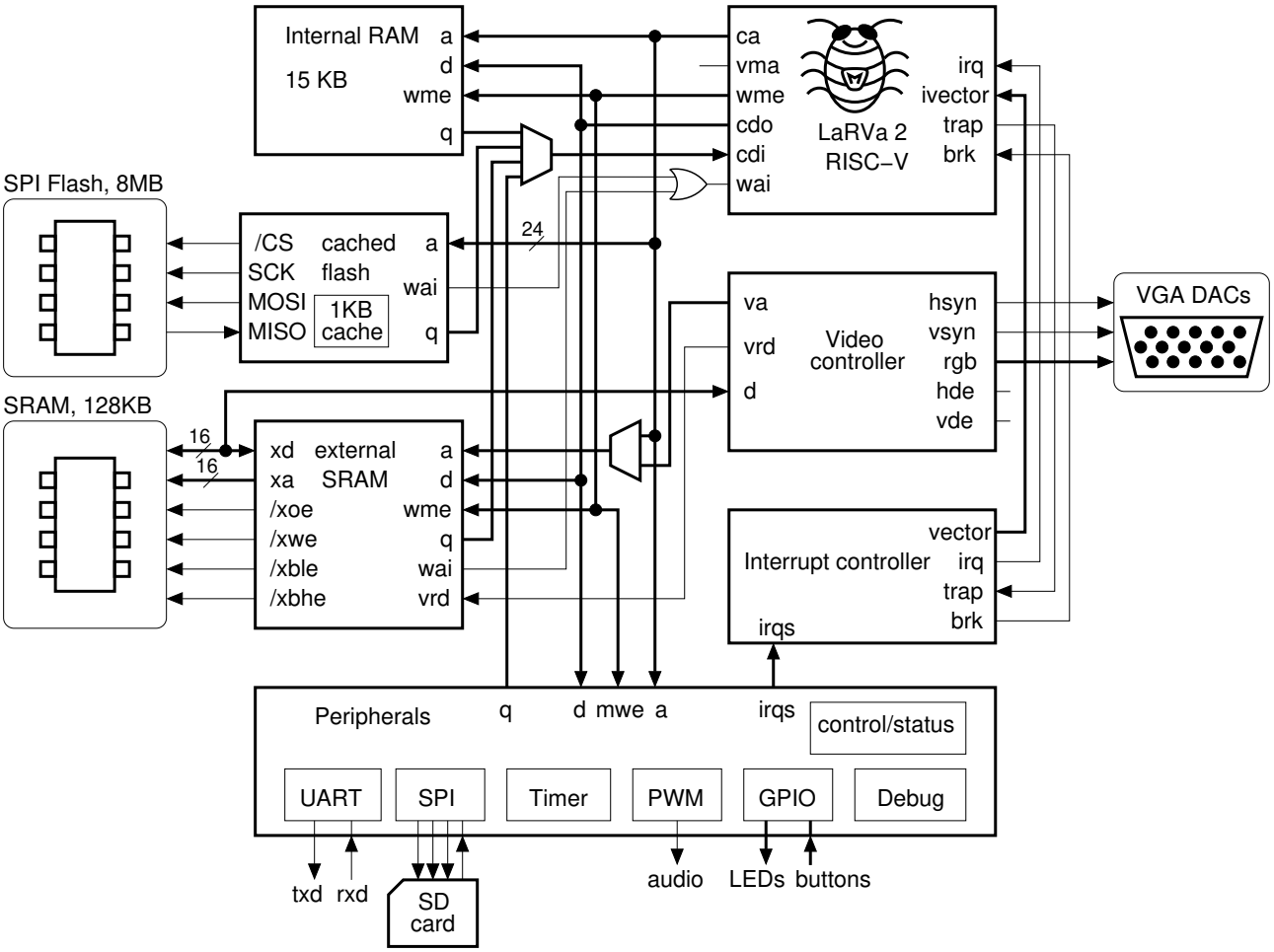
LaRVa-based computer

Jesús Arias

1 System description



The above figure shows the block diagram of the system, that includes in a ICE40HX4K FPGA a laRVa-2 CPU along with 15KB of fast internal RAM, a controller for an SPI-flash with 1KB of cache, an interface for an external SRAM with 128KB of memory and a 16-bit data bus, and the following peripherals:

- A VGA video controller with its framebuffer located on the external SRAM and 4 possible graphics modes:
  - 640×480, 1bpp (2 colors). Framebuffer size 38400 bytes (37.5KB)
  - 640×480, 3bpp (8 colors). 5-pixels per 16-bit. Framebuffer size: 122880 bytes (120KB)
  - 320×240, 8bpp (256 colors). 2-pixels per 16-bit. Framebuffer size: 76800 bytes (75KB)
  - 320×240, 12bpp (4096 colors). Packed: 4 pixels into three 16-bit words. Framebuffer size: 115200 bytes (112.5KB)
- An Universal Asynchronous Receiver/Transmitter, UART, that provides a bidirectional communication channel with the outside World.

- A simple second 8-bit SPI controller for the interface of an SD card.
- A free running timer counter for time measurement. Its value is compared with that of a matching register, and an interrupt request can be posted to the core on matches.
- A PWM generator that runs using the horizontal counter of the video controller, resulting in a 31.25KHz sampling rate and 640 possible PWM levels (9.3 bits)
- General purpose inputs and outputs.
- Debugging support that includes:
  - Single-step interrupt when on user mode.
  - Break (control-C) interrupt.
  - Two hardware breakpoints.
  - Convenience flags: “control-C” and “screen dirty”.
- An interrupt enable register with one enable bit for each interrupt source.
- A Vectored Interrupt Controller with up to 8 different programmable interrupt vectors. These vectors are presented to the CPU when some interrupt is requested (including the execution of ECALL and EBREAK opcodes)

The RAM and the peripheral registers are mapped to the address space by means of address decoding blocks (mainly for writings) and multiplexers (for reads).

1.1 Memory map

The decoding of the address lines along with the ’vma’ signal and the byte-write signals, ’mwe[3:0]’ results in the mapping of the memories and peripherals into the address space of the CPU. As data buses are 32 bits wide but the address is for bytes, the two lowest address bits are always 00 and they aren’t included in the address bus. The four byte-write signals are provided instead because we have to select which bytes to write during Store-Byte and Store-Halfword instructions. The memory map is:

Write data	address bits	mwe	size	Base address
IRAM	0000.xxxx.xxxx.xxxx.xxAA.AAAA.AAAA.AA00	—-	w,h,b	0x00000000
XRAM	0001.xxxx.xxxx.xxxA.AAAA.AAAA.AAAA.AA00	—-	w,h,b	0x10000000
FLASH	001x.xxxx.AAAA.AAAA.AAAA.AAAA.AAAA.AA00	0000	w,h,b	0x20000000
I/O	111x.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xx00	—-	w,h,b	0xE0000000
UART TX	111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x000	xxx1	b	0xE0000000
GPO	111x.xxxx.xxxx.xxxx.xxxx.xxxx.001x.xx00	xxx1	b	0xE0000020
CONTROL	111x.xxxx.xxxx.xxxx.xxxx.xxxx.010x.xx00	xxx1	b	0xE0000040
STATUS[1]		xx1x	b	0xE0000041
TMATCH[2]	111x.xxxx.xxxx.xxxx.xxxx.xxxx.011x.xx00	1111	w	0xE0000060
BGcolor	111x.xxxx.xxxx.xxxx.xxxx.xxxx.100x.xx00	xxx1	b	0xE0000080
FGcolor		xx1x	b	0xE0000081
VBase		11xx	h	0xE0000082
PWM	111x.xxxx.xxxx.xxxx.xxxx.xxxx.101x.xx00	xx11	h	0xE00000A0
SPI TX	111x.xxxx.xxxx.xxxx.xxxx.xxxx.101x.xx00	x1xx	b	0xE00000A2
IRQEN	111x.xxxx.xxxx.xxxx.xxxx.xxxx.110x.0x00	xxx1	b	0xE00000C0
BRK1	111x.xxxx.xxxx.xxxx.xxxx.xxxx.110x.1000	1111	w	0xE00000C8
BRK2	111x.xxxx.xxxx.xxxx.xxxx.xxxx.110x.1100	1111	w	0xE00000CC
VECTORS	111x.xxxx.xxxx.xxxx.xxxx.xxxx.111A.AA00	1111	w	0xE00000E0

The above table is for writes, where a strobe signal is generated for each register to be written. On the other hand, the mapping for reads is performed in the multiplexers for the CPU data input bus, resulting in the following table:

data read	address bits	size	Base address
IRAM	0000.xxxx.xxxx.xxxx.xxAA.AAAA.AAAA.AA00	w,h,b	0x00000000
XRAM	0001.xxxx.xxxx.xxxA.AAAA.AAAA.AAAA.AA00	w,h,b	0x10000000
FLASH	001x.xxxx.AAAA.AAAA.AAAA.AAAA.AAAA.AA00	w,h,b	0x20000000
I/O	111x.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xx00	w,h,b	0xE0000000
UART RX[3]	111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x000	b	0xE0000000
UART flags	111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x100	b	0xE0000004
GPO	111x.xxxx.xxxx.xxxx.xxxx.xxxx.001x.xx00	b	0xE0000020
GPIN		b	0xE0000021
CONTROL	111x.xxxx.xxxx.xxxx.xxxx.xxxx.010x.xx00	b	0xE0000040
STATUS		b	0xE0000041
TCNT	111x.xxxx.xxxx.xxxx.xxxx.xxxx.011x.xx00	w	0xE0000060
BGcolor	111x.xxxx.xxxx.xxxx.xxxx.xxxx.100x.xx00	b	0xE0000080
FGcolor		b	0xE0000081
Video base		h	0xE0000082
SPI RX	111x.xxxx.xxxx.xxxx.xxxx.xxxx.101x.xx00	b	0xE00000A2
IRQEN	111x.xxxx.xxxx.xxxx.xxxx.xxxx.110x.0x00	b	0xE00000E0
BRK1	111x.xxxx.xxxx.xxxx.xxxx.xxxx.110x.1000	w	0xE00000E8
BRK2	111x.xxxx.xxxx.xxxx.xxxx.xxxx.110x.1100	w	0xE00000EC

[1]Many flags are cleared when writing its corresponding bits with 1. Zeroes have no effect

[2]Writing this register also clears the timer IRQ flag

[3]Reading this register also clears the RXvalid and RXoverrun flags

2 Clock

The input clock for this system is a 50MHz signal obtained from a PLL. This frequency is used in the Flash SPI controller only, with the rest of the system running at 25MHz. In particular 25MHz is a convenient frequency because it is the one required for the pixel clock of the video controller. A simple toggle flip-flop is used for frequency division.

