Cache for SPI flash

Jesús Arias

E.T.S.I. Telecomunicación. Valladolid. Spain

1 Introduction

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:

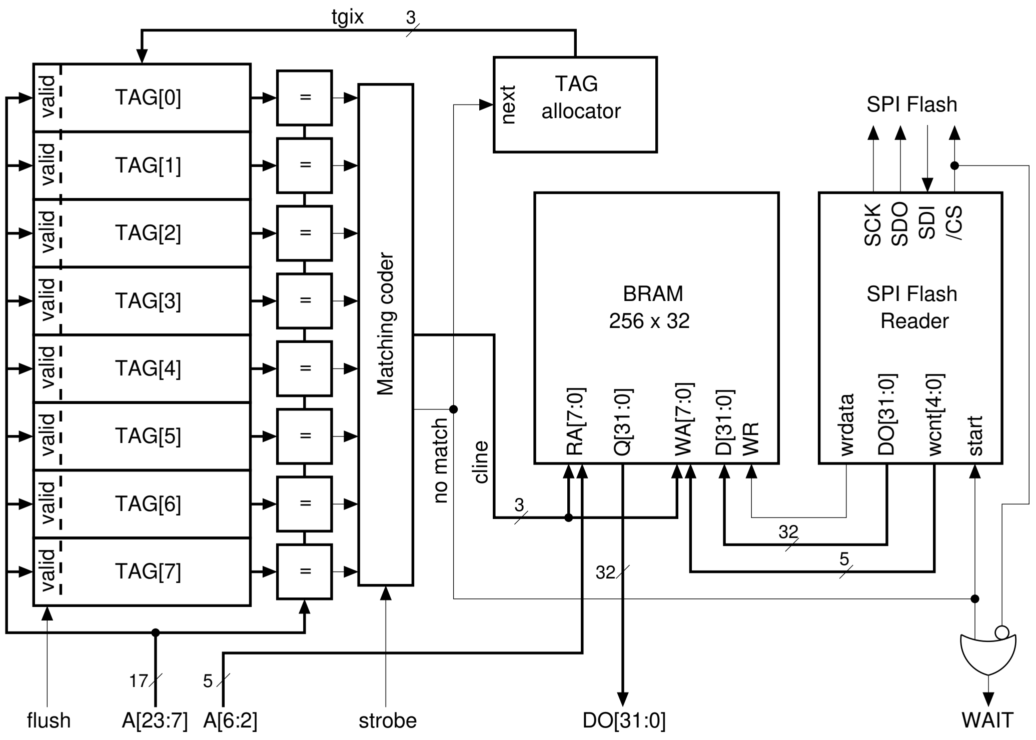
- 1 clock cycle for the activation of the /CS input of the flash (/CS has to go high between successive flash reads)
- 8 clock cycles for sending the read command (0x03)
- 24 clock cycles for the sending of the address of the data. 24 bits allows a maximum capacity of 16MB.
- 32 clock cycles for the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

2 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:

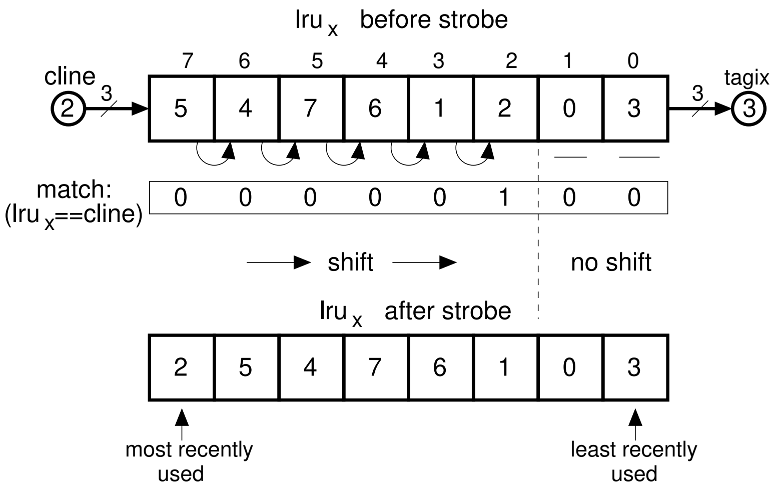
- The core is forced to stop its execution via a "wait" output (the actual name is "wai" because "wait" is a Verilog keyword).
- The current address is written to the tag register selected by "tagix" and its valid bit is set.
- The Flash reader starts a read operation for the upload of the whole cache 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. It is described next:

2.1 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 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.

4 Design trade-offs and possible improvements

4.1 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.

4.2 Speeding up flash reads

The laRva core runs no much faster than 20MHz, but the SPI flash can run up to 108MHz according to its datasheet. Therefore we can use a 4 times faster clock for the memory (5 times in fact, but power of two dividers are very desirable), 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 $50\mu s$)

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.

4.3 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.

4.4 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)