TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

IMPROVING IP BLOCK DESIGN FLOW PRACTICES

LAURI JUHOLA

Master of Science thesis

Examiner: Prof. Timo D. Hämäläinen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 4th April 2016

ABSTRACT

LAURI JUHOLA: Improving IP block design flow practices
Tampere University of Technology
Master of Science thesis, 69 pages
May 2016
Master's Degree Programme in Electrical Engineering
Major: Embedded Systems
Examiner: Prof. Timo D. Hämäläinen
Keywords: IP design, hardware design, System-on-Chip (SoC) design

Digital systems are really complex and, thus, their design is not straightforward either. Big part of the actual design work is done by designing IP (Intellectual Property) components, re-using them and integrating them. This thesis focuses on IP development flow at Nokia Solutions & Networks (NSN). The goal is to identify working habit related issues and their importance, and provide some easily implementable actions to improve the situation.

This thesis includes a literature study of the common flow issues and solutions. The issues could be categorized in the following way: inefficient communication, finding correct documents/files, unclear documentation, unclear code, scripting, version control, software tool related problems including computational platforms and IT support. The field of IP design is not studied widely so software development was used as a reference when applicable. After the most common issues were identified, 7 employees of NSN were interviewed to get more specific information and find out the impact of different issues. The issues with most impact were: IT related issues, unclear documentation, finding correct documents and tool related problems.

As a conclusion, action points were suggested to improve the situation with most important issues. Since there were already ongoing actions to improve IT and unclear documentation, the point of focus became finding correct documents and tool related problems. Based on the more specified results from the interviews the following actions were suggested: link library with short descriptions for different kinds of documents and locations (especially for not-design-related documents such as tool instructions and guidelines) and more comprehensive tool training and instructions. Additionally, the need for script training and guidelines came up in most of the interviews, so that is also recommended.

TIIVISTELMÄ

LAURI JUHOLA: IP-lohkon suunnitteluvuon käytäntöjen parantaminen
Tampereen teknillinen yliopisto
Diplomityö, 69 sivua
Toukokuu 2016
Sähkötekniikan koulutusohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: Prof. Timo D. Hämäläinen
Avainsanat: IP-suunnittelu, System-on-Chip-suunnittelu

Digitaaliset järjestelmät ovat erittäin monimutkaisia, kuten on myös niiden suunnittelu, joka perustuu suurelta osin IP-lohkojen (Intellectual Property) suunnitteluun, uudelleenkäyttöön ja integrointiin. Tässä diplomityössä keskitytään Nokia Solutions & Networks:in (NSN) IP-kehitykseen. Tavoitteena on selvittää työtapoihin liittyviä ongelmia, niiden keskinäinen tärkeysjärjestys ja antaa helposti toteutettavia parannusehdotuksia.

Työ koostuu kirjallisuusselvityksestä liittyen yleisiin työtapaongelmiin ja niiden ratkaisuihin. Yleisempiä ongelmia olivat tehoton kommunikaatio, oikeiden dokumenttien/tiedