**Journal**

**Exercise 2 HW/SW Co-design**

Date: **07/12-10**

Authors: **Brian Vestergaard Danielsen,**

**Teddy Holm Roskvist,**

**Anders Hvidgaard Poder**

# Introduction

In this journal contains our findings and conclusions for Exercise 2. The first part of the Journal will contain the answers to the questions posed in the assignment, and the appendix contains the detailed rationales and minutes of meetings that lead us to these answers.

# Assignment 2.1

During our first meeting we arranged to come up with our individual suggestions and then agree on a method.

The individual suggestions may be found in the appendix.

The conclusion was that we would use SysML as much as possible, and amend with custom timing diagrams if needed. We did however decide to not use the requirements diagram, but instead simply use a table for the non-functional requirements. We also decided to try and map our diagrams to the Y-chart, in order to maintain a good link to the HW/SW co-design methodology.

For details about the individual diagrams please refer to the appendix, or simply see them in the following parts of the journal.

# Assignment 2.2

We decided to do assignment 2.2 and 2.3 in plenum during our first design session, based on prepared input from the participants with respect to use cases. The final Use-case diagram came to be as follow:



It may be seen that we have included a HW/SW separation, and though this is not officially a part of the use-case diagram, we would like to illustrate that we already here started to notice an “obvious” separation into HW and SW components.

The rational for this separation may be found in detail in the appendix, but will also be discussed in assignment 2.3, but it is a matter of how likely the functionality is to change, how math-intensive it is, and what its performance requirements is. As for performance we talked about the non-functional requirements, and decided on the following (not real requirements, but more a talk about what is needed to do the processing):

|  |  |  |
| --- | --- | --- |
| ReqID | Related UseCase(s) | Description |
| 1 | Noise cancellation | We must process two 24bit samples in 1/48000 of a second, or 20.8μs.  NB: Remember if 16bit architecture \* 2 for MUL |
| 2 | Audio/Video Processing | We must maintain at least 25 VGA frames a second.  NB: At 640x480 that is 25\*640\*480 = 7.68Mpixel/s, or 130ns per pixel. |
| 3 | Audio/Video Processing | We must be able to filter the audio signal with respect to bass, treble and volume within the given time constraints (20.8μs). |
| 4 | Audio/Video Processing | We must be able to mux the audio and video in an mpeg4 stream within the given time constraints. |
| 5 | Audio/Video Processing | We must be able to place the muxed data on Firewire with the given speed.  NB: Firewire has more than enough bandwidth, but its protocol must be met (Hard real-time). |
| 6 | Audio Control | We must update the filter coefficients for bass, treble or volume within 500ms of altering the dials on the controls. |
| 7 | Remote control | We must update the Audio control filter coefficients within 1 second of receiving them form the remote computer. |
| 8 | Display Status | We must display status changes within 500ms of a given status change occurring. |
| 9 | Firmware update | We must be able to complete a Firmware update within 30 minutes. |

Finally there are some design constraints dictated by the assignment, like the use of microphones, VGA, S-video, … These should naturally be described in the design constraints part of the requirements specification, but we have left them out for simplicity.

This is clearly not a complete requirement specification, just like the detailed use-case descriptions have been left out. This is done on purpose to focus on the architectural design and not the requirements.

Furthermore the assignment calls for analyzing the functionality with diagrams, but this we would like to postpone to assignment 2.3, where the architecture and design will describe the functionality and design.

# Assignment 2.3

There are many ways to document a system like this, and many different diagrams one may choose, but to keep the journal manageable we have decided to use the block diagram type (basic and internal) to describe the composition and communication flow of some of the important components.

Firstly we look at the static structure of the overall system with a basic block diagram



As it may be seen no decision has been made as to what is implemented in HW or SW, except for the parts that is a physical unit, e.g. the physical microphone, which has been moved from actor to HW block.

Looking at the internal block diagram we start adding more detail, and yet we still do not have to decide on HW or SW implementation, but the added details may aid us in our decision.



This diagram details the inner workings of the Noise cancellation on the block level (no selection of algorithm or mapping yet), but the different required parts may be seen, as may the input and output and speed of same. We can see that the flow into the noise cancellation and out of the noise cancellation is quite intense (data flow oriented), with the exception of the Firmware update which is sporadic (asynchronous means no lower limit between events) and do not have a “short” time requirement. The received audio signals must be read in and cross-correlated (or other filtering technique like LMS ☺) to remove the noise and written to the output before the next sample is ready.



This diagram shows the Audio/Video processing, and here the data-flow structure and high speed input and output becomes even more pronounced, with the exception of the Audio control. Here we must read and filter not only the audio signal from the noise cancellation block, but also read and process video and combine the audio and video into an Audio/Video stream, and all before the next sample is ready for processing.



This diagram shows the Remote control block, and unlike the other blocks, this has no hard real-time requirements. Granted the serial bus has a data flow, but not only a rather slow one, but also one where the data comes sporadic and may therefore be buffered, and where (if we use RTS/CTS flow control) we even have the ability to request a pause in transmission if our buffer is full, and all without losing data or sacrificing any deadline. Furthermore, though it may not be seen here, the Command interpreter is pure control logic.

Instead of doing the remaining diagrams we have decided to focus on the HW/SW separation. Naturally after the HW/SW separation many more diagrams should be drawn, like state diagrams for the Remote Control would be an obvious choice, and timing diagrams for the delay/filter and mixing in Audio Control.

## Separation of HW and SW

There is much information what must be considered when deciding on whether functionality should be mapped to HW or SW:

* Our current platform has no HW support, live with it.
* We have no HW development resources available.
* The given functionality cannot be implemented in SW and meet its deadline.
* The lower unit cost cannot outweigh the added cost and complexity of developing this in HW (neither ASIC or FPGA).
* We already have HW RTL components available for this functionality, so HW implementation is not a problem.

The only direct restriction that we are facing is whether a SW mapping is fast enough, which we may determine by analyzing the number of instructions required to process one sample and the speed at which the samples are arriving – that way we can determine the required size of a CPU or HW (not the same, as required instructions in SW is not the same as the required instructions in HW).

We have decided to do a simple table based approach and look at the different blocks and determine how suited they are for HW mapping, based on the following criteria:

* Performance requirements
  + If there is a high throughput performance requirement then HW is a good alternative to SW, and might also be the only possibility.
* Risk of change
  + Changing HW is much more complicated then changing SW, so if there is a high probability of change to the block then a HW implementation should be avoided.
* Data flow vs. Control logic
  + HW is not very well suited to implement control logic, but is much more suited for simple filtering and other basic block algorithms. A good way to determine this in practice is to look at the state diagram and the breakdown into Basic Blocks. If there are many states and Basic Blocks then it is not very suited for HW-mapping.
* Availability
  + Using COTS to minimize risk and development time and cost is a vital tool in a designer’s tool-box. If a suitable existing tried and proven IP or SW library exists then it is almost always preferable to developing it yourself.

Based on these criteria we have come up with two suggestions for an architectural mapping. First of all we look at the tings that are “no brainers”.

The high throughput, standardised protocol and well tested existing solutions makes the design of the SVideo -> VGA and RAW -> MPEG4 -> Firewire simple to map to HW and the performance requirements actually makes it difficult to map it to SW. Actually the Firewire you usually purchase as an ASIC, but technically it could also be an IP and run on the same HW as the other transformers. The blocks that are already designated as HW (e.g. the ADC and DAC) will not be mentioned, as they are already decided. These components are standardised, meaning very low risk of change. They have a high requirement for throughput, they are heavy data-flow oriented, and they are readily available to purchase as HW (either IP or ASIC).

In the other end of the spectrum is the Remote Control, Firmware update, Display Status and Audio Control. Here the real-time requirements are practically non-existing. A serial protocol for commanding the system from a PC and uploading new firmware running on a serial bus (RS232) at 115200 baud does not set high performance requirements. How long it takes to upload new firmware is almost irrelevant, as it would never be done while the system is “active”. As for audio control it is irrelevant if it takes 100, 200 or 500ms to update the audio – at most a very soft real-time requirement, easily met in SW. And this holds for both commands received from the Remote PC or the buttons, which can easily be polled or hooked up to an interrupt. Furthermore these blocks have quite a bit of control logic – even the audio control has to distinguish between audio set from the remote PC and via the buttons. It cannot simply sample the buttons and set the audio coefficients accordingly, as that would override any audio settings from the remote PC. The system must have some kind of control logic to distinguish when a given input is master. Then there is the risk of change. A protocol to the remote PC and what should be written to the display status LCD is both proprietary protocols and therefore often subject to change from user responses and also not available as COTS. All of this points to an easy SW implementation.

This leaves two blocks or sub-blocks; Noise cancellation and the part of the Audio/Video processing that has to do with bass, treble and volume filtering. The noise cancellation algorithm can either be purchased as an IP, making it a no brainer as a HW implementation, but assuming the COTS noise cancellation algorithms are not sufficient then we have to implement it our selves. Naturally a SW solution is faster to implement, but filters like this generally have a high throughput requirement, making them difficult to implement in SW. This point towards making a SW solution for testing and then porting it to HW when it is good enough. Unfortunately the algorithm might have to be changed after deployment, which is not possible if we assume an ASIC or a single FPGA running everything without the ability to do partial burn. This risk of change points to a SW solution, but is it possible? Let us have a look at some figures:

Looking at “Understanding active noise cancellation” by Colin H. Hansen we see that the common noise cancellation algorithms are very clock cycle intensive in the magnitude of many million clock-cycles for processing a single sample through a 100 coefficient filter. Looking at “A FPGA-BASED ADAPTIVE NOISE CANCELLING SYSTEM” by Wolfgang Fohl and Jörn Matthies, we can see that realizing this algorithm in a 400k gates FPGA loads the FPGA about 25% when run at 48kHz. This may quickly be translated to a very serious CPU if it was to be implemented in SW. Therefore it maybe worth running the risk of change and implement it in HW, simply because the CPU requirements, if at all possible, would be so high.

The bass, treble and volume filter may most likely be implemented in SW, but if the filter does not have to be exceptionally precise then good COTS IPs exist, making it easy to implement in HW. On the other hand if it has to be more accurate than existing IPS, then it most likely requires a high order filter, meaning that the SW throughput may be a problem. This again point towards a HW implementation.

Combining this gives us the table below, which contains our suggestion for an architectural mapping.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **HW non-changeable gate arrays** | **HW changeable gate arrays** | **HW/SW boundary** | **SW** |
| **Noise cancellation** | Noise cancellation algorithm | Coefficients registries |  |  |
| **Audio/Video processing** | Volume, bass, treble algorithm | Volume, bass, treble coefficients registers. |  |  |
| Audio/Video MUX -> Firewire (IEEE1394) |  |  |  |
| S-Video -> VGA |  |  |  |
| Audio -> speakers |  |  |  |
| **Remote Control** |  |  |  | Protocol for controlling audio levels. |
|  |  |  | Protocol for controlling and transmitting firmware update. |
|  |  | Firewire Audio/Video streaming – not really a boundary. |  |
| **Audio control** |  |  | Update volume, bass, treble Coefficients registries | SW interface for updating bass, treble, volume coefficients. |
|  |  |  | Poll bass, treble and volume dials. |
| **Display Status** |  |  |  | Update status on display according to input from Audio control and Remote control. |
| **Firmware update** |  |  | Update noise cancellation coefficients registry. | SW interface for updating coefficients. |

## Mapping to Y-chart

The above architecture concentrates on how the static mapping of the architecture should be, and not on how we get from the Customer document of intent to the finished product. The Y-chart has a suggestion for a process which may be used, at least for the last part of the process. The first part, which is the first level refinement of the requirements are somewhat outside of the Y-chart domain. The Y-chart starts when a system requirements specification has been defined, at least at the overall level, meaning that the Customer intent document has been refined to a functional requirements (use cases) divided into processes and a table of non-functional requirements.

From here you can either use Top-down, bottom-up or meet-in-the-middle methodology. Our approach has been Top-down, as we have designed the architecture from the requirements specification and then broken it down to smaller and smaller parts until it is realizable. In the Y-chart this follows a specific structure at four different levels.

|  |  |
| --- | --- |
| System level | 1. Take the functional processes (Use cases) and divide them on different processing elements (PE) connected by communication elements (CE). This may be done in SysML, as may be seen above. 2. Define the HW/SW separation and implement an executable (untimed) model of the system including PEs and CEs. This constitutes the TLM and can e.g. be done in SystemC. 3. Possibly add timing to the PEs and/or CEs, obtaining a timed TLM. This may be used to determine what should be done in HW and SW based on the clock requirements. The Timed TLM may also be done in SystemC. |
| Processor level | 1. Model the SW PEs/CEs on an ISS (Instruction Set Simulator) and the HW PEs/CEs as connected RTL components. 2. This new model may now be simulated at the CAM level – this is no longer SystemC, but perhaps ModelSim. |
| Logic level | The CAM model is now refined to include the actual logic layout on the chip. |
| Circuit level | The logic layout is refined to an actual HW, specific FPGA or ASIC. With this level you can burn an FPGA or have a chip manufacturer create an ASIC for you. |

The last two levels are for FPGAs always done automatically, and for ASICs most of it is done automatically, with perhaps a little manual tweaking.

The Y-chart further divides the process into two stages for each level, yet for a top-down approach only the first stage is executed for all level but the last one. The reason for this is that the second stage is applying the structural model (output of the above table) to an actual circuit, and this is only possible for bottom-up (and some of meet-in-the-middle). Therefore this division is not very important for our approach, as we do not continue all the way to the circuit level.

# Assignment 2.4

We focussed on getting a good design and an architecture with a well thought through mapping. We believe that determining the correct mapping is more important than doing the actual realization from a learning point of view. We have had this confirmed by Kim Bjerge, who informed us that we were allowed to not complete assignment 2.4, as long as we had an understanding of how the realization could be done. We believe we have covered that in the previous sections, and will therefore not include a Quartus SoPC project with the implementation for the DE2 board.

# Conclusion