Clock	Freq	Blocks
clkin	50MHz	Cached Flash controller
clk	25MHz	CPU, video, peripherals

3 Memories

3.1 Internal RAM

The FPGA has 32 512-byte BRAMs, totaling 16KB. But we must save two BRAMs for the cache of the flash controller, so, only 15KB are available. In order to map this unusual amount of memory four blocks are actually used, with 8KB, 4KB, 2KB, and 1KB each, along with the corresponding selection logic and output multiplexing.

The other peculiar aspect of the BRAMs is their synchronous design. So, here we resort to the following timing:

- After the rising edge of 'clk' a new address value is presented to the IRAM.
- On the falling edges of 'clk' the IRAM is read. That means a new data is available at its output after a short delay.
- On the following rising edge the IRAM is written if required with the data presented at its input.

3.2 Cached Flash

SPI flash memories are a very convenient place to store programs for our FPGA cores due to their high capacity and simple interface, but they also have a very big problem: the serial data transfer also means they are very slow. As an example consider the case where a 32-bit opcode has to be read from such a memory. It will take:

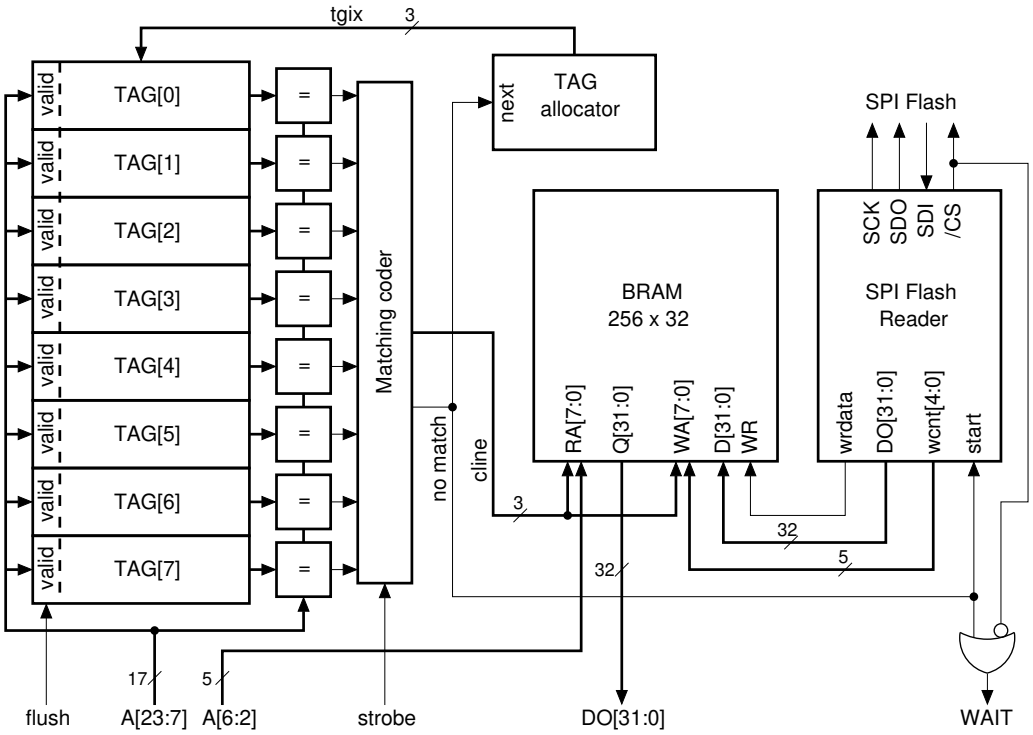
- 1 clock cycle for the activation of the /CS input of the flash (/CS has to go high between successive flash reads)
- 8 clock cycles for sending the read command (0x03)
- 24 clock cycles for the sending of the address of the data. 24 bits allows a maximum capacity of 16MB.
- 32 clock cycles for the reading of the data

This makes the flash read 65 cycles long while the same 32-bit data can be read from an internal RAM in just a single cycle. Therefore, if the SPI flash stores code its execution is going to be 65 times slower than when running from internal RAM. This makes the SPI flash almost unusable for code storage.

Unless we also provide a cache memory for the flash, a well known remedy for slow but big memories. The cache stores an small amount of data copied from the flash, and the main idea is to read the data from the fast cache if it is already there, and only wait a long number of cycles if it is not, also keeping a copy for future reads. As programs execute code in loops there is a good chance to have the loop code in the cache after the first iteration and to run the following iterations much faster.

With this goal in mind I started the design of a simple SPI flash interface with cache, with very good results.

3.2.1 Cache implementation



In the above diagram the main blocks of the interface are presented. Its main block is a 256x32, dual port, RAM that requires 2 BRAM blocks in the FPGA (2 blocks is the minimum for 32-bit memories), totaling 1KByte of cache. The dual port feature isn't really needed because there are no simultaneous reads and writes, but the RAM blocks of the FPGA are of this type and the use of two ports can save an address multiplexer. This memory is divided into 8 lines of 32 words each (128 bytes), and each line has an associated tag register where the flash address of the line is stored along with a valid bit. All valid bits start with a false value and they can also be cleared thanks to an asynchronous "flush" input. These bits get set when the corresponding cache line is filled from the flash.

When the core tries to read a data word from the flash (signaled by asserting "strobe") the 17 most significant bits of the flash address are compared in parallel with the contents of the 8 tag registers. If a match happens the 3 bits of the "cline" signal are assigned to the number of the tag register that matches the address, and the particular word inside that cache line is selected by the 5 lower bits of the address.

But if the address don't match any tag register (with a valid bit set), then the "nomatch" signal is asserted. This has the following effects:

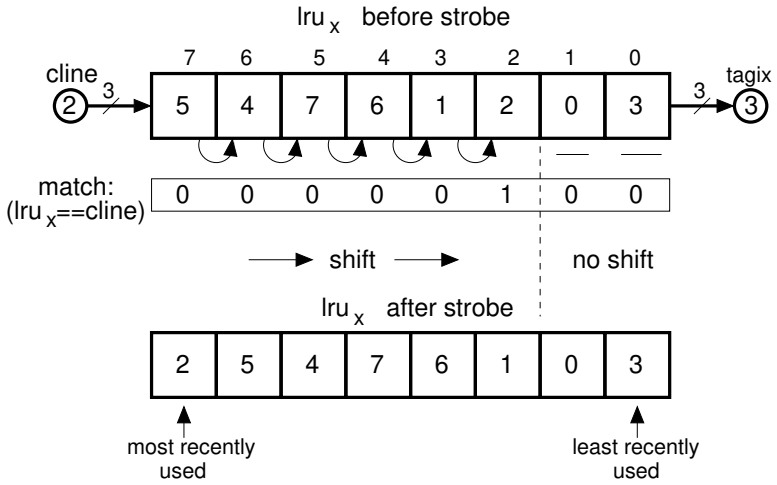
- The core is forced to stop its execution via a "wait" output (the actual name is "wai" because "wait" is a Verilog keyword).
- The current address is written to the tag register selected by "tagix" and its valid bit is set.
- The Flash reader starts a read operation for the upload of the whole cache line. This read will last 1056 cycles until the deassertion of the "wait" signal. After the initial 32 cycles for the read command and flash address a word is retrieved each successive 32 cycles and written to the cache RAM.

The flash reader is basically a 32-bit shift register that stores both the data to be sent to the flash (read command plus address) and the words retrieved from the memory. Along with this register we also have an 11-bit counter that keeps track of the state of the reader, generating write pulses to the cache every 32 cycles during the data phase, and stopping the read when complete.

After the reading of a cache line the selection of the next tag register also has to be performed and there are various possible strategies to follow:

- Sequential allocation: Every time a cache line is read the "tagix" value is incremented. This only requires a 3-bit counter, a very simple block, and its performance is probably not very optimal because all cache lines are discarded without any regard to memory usage.
- Random allocation: The "tagix" value is the 3 LSBs of a pseudorandom number generator (a 7-bit LFSR). Its performance is probably as bad as the former but it is also less dependent on the code actually being executed.
- Least Recently Used allocation. The allocator block monitorizes the tag registers selected during reads and it provides the value of the least recently used one as the "tagix" value. This block is the more complex of the three versions tested, but it can also offer the best cache performance and therefore this is the solution selected. It is described next:

3.2.2 The LRU allocator



The LRU algorithm was a bit complex and it took some time to figure out a simple hardware implementation. At the end I resorted to follow the approach shown in the above figure, that includes:

- An eight position shift register for 3-bit values, lru, where each datum starts with $lru_x = x$, so, there are no repeated values in the whole register.
- A 3-bit match logic for each register datum. This logic provides a TRUE output when the contents of an lru_x datum matches the *cline* signal.
- A conditional shifting logic where only the data on the left of the matching datum, and the matching datum itself, are shifted towards the right. The *cline* value is shifted into the leftmost datum, lru_7.

In this way the register always contains all the possible values of the tag indexes, with the most recently used tag stored in lru_7, and the least recently used in lru_0. The register is updated when the "strobe" input is active and some tag register matches the address. The content of lru_0 is presented as the next cache line to overwrite when "nomatch" is asserted.

3.2.3 Results

In order to test the usefulness of the cache a benchmark program was stored into the SPI flash and its execution time measured. This program was the same fixed-point FFT test that was already executed in the internal RAM where each word was read in a single clock cycle. It's size is about 6.28KB, bigger than the available cache, and it also includes a 2KB sine table that is stored along the code, so, the cache is going to have a hard time mapping pieces of code intermixed with parts of the data table.

The three different strategies for tag allocation were tested and the results are listed in the following table:

TAG allocation	Logic cells	Exec. time (s)	Average cycles/read
control (running on RAM)	3431	6.573	1
Sequential	+453	11.975	1.82
Random	+416	14.774	2.25
LRU	+550	9.527	1.45

Apart from the extra logic cells, the flash controller also requires two RAM blocks in all the cases. Also, the extra logic cell count isn't exactly the size of the controller because some glue logic (address decoding and input data multiplexing) is also required in the system for its attachment and Yosys is smart enough to remove all that logic when not really used (control case).

And talking about performance, the random tag allocator is the worst one while the LRU shows the maximum gain in execution time. We must remark that a direct execution from flash without caching will have 65 cycles per word read instead of 1.45, so, the cache is doing a very good job.

3.2.4 Design trade-offs and possible improvements

- Granularity vs latency

  In this design each cache line stores 32 words, but this was an arbitrary design decision. A design with 16 words per line would require double the number of tag registers and related logic, but the time needed to read a cache line is almost halfted. This would be desirable from the interrupt latency point of view because no interrupts can be serviced while the core is waiting for a cache line fill.

  Alternatively, if we don't want to increase the amount of logic of the controller, we can resort to keep the size of the cache lines as it is but to increase the speed of the flash memory by means of using a faster clock or dual or quad flash read commands.

- Speeding up flash reads

  The laRva core runs no much faster than 20MHz, but the SPI flash (a Micron model) can run up to 108MHz according to its datasheet. Therefore we can use a 4 times faster clock for the memory, but we have to replace the read command (0x03) for the "fast read" command (0x0B) and also to include some dummy clock cycles between the command and the data. This would reduce the latency to about 1/4 of the current value (now about $40\mu s$)

  Also, modern SPI flash chips usually allows the reading of their data 2 or 4 bits at a time. The use of these modes will require tristate pins for the MISO and MOSI signals in dual modes, and also for the /HOLD and /WP signals in quad modes, and flash chips usually also require some vendor-dependent configuration commands prior to use. These modes could be used in combination with a faster clock, allowing for a maximum reduction in latency to about 1/16 of the current value.

- Code and data

  The flash can store both code and constant data, but this intermixing could have a negative impact on performance. Data tables are usually read sequentially and this will have the effect of overwriting all the cache lines with data that is no longer needed while loosing most of the useful code meanwhile.

  Probably it would be better to differentiate between code and data reads and to reserve some specific tag registers only for data, maybe a single one. In order to support this we need some way to signal a data read in the core. This signal will be active during the execution cycle of the Load instructions, and, when asserted, only the tag registers reserved for data will be selected in the cache, leaving all the code already loaded untouched.

- Wraparounds

  In the current design when a cache line has to be filled it is read starting from offset #0, meaning that we have to wait until all the line is complete before resuming execution. But, it could also be possible to start the reading at the particular offset we have for the current address. In this way the data we need would be available after 65 cycles instead of 1057 and we could resume the execution while the cache continues reading the rest of the line. Well, this can have some complications:

  - In order to read the whole line we have to stop the reading when reaching the offset #31 and to wraparound to offset #0. This would imply the sending of two read commands with different lengths to the flash unless the data read is the first of the cache line. Some flash chips can perform the wraparound automatically if properly configured, thus allowing the update of the whole line with a single read command.
  - Now the tag address is no longer enough to know is the cache content is valid. The word counter of the flash reader also have to be checked and the core put into a wait state if a read is made while the cache line is still being uploaded and the word counter is below the address requested (with the wraparound included in the check)

Of all these possible improvements only the second was actually considered and the flash controller is running with double the frequency than the rest of the system. With a 50MHz clock the flash can still use its read command without dummy bytes, and the time the CPU is halted in the event of a cache miss is 21.14μs. This time is lower than the fastest interrupt period (horizontal display enable: 32μs between interrupts).

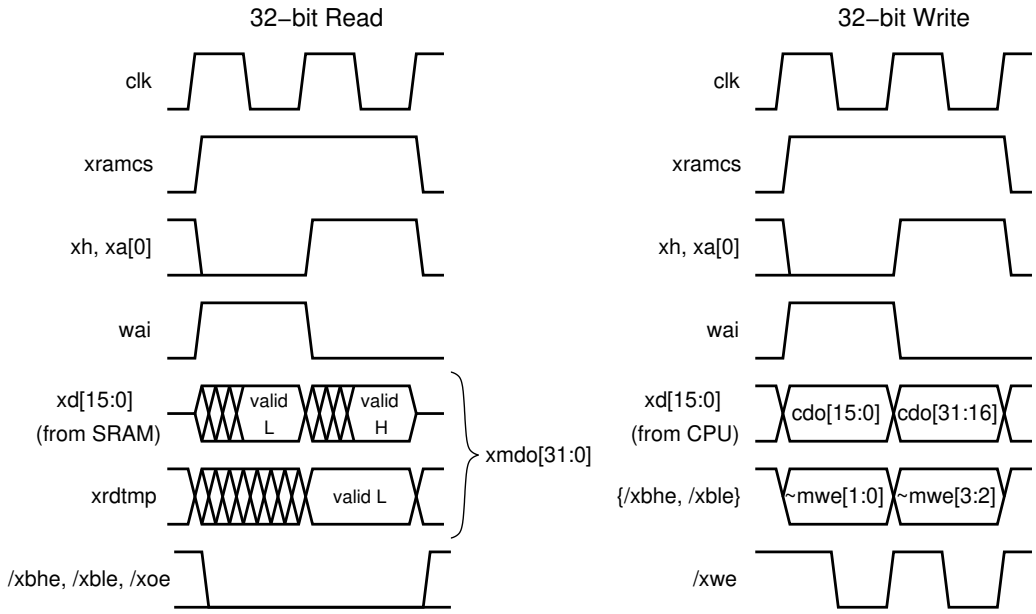
One detail is worth mentioning: The sampling of the MISO input is actually done on the falling edges of SCK instead of the rising edges. This makes the flash reading much more reliable.

The LaRVA-2 core has a configurable reset value for the PC and in this case it has the value 0x20030000, meaning the execution starts 192KB after the beginning of the flash. (The first flash blocks are reserved for the FPGA bitstream). So, the system boots from the SPI flash.

3.3 External RAM

An external SRAM with 128KB and 16-bit data bus is available on the ICECREAM and SIMRETRO boards, and this memory was also interfaced to the computer after overcoming a few problems:

- The memory has a 16-bit data bus while the core uses 32-bits. This was solved by adding a wait state during reads and writes, so, a 32-bit read or write is converted into two 16-bit operations. A temporary 16-bit register, 'xrdtmp', is included to store the low halfword during reads. The content of this register is presented along with the memory data bus during the second cycle as a 32-bit word. Some details of this timing are shown in the next chronograph:



- The data bus is bidirectional. But now the new "yosys" versions can deal with these buses quite well and no more vendor-specific modules (SB_IO) are needed. All we have to do is to define their pins as "inout", and to assign to them a floating value, 'Z', during reads:

```
assign xd = rd ? 16'hZZZZ : bus_out;
assign bus_in = xd;
```

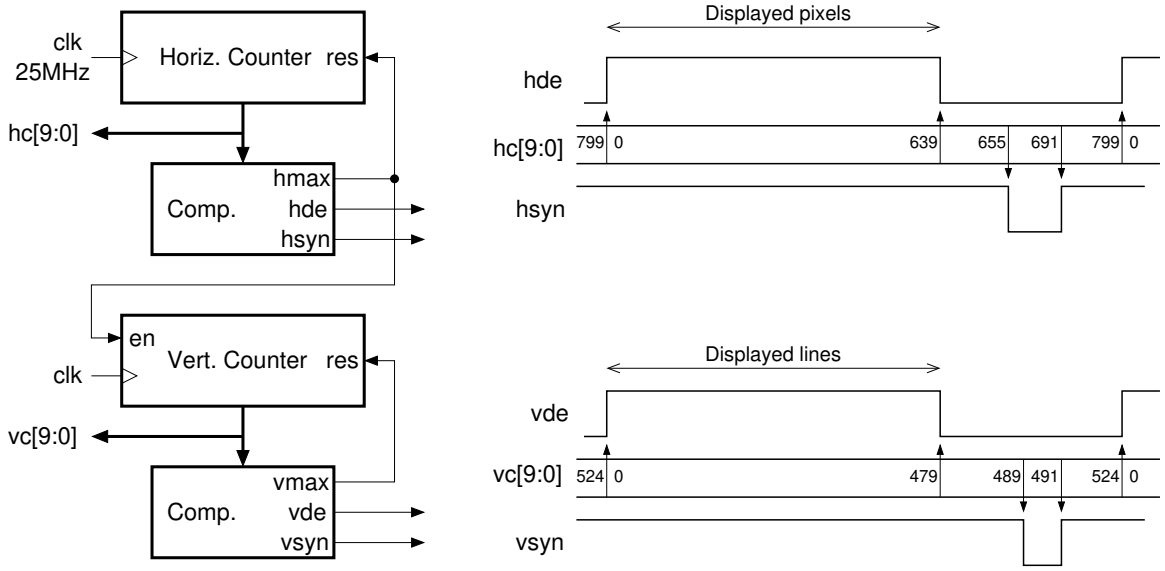
- The external RAM is shared with the video controller. That was needed because the internal memory is too small for the required framebuffer (its size ranges from 37.5KB to 120KB). In this case video reads are single cycle, 16-bit, halfwords. If a video read happens at the same time the CPU is reading or writing the external memory an additional wait state is inserted, resulting in a 3-cycle access. The video controller always has more priority than the CPU.

So, the external memory is an slow one but it has a decent size, in addition to store the framebuffer.

4 Video controller

This is an special peripheral because it is a memory master that reads the video data from the external RAM and sends it to the color component DACs of the VGA adapter, along with generating the horizontal and vertical timing pulses.

The video controller has three main components. The first one is the timing generator whose diagram and waveforms are presented next. It includes a pixel counter, 'hc', and a line counter, 'vc' that, in addition to the VGA's sync pulses, also generates the horizontal display enable, 'hde', and vertical display enable, 'vde' signals. When these two signals are one we are actively refreshing the image on the screen with the pixel data read from the external RAM.



The second component is the video address generator. This is a 16-bit register that stores the halfword address for the next group of pixels. It gets loaded with the video base address when 'vde' is low, and then its value is incremented with every video memory read. Well, that's true for video modes with 480 lines, but for the two modes with 240 lines things are a little different because now the lines have to be repeated two times. So, when 'vc' is even, and we are at the end of the line, the number of words per line is subtracted from the video address register and the same line is displayed again.

And finally, the pixel shifter is the last component of the video controller. But in fact, we have 4 pixel shifters, one per video mode, because video data is packed into memory in quite different ways depending on the mode. But, in all cases the LSB bits of the halfwords retrieved from the external RAM are displayed first. A video read, 'vrd', signal is also generated for each mode and the proper one selected. These are the details of each mode:

- On video mode #0 (640x480, 1bpp) a halfword is read every 16 cycles (when 'hc[3:0]' is zero), and stored into a 16-bit right-shift register. Then, the register is shifted one bit at a time and its LSB bit selects between the foreground and background colors. These colors are extended from 8 bits (BGR233 format) to 12 and presented as the video output.
- On video mode #1 (640x480, 3bpp) a halfword is read every 5 cycles (an additional, modulus-5 counter is included for that). The MSB is discarded and the remaining 15 bits are stored into a shift register. This register shifts right 3 bits at a time, and its 3 LSB bits are the color for the pixel (BGR111 format). This color is extended to 12 bits and presented as the video output.
- On video mode #2 (320x240, 8bpp) a halfword is read every two pixels (or 4 cycles, because pixels are repeated two times) and stored into a 16-bit shift register. This register shifts 8 bits at a time, and its lower 8 bits are the color of the pixel (BGR233 format). The color is extended to 12 bits and presented at the output.
- On video mode #3 (320x240, 12bpp), a halfword is read every three out of four pixels (3 out of 8 cycles). The lower 12 bits of the halfwords are presented as the output pixel with its BGR444 color, while the higher 4-bits are stored on temporary blue, green, and red component registers for the fourth pixel, and are presented at the output at the proper time.

5 Peripherals

The peripheral address space is located at the upper 512MB area (base address: 0xE0000000).

5.1 UART

A minimal UART was designed for interfacing. It has the following characteristics:

- 8 data bits, No Parity, 1 Stop bit, and fixed baud rate (115200 bps).
- No buffer register for TX. One byte buffer for RX.
- Interrupts on RX available and TX ready.

The UART has the following registers:

Register	address	size	Write	Read
UARTDAT	0xE0000000	b	TX data	RX data

Register	r/w	Address	size	Bits				
				4	3	2	1	0
UARTSTA	ro	0xE0000004	b	CTRL-C	OV	FE	TXRDY	RXVAL

- Bit #0, RXVAL: Set to one when a character is received, cleared when reading UARTDAT. This flag can request an RX interrupt if one.
- Bit #1, TXRDY: Set to one when the transmitter is idle (ready to transmit), or to zero if busy. This flag can request a TX interrupt if one.
- Bit #2, FE: Framing Error, set to one if the stop bit of the received character was zero.
- Bit #3, OV: Overrun, set to one if RXVAL was one when a character is received. Cleared reading UARTDAT.
- Bit #4, CTRL-C: Set to one if UARTDAT has the ASCII 3 value (ctrl-C) and RXVAL is one. This flag can request a debug interrupt if set.

5.2 GPIO

These are two general purpose, 8-bit, ports. One for output, GPO, and another for input, GPIN. Notice that some bits could be missing in some ports of the design.

Register	r/w	Address	size
GPO	rw	0xE0000020	b
GPIN	ro	0xE0000021	b

5.3 Control & Status

These are two 8-bit registers with the following bits:

Register	r/w	address	size	Bit						
				6	5	4	3	2	1	0
CONTROL	rw	0xE0000040	b	-	-	/SDCS	SPIFAST	-	MODE	
STATUS	rd	0xE0000041	b	BRK2F	BRK1F	SCRDTY	SPIBUSY	TIMIRQ	VDE	HDE
	wr			clr	clr	clr	-	-	clr[4]	clr[5]

[4]Clears IRQ, not VDE
[5]Clears IRQ, not HDE

CONTROL bits:

- MODE: sets the video mode:
  - 00 : 640x480, 1 bit per pixel with selectable colors.
  - 01 : 640x480, 3 bit per pixel, BGR111, 5 pixels per 16-bit halfword.
  - 10 : 320x240, 8 bit per pixel, BGR233, 2 pixels per 16-bit halfword.
  - 11 : 320x240, packed 12 bit per pixel, BGR444, 4 pixels per three 16-bit halfwords.
- SPIFAST: set the clock speed for the second SPI bus:
  - 0 : 25 MHz / 256 = 97.6 kHz (SD cards must use an slow clock until initialized)
  - 1 : 25 MHz
- /SDCS: this is the Chip Select line for the SD card attached to the second SPI bus.

STATUS bits:

- HDE: Horizontal Display Enable, set to one during the visible pixels of the video line. Writing a one to this flag clears the HDE interrupt request. Writing a zero has no effect.
- VDE: Vertical Display Enable, set to one during the visible lines of the video frame. Writing a one to this flag clears the VDE interrupt request. Writing a zero has no effect.
- TIMIRQ: Timer interrupt flag. Set to one after a matching and cleared by writing the matching register.
- SPIBUSY: Set to one when the second SPI bus controller is transmitting / receiving.
- SCRDTY: Screen dirty flag. Set to one when data is written to the UARTDAT register. Writing a one to this flag clears it. Writing a zero has no effect.
- BRK1F: Breakpoint 1 flag. Set to one when the address bus matches the BRK1 register. Writing a one to this flag clears it. Writing a zero has no effect.
- BRK2F: Breakpoint 2 flag. Set to one when the address bus matches the BRK2 register. Writing a one to this flag clears it. Writing a zero has no effect.

