DIFFERENCE BETWEEN FPGA AND CPLD

FPGA-Field Programmable Gate Array and CPLD-Complex Programmable Logic Device--both are programmable logic devices made by the same companies with different characteristics.

- "A Complex Programmable Logic Device (CPLD) is a Programmable Logic Device with complexity between that of PALs (Programmable Array Logic) and FPGAs, and architectural features of both. The building block of a CPLD is the macro cell, which contains logic implementing disjunctive normal form expressions and more specialized logic operations".

- This is what Wiki defines.....!!
- Click here to see what else wiki has to say about it !

Architecture

- Granularity is the biggest difference between CPLD and FPGA.

- FPGA are "fine-grain" devices. That means that they contain hundreds of (up to 100000) of tiny blocks (called as LUT or CLBs etc) of logic with flip-flops, combinational logic and memories.FPGAs offer much higher complexity, up to 150,000 flip-flops and large number of gates available.
- CPLDs typically have the equivalent of thousands of logic gates, allowing implementation of moderately complicated data processing devices. PALs typically have a few hundred gate equivalents at most, while FPGAs typically range from tens of thousands to several million.
- CPLD are "coarse-grain" devices. They contain relatively few (a few 100's max) large blocks of logic with flip-flops and combinational logic. CPLDs based on AND-OR structure.
- CPLD's have a register with associated logic (AND/OR matrix). CPLD's are mostly implemented in control applications and FPGA's in datapath applications. Because of this course grained architecture, the timing is very fixed in CPLDs.

- FPGA are RAM based. They need to be "downloaded" (configured) at each power-up. CPLD are EEPROM based. They are active at power-up i.e. as long as they've been programmed at least once.
- FPGA needs boot ROM but CPLD does not. In some systems you might not have enough time to boot up FPGA then you need CPLD+FPGA.
- Generally, the CPLD devices are not volatile, because they contain flash or erasable ROM memory in all the cases. The FPGA are volatile in many cases and hence they need a

configuration memory for working. There are some FPGAs now which are nonvolatile. This distinction is rapidly becoming less relevant, as several of the latest FPGA products also offer models with embedded configuration memory.

- The characteristic of non-volatility makes the CPLD the device of choice in modern digital designs to perform 'boot loader' functions before handing over control to other devices not having this capability. A good example is where a CPLD is used to load configuration data for an FPGA from non-volatile memory.

- Because of coarse-grain architecture, one block of logic can hold a big equation and hence CPLD have a faster input-to-output timings than FPGA.

- Click here to read one good article.

Features

- FPGA have special routing resources to implement binary counters,arithmetic functions like adders, comparators and RAM. CPLD don't have special features like this.
- FPGA can contain very large digital designs, while CPLD can contain small designs only.The limited complexity (<500>

- ***Speed***: CPLDs offer a single-chip solution with fast pin-to-pin delays, even for wide input functions. Use CPLDs for small designs, where "instant-on", fast and wide decoding, ultra-low idle power consumption, and design security are important (e.g., in battery-operated equipment).

- ***Security***: In CPLD once programmed, the design can be locked and thus made secure. Since the configuration bitstream must be reloaded every time power is re-applied, design security in FPGA is an issue.

- ***Power***: The high static (idle) power consumption prohibits use of CPLD in battery-operated equipment. FPGA idle power consumption is reasonably low, although it is sharply increasing in the newest families.

- ***Design flexibility***: FPGAs offer more logic flexibility and more sophisticated system features than CPLDs: clock management, on-chip RAM, DSP functions, (multipliers), and even on-chip microprocessors and Multi-Gigabit Transceivers.These benefits and opportunities of dynamic reconfiguration, even in the end-user system, are an important advantage.

- Use FPGAs for larger and more complex designs.

Click here to read what Xilinx has to say about it.

- FPGA is suited for timing circuit becauce they have more registers , but CPLD is suited for control circuit because they have more combinational circuit. At the same time, If you synthesis the same code for FPGA for many times, you will find out that each timing report is different. But it is different in CPLD synthesis, you can get the same result.

As CPLDs and FPGAs become more advanced the differences between the two device types will continue to blur. While this trend may appear to make the two types more difficult to keep apart, the architectural advantage of CPLDs combining low cost, non-volatile configuration, and macro cells with predictable timing characteristics will likely be sufficient to maintain a product differentiation for the foreseeable future.

- There are people who discuss about this. Click here to listen them.
- Finally here is one pdf document whcih is downloadable: "Architecture of FPGAs and CPLDs: A Tutorial" Download

Hoping that information and references helps you ..... comments and further references are welcome !

1 comments Tags: ASIC, FPGA, VLSI

Asynchronous FIFO Design

Asynchronous FIFOs are used as buffers between two asynchronous clock domains to exchange data safely. Data is written into the FIFO from one clock domain and it is read from another clock domain. This requires a memory architecture wherein two ports of memory are available- one is for input (or write or push) operation and another is for output (or read or pop) operation. Generally FIFOs are used where write operation is faster than read operation. However, even with the different speed and access types the average rate of data transfer remains constant. FIFO pointers keep track of number of FIFO memory locations read and written and corresponding control logic circuit prevents FIFO from either under flowing or overflowing. FIFO architectures inherently have a challenge of synchronizing itself with the pointer logic of other clock domain and control the read and write operation of FIFO memory locations safely. A detailed and careful analysis of synchronizer circuit along with pointer logic is required to understand the synchronization of two FIFO pointer logic circuits which is responsible for accessing the FIFO read and write ports independently controlled by different clocks.

Why Synchronization?

It is very important to understand the signal stability in multi clock domains since for a traveling signal the new clock domain appears to be asynchronous. If the signal is not synchronized to new clock, the first storage element of the new clock domain may go to metastable state and the worst case is that resolution time can't be predicted. It can traverse throughout the new clock domain resulting in failure of functionality. To prevent such failures setup time and hold time specification has to be obeyed in the design. Manufacturers provide statistics of probability of failure of flip-flops due to metastability characters in terms of MTBF (Mean Time Before Failure). Synchronizers are used to prevent the downstream logic from entering into the metastable state in multiclock domain with multibit data values.

Issues in Designing Asynchronous FIFO

It has been mentioned that designing of FIFO pointers for efficient working of FIFO is the key issue while designing FIFO architecture. Let us go deep into the FIFO read and write pointers. On reset both read and write pointers are pointing to the starting location of the FIFO. This location is also the first location where data has to be written at the same time this first location happens to be first read location. Therefore, in general we can say, read pointer always points to the word to be read and write pointer always points to the next location to which data has to be written.

Now let us examine data write operation. When both read and write pointers are pointing to first location of FIFO empty flag is asserted indicating the FIFO status as empty. Now data writing can be performed. Data will be written to the location where the write pointer is pointing and after the data write operation write pointer gets incremented pointing to the next location to be written. At the same time, empty flag is deasserted which indicates that FIFO is not empty, some data is available. One notable point regarding read pointer is with empty flag active the data pointed out by the read pointer is always invalid data. When first data written and empty flag status cleared (i.e. empty flag inactive) read pointer logic immediately drives the data from the location to which it was pointing to the read port of the dual port RAM, ready to be read by read logic. With this implementation of read logic the biggest advantage is that only one clock pulse is required to read from read port since previous clock cycle has already incremented read

pointer and drives the data to read port. This will help in reducing latency in detecting empty and full pointer flag status. Empty status flag can be asserted in one more condition. After some n number of data write operations if same n number of read is performed then both pointers are again equal. Hence if both pointers “catch up” each other then empty flag is asserted.

Now let us examine about FIFO full status. When write pointer reaches the top of the FIFO, it is pointing towards the location, which can be written and is the last location to be written. No read operation is performed yet and read pointer is pointing to first location itself. This is one method is to generate FIFO full condition. When write pointer reaches the top of the FIFO, if full flag is asserted then it is not the actual FIFO full condition, this is only ‘almost full’ as there is one location which can be written. Similarly almost empty condition can exist in FIFO. Now a write operation causes the location to be written and increment of write pointer. Since the location was the last one write pointer wraps up to first location. Now both read and write pointers are equal and hence empty flag is asserted instead of full flag assertion, which is a fatal mistake. Hence wrap around condition of a full pointer may be a FIFO full condition.

After writing the data to FIFO (consider write pointer is in top of FIFO) some data has been read and read pointer is somewhere in between FIFO. One more write operation causes the write pointer to wrap. Note that even though write pointer is pointing to first location of FIFO this is NOT FIFO full condition, since read pointer has moved up from the first location. Further data writing pushes write pointer up. Imagine read pointer wraps around after some more read operation. Present condition is that both pointers have wrapped around but there is no FIFO full or FIFO empty condition. Data can be written to FIFO or read from the FIFO. This is being the situation how to identify and generate full and empty condition? How to synchronize and compare these two pointers to generate full and empty status? While synchronizing how to avoid possible metastable state and ‘pessimistic reporting’ (i.e. harmless wrong report; will be discussed later)? These are some key issues in designing an asynchronous FIFO.

Source : http://asic-soc.blogspot.in/2007/11/what-is-difference-between-fpga-and.html