UPPER RAM REGION**	**I/O REGION**	**KERNAL REGION**
Mode / Address		**$8000..$9FFF**	**$A000..$BFFF**	**$C000..$CFFF**	**$D000..$DFFF**	**$E000..$FFFF**
0	Normal mode, cartridge inactive	Normal	Normal	Normal	Normal I/O	Normal
1	8KB cartridge SRAM	8K crt	8K crt	8K crt	8K crt	8K crt
2	16KB cartridge SRAM	16k crt	16k crt	16k crt	16k crt	16k crt
3	SRAM writable	Ultimax	Normal	Ultimax	Ultimax	Ultimax
4	SRAM ROMH	Normal	16k crt	Normal	Normal	Normal
5	SAM 8K	8K crt	8k crt	Ultimax	Ultimax	Normal
6	SAM 16K	16k crt	16k crt	Ultimax	Ultimax	Normal
7	SRAM/SAM mix	Ultimax	16k crt	Ultimax	Ultimax	Normal

The Controller Greenpak will have to drive the EXROM and GAME lines according to the above table.

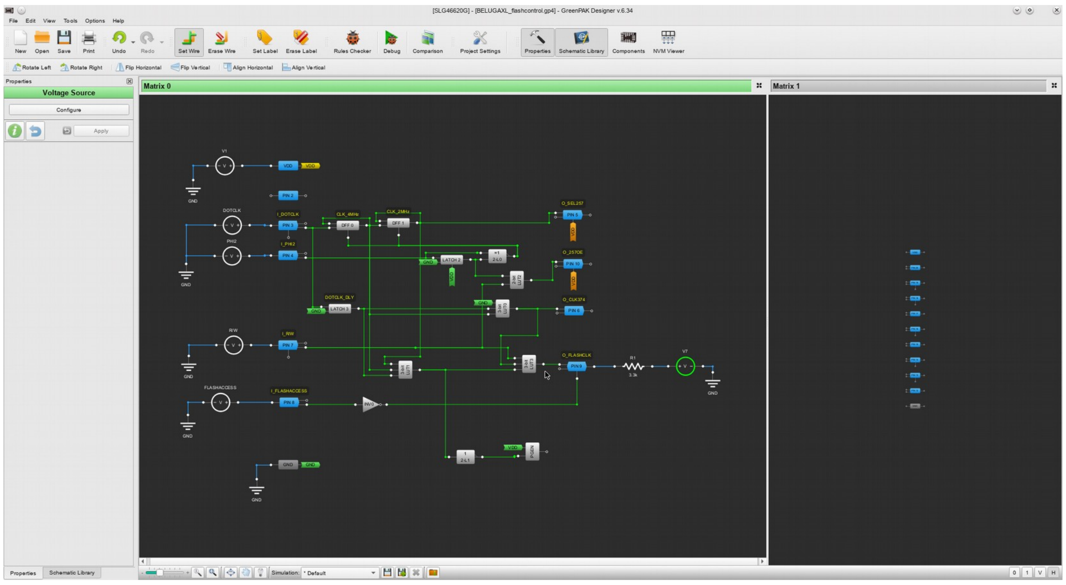
To give the programmer even more control over the memory map, the SRAM or SAM can also appear in the IO2 region at $DF00..$DFFF. Bit 3 of the register controls wether SRAM or SAM is visible.

Controller Greenpak

We now know that the Controller Greenpak needs to:

- generate the control signals for the 74x257, 74x374 chips
- generate the flash chip select signal and the flash chip clock signal
- and generate the correct EXROM/GAME signals for the Commodore 64

Generating the control signals is already done in an SLG46620. Inputs are the DOTCLOCK, PHI2 clock, R/W line and (temporary) flash access signal, which indicates the C64 wants to read from sequential access memory:

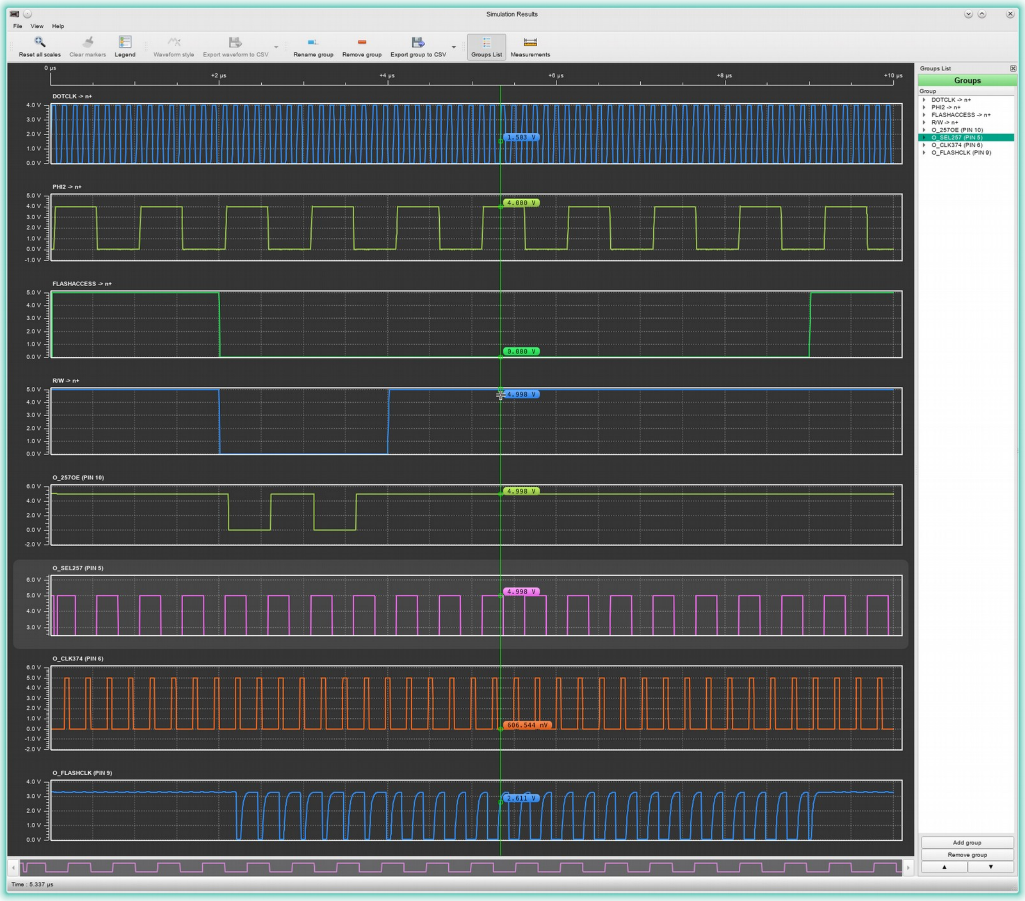


This went exceptionally well. Matrix 1 is still completely empty and there still is an unused pin in Matrix 0. This is a success for Greenpak, because I will tell you that I initially started to design the controller with ATF16V8 and 74xx logic chips. This went absolutely nowhere, I kept adding chip after chip until I concluded that my design had become way too complex.

The Greenpak appears to generate the right signals. The similation below simulates 5 consecutive flash access clock cycles. The first two are writes to the flash memory, the last three are reads from the flash memory.

You can see that:

- OE for the 74x257 is only generated during writes to the flash memory
- The flash clock is only generated during flash access
- Two flash clocks are generated per half C64 cycle. So for example when PHI2 is high (this is when the C64 CPU has the bus) there are two flash clocks while PHI2 is high, in order to send both D0..D3 and D4..D7 of the data bus to the flash memory in sequence.
- The timing for the flash memory clock is different for read and write cycles



While my reaction on the forum might suggest I consider moving from GreenPAK to GAL, it is actually the other way around. GreenPAK is going to make this cartridge happen.

However, work remains, the controller still needs to:

- Generate the EXROM and GAME signals
- Generate chip select signals
- Generate the initialization sequence to the flash during boot

The last point will not be a problem for the GreenPAK, the pattern generator will be very useful. However generating the remaining signals appears to be problematic.

As I already tried to implement the controller on an ATF16V8, I already have the required logic equations in CUPL syntax. Since CUPL works quite well to write down logic equations in a readable way, I'm writing down my CUPL equations here.

The inputs are:

- C64mode2..0: The C64 mode bits from the Register GreenPAK
- IO2mode: The IO2 mode bit from the Register GreenPAK
- a15..a13: The upper 3 bits of the C64 address bus
- a3,a0: Another 2 bits of the C64 address bus
- IO1, IO2, ROML, ROMH: Commodore 64 PLA outputs

The outputs to be generated are:

- flashaccess: Access from the C64 to flash memory
- flashcs: Chip select signal to flash memory
- exrom, game: Discussed before

- sramcs: Chip select signal for the SRAM memory

The logic equations are:

```
FIELD c64mode = [i_c64mode2..0];
FIELD addr = [i_a15..13];

/* 8K or 16K CRT*/
o_exrom = !(c64mode:1 #
            c64mode:2 #
            c64mode:4 & addr:[A000..BFFF] #
            c64mode:5 & !addr:[C000..DFFF] #
            c64mode:6 & !addr:[C000..DFFF] #
            c64mode:7 & addr:[A000..BFFF]);

/*16K CRT or Ultimax*/
o_game = !(c64mode:2 #
           c64mode:3 & (addr:[A000..BFFF] # addr:[C000..FFFF]) #
           c64mode:4 & addr:[A000..BFFF] #
           c64mode:5 & addr:[C000..DFFF] #
           c64mode:6 & addr:[C000..DFFF] #
           c64mode:7 & addr:[8000..DFFF]);

samaccess = c64mode:5 & !i_roml #
            c64mode:6 & !i_roml #
            c64mode:6 & !i_romh #
            c64mode:7 & !i_romh #
            i_io1 & !i_a3 & !i_a0 #
            i_io2mode & !i_io2;
o_flashaccess = !(samaccess);

/* Note: This is how you implement an SR-flipflop inside an ATF16V8:
   flipflop is set on any access to sequential access memory,
   flipflop is reset on access to $DE01 (partially decoded): */
o_flashcs = !(samaccess #
             !o_flashcs & !(!i_io1 & !i_a3 & i_a0));

o_sramcs = !(c64mode:1 & !i_roml #
             c64mode:2 & !i_roml #
             c64mode:3 & !i_roml #
             c64mode:3 & !i_romh #
             c64mode:7 & !i_roml #
             !i_io2mode & !i_io2);
```

Now the question:

Can this be implemented inside a GreenPAK?

I see no alternative for building a 74LS138 style binary decoder, because I need all values of "c64mode" in the logic equations except 0. Because we don't need value 0, we need "only" 7 3-bit LUTs.

We need 3-bit LUTs as well for the address ranges used, for example addr[8000:BFFF] will require a 3-bit LUT.

Then we need quite a few LUTS to implement the OR functions in the logic equations.

Note that:
- The controller does not need to fit inside a single GreenPAK
- Using an ATF16V8 is in combation with a GreenPAK is still an option
- Using 74xx chips in combination with a GreenPAK is an option
- Nothing in the design is set in stone. If something needs to be modified for ease of implementation, it can be modified.