5.4 Timer & Match

Register	r/w	Address	size
TMATCH	wo	0xE0000060	w
TCNT	ro	0xE0000060	w

The timer, TCNT, is simply a 32 bit counter that gets incremented each clock cycle. This counter never stops, not even on resets.

The matching register, TMATCH, is a write-only, 32-bit register, that is compared to the current value of the counter. When the two values match an interrupt flag is set (TIMIRQ, bit #2 of the STATUS register). This flag can request a timer interrupt when one, and it gets cleared when a new value is written to TMATCH.

5.5 Video colors and base address

Register	r/w	address	size	
BGCOLOR	rw	0xE0000080	b	BGR233 color for 0 pixels
FGCOLOR	rw	0xE0000081	b	BGR233 color for 1 pixels
VIDEOBAS	rw	0xE0000082	h	Video base address

The BGCOLOR and FGCOLOR values are only used for video mode #0.

The video base address is the framebuffer address offset from the beginning of the external RAM divided by 2 (halfword address)

5.6 PWM

The PWM generator as a single write-only register:

Register	r/w	address	size	bits 9 to 0
PWM	wo	0xE00000A0	h	PWM level (0 to 640)

The PWM generator runs using the horizontal counter of the video controller, so, the horizontal display enable interrupt can also be used to update the PWM register.

In order to avoid glitches in the PWM waveform another 10-bit buffer register is placed between the PWM register and the waveform generator. This buffer is updated with a new PWM value on the falling edges of the HDE signal (when the HDE interrupt is also requested).

5.7 Second SPI

The second SPI controller has its data register at the following address:

Register	address	size	Write	Read
SPI	0xE00000A2	b	TX data	RX data

Also, in the CONTROL / STATUS registers there is a flag, SPIBUSY, and a control bit, SPIFAST. This, along with the chip-select signal of the external SD card (also in the CONTROL register) allows the reading and writing of files from the card.

A write to the SPI register starts a byte transaction between the controller and the slave device (the SD card) and sets SPIBUSY.

5.8 Hardware Breakpoints

The addresses of these registers are compared to the current address presented by the CPU. A match can cause a trap to the core.

Register	r/w	address	size	bits		
				31 to 2	1	0
BRK1	rw	0xE00000C8	w	address	C/D	EN
BRK2	rw	0xE00000CC	w	address	C/D	EN

EN: Breakpoint enabled if one.

C/D: Code / Data. A value of zero compares the address during instruction fetches, while a value of one compares the address during the execution of LOAD or STORE instructions.

There are also two flags in the STATUS register related to these hardware breakpoints.

A hardware trap differs from an interrupt in the following aspects:

- It activates the 'trap' output and gets the same interrupt vector than the execution of an ECALL / EBREAK instruction.
- Its latency is just one cycle instead of two.

Notice that code breakpoints jumps to the trap vector before the execution of the related instruction, while data breakpoints have the instruction executed before jumping to the trap vector.

These addresses lack the bits 0 and 1. (must be multiple of 4)

5.9 Vectored Interrupts

The interrupt subsystem includes three parts:

- An interrupt enable register, INTEN, where the desired interrupt sources can be enabled.
- A priority encoder that generates a vector number depending on the interrupts being requested.
- An interrupt vector table: An small dual-port RAM where the addresses of the related interrupt service routines are stored.

The bits of INTEN are:

Register	r/w	Address	size	Bit							
				7	6	5	4	3	2	1	0
INTEN	rw	0xE00000C0	b	SSTEP	CTRLC	-	TIMIE	UTXIE	URXIE	VDEIE	HDEIE

- HDEIE: Enable the Horizontal Display Enable interrupt. (falling edges)
- VDEIE: Enable the vertical Display Enable interrupt. (falling edges)
- URXIE: Enable the UART RX interrupt. (RXVAL requests the interrupt)
- UTXIE: Enable the UART TX interrupt. (TXRDY requests the interrupt)
- TIMIT: Enable the Timer match interrupt.
- CTRLC: Enable the 'ctrl-C' interrupt.
- SSTEP: Enable the single-step interrupt. If enabled, an interrupt is requested as soon as the core returns to user mode after executing MRET. Only a single instruction of the user program is executed before jumping to the ISR again.

And these are the related vectors ordered from high priority to low:

Vector	r/w	Address	Size	Sources
0	wo	0xE00000E0	w	ECALL/EBREAK, CTRLC, SSTEP, hard BRK
1	wo	0xE00000E4	w	HDE
3	wo	0xE00000EC	w	UART RX
2	wo	0xE00000E8	w	VDE
4	wo	0xE00000F0	w	UART TX
5	wo	0xE00000F4	w	TIMER

Notice the vector #0 is the main entry point to the debugger code and includes all the related trap and interrupt sources.

6 Summary

This design can turn the SIMRETRO board into a quite decent RISC-V computer. And also the ICECREAM board if a simple VGA adapter is added (an adapter for just the BGR111 mode only requires three 270Ω resistors). The amount of RAM is small, specially the fast internal RAM, but on the other hand we can have a quite big ROM and a virtually unlimited storage thanks to an SD card. The debugging features included can also be useful for the development of RISC-V code, and in fact, a debugger is provided as the main ROM program.

And, with 5430 logic cells we still have about 1/3 of the FPGA space available for further additions, like more peripherals, or the synthesis of an RV32IM core with the full 32-register set (this core will also require a few changes in the debugger program)