Abstract—This paper describes the FPGA interface from the software point of view. The software department may have own special wishes and requirements which should take account when FPGA implementation is designed.

Index Terms—FPGA, software interface, ASIC

I. INTRODUCTION

Traditionally software and hardware development of the embedded devices has had quite different methods and requirements. Also the common design procedures have been quite rare. Often the hardware development has made the hardware design and even if there has been a software architecture person who has taken care of the main requirements of the software, the software department has gotten quite ready prototype with fixed interfaces without possibility to review all the software interfaces of the device. After that the software department has to live with the device, even if there might be better design solutions which benefit both hardware and software implementation.

Nowadays very strict time schedules of the implementation forces the hardware and software development to make much more co-operation during design time. Also the importance of the software is increasing in embedded systems. Moreover, the availability of the modern hardware design languages like VHDL (VHSIC Hardware Description Language, VHSIC = Very High Speed Integrated Circuit) actually has made the hardware and software implementation more equal than before and maybe the hardware and software persons have more common language than before.

One of the problematic interfaces between hardware and software implementation is the design of the ASIC (Application Specific Integration Circuit) or FPGA (Field Programmable Gate Array). The FPGA- and ASIC -interfaces are quite same in the software point of view. The main difference is normally the performance [1]. Usually the chip makes the hard work of the required functionality and the software gives the parameters how to make this bit crushing. Good examples of this kind of implementation are different audio and video codecs, image and video handling generally and telecommunication stack implementation. The image handling example is described in Figure 1.

![Processor to FPGA connection diagram](image_url)

Fig. 1. An example of FPGA usage

Even if the chip makes the actual work, it has a lot of parameters and functionalities which has to be set by software. If there is a good co-operation between hardware and software designers during the design time of the chip, the hardware and software components are easier to integrate and the result meets the expectations.

This paper defines some of the most problematic areas in the software point of view and gives some suggestions how to make the interface of the FPGA or ASIC more software friendly.

II. FPGA AND PROCESSORS

When FPGA implementation is designed with the processor or MCU (Micro Controller Unit), there are two different basic implementation methods. The FPGA design can contain the MCU part or the MCU and FPGA are separate chips [2]. This paper concentrates to the solution where these logics are separated.

Connection between FPGA and MCU

Obviously, separated MCU and FPGA require some media which transports information between these two. There is several solutions which may make this functionality. I2C (Inter-Integrated Circuit) is maybe the most popular bus selection between FPGA and the MCU. Other possibilities are, for example, SPI (Serial Peripheral Interface) or ever serial type connection.

FPGA inside the memory space of the MCU

The best option in software point of view is solution when the registers of the FPGA are visible straight in the memory space of the MCU. The key word in this solution is easiness, it is easy to use, easy to implement and easy to debug. When all registers can be seen from the memory space, an equal software structure than the register chart can be fitted on top of the FPGA's registers and using this method, the FPGA (or ASIC) can be accessed as normal variables of the programming language. This allows also ready and library based memory operations to use. For example, if software wants to dump the whole content of the FPGA's registers, normal memcpy() (memory copy) function can be used. Actually, quite often software wants to keep copy of the register content, at least when the functionality is debugged and this is extremely important feature when the software and hardware are integrated and debugged at the first time. The C-style implementation of this method can be seen from Figure 2. The example is very simplified and thus, for example, the bit fields of C-language are not used.
### Fig. 2. Register of different groups of bits

The total value of the register is wrong. If the register value is this value, it is very probable that between these operations, use at once. When the software makes two write operations to FPGA, the software has to handle it using two operations. This may be problematic situation, if the value of the register is taken to when register is written. Figure 4 shows this kind of feature FPGA, the software has to handle it using two operations Long register values

<table>
<thead>
<tr>
<th>C-solution</th>
<th>FPGA registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>typedef struct FPGA {</strong></td>
<td>Version, 8 bits</td>
</tr>
<tr>
<td>unsigned char version;</td>
<td>Enable, 8 separate bits</td>
</tr>
<tr>
<td>unsigned char enable;</td>
<td>Mask, 8 separate bits</td>
</tr>
<tr>
<td>unsigned char mask;</td>
<td>Data, 8 bits</td>
</tr>
<tr>
<td>unsigned char data_LSB;</td>
<td>Data LSB, 8 bits</td>
</tr>
<tr>
<td>unsigned char data_MSB;</td>
<td>Data MSB, 8 bits</td>
</tr>
<tr>
<td>} FPGA;</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 4. One value is divided to the several registers**

### III. DIFFERENT REGISTER TYPES

Of course, this kind of memory solution has own challenges. The word length and the big/little endian characteristics have to be carefully considered. **Maybe the most critical part in this solution is the cache memory settings of the MCU.** The memory space, where the FPGA is settled, should not use the cache and this may be difficult to configure, depending on the memory handling possibilities of the processor. At least, the cache disabling reduces the performance of the MCU [3]. If the cache is active in the FPGA part of the memory, the read and write operations may use the cache instead of the FPGA itself. This may cause very odd problems. Even if the cache memory can be forced to write its content to the target memory by using flush-type commands, this is not a bullet-proof solution.

**FPGA via I2C bus**

Of course, the I2C solution is usually used and definitely, it is also working method. The problematic time is normally when the hardware and software are integrated together in the very beginning. Then the designers can not be sure if the I2C working correctly and does it really writes and reads the correct data from the FPGA. Even if the I2C is very standardized bus, it contains quite a lot of hardware and software implementation, and therefore probable errors, before the FPGA is accessible via the bus.

### IV. ASYNCHRONOUS ACTIONS

Depending the designing of the FPGA, the software has to handle the corresponding software variables in different ways. C-language offers bit fields, which are very handy when registers as in Figure 3 are located to the FPGA. Even if the bit fields are not used, the normal bitwise operations handle these situations.

<table>
<thead>
<tr>
<th>B7-B6</th>
<th>B5-B3</th>
<th>B2</th>
<th>B1</th>
<th>B0</th>
</tr>
</thead>
</table>

**Fig. 3. Register of different groups of bits**

**Long register values**

If the value is so big that it does not fit in one register of the FPGA, the software has to handle it using two operations when register is written. Figure 4 shows this kind of feature (M/LSB = Most/Least Significant Bits). This may be problematic situation, if the value of the register is taken to use at once. When the software makes two write operations to this value, it is very probable that between these operations, the total value of the register is wrong. If the register value is not critical, this can be acceptable, otherwise there has to be some activation mechanism which is informed when both values are written.

**Read-only registers**

Flag-type registers defines normally the state of some phenomenon of the FPGA, these can be read-only registers of reset-on-read -type registers. Normally this is not a problem in software point of view as long as these register types are not mixed inside one register.

**Invisible data**

Sometimes data is transferred from the FPGA chip to some third component without MCU actions or visibility. An example of this kind of action is the situation, when image is transferred from the FPGA straight to the device's display using DMA (Direct Memory Access). In this case, MCU has only very limited visibility to the data which is transferred, and the debugging of this feature is extremely difficult by the software. Therefore, **it is recommendable that the software has at least some way to see the data which is normally sent to another component without MCU.**

**Essential register: the version number**

Surprisingly, **one important register, especially in case of ASIC solution, is the version number of the ASIC.** Also the location of the version number is essential. The version number has to be the first register of the ASIC. Sometimes, the first version of the ASIC is not good enough and there might be several versions of ASIC in the field. This may mean that the same software version have to handle different versions of same ASIC and also different register structures of the ASIC. Therefore, software has to check the version of the ASIC before it can be sure what is the register content of the chip. Obviously, the version number has to be in the first register, otherwise software does not know where to search the version number, if the register chart has been changed.
V. INTERRUPTS VERSUS POLLING

The interrupt is very important tool in the embedded system. However, it can be also very problematic one in the software point of view. Of course, part of the interrupt characteristics are defined by the interrupt settings of the MCU, but also the FPGA implementation has own role to define the interrupt features.

Obviously each interrupt has to be able to mask individually. A long burst of interrupts can jam the MCU easily and therefore it is essential that software can mask every interrupt.

**Level sensitive or edge sensitive?**

The interrupt can be parameterized as edge sensitive or level sensitive. **Software prefers edge sensitive** even if it means that any interrupt should not be missed or processed. Otherwise, the software ends in wrong state. Quite often a backup mechanism, a polling loop, is implemented to ensure the robustness of the interrupt handling. Also other reasons can be found, which enforces to use the polling mechanism, at least temporarily. For example, in the early software implementation phase, it is easier to ensure the functionality by polling the status, which is normally informed by an interrupt. Also some error situations, like long continued interrupt burst may require to mask the interrupt and scope the state of some phenomenon using a polling loop. Therefore, it is important that there is also possibility to poll the values which cause interrupts.

VI. CONCLUSIONS

Two different aspects were noticed when this paper was written. First of all, this paper gives only the software point of view to the topic and actually this perspective is only the half of the truth or maybe even less than half. Even if the author knows that every suggestion of this paper is possible (because they have implemented in the real life), there is no knowledge, how much those proposals pays when a FPGA or ASIC is designed or is there better designing methods which fulfills the requirements of software and hardware. Therefore, there should definitely be two authors in this kind of paper, a software designer and hardware designer. This allows true dialog between hardware and software designing and probably ends up to the best result.

Another issue, which was noticed, was the shortage of the literature of the co-design between the software and the hardware in the embedded system. Especially, in case of FPGA or ASIC design, it was very difficult to find any guidelines which are considering also the software requirements. No doubt, there has to be papers or books about this area, but it seems, not so many.

**What is missing?**

One area, which was not discussed in this paper, was the software requirements of the FPGA programming itself. Indeed, the binary code of the FPGA has to be loaded to the chip, when the device is booted up. This is done by the software and it also has own requirements and best practices which should be taken account. However, the programming structure is very strictly defined by the FPGA manufacturer and there is not much what can be changed. However, the Christmas is coming and software guys are always kind – so we can at least wish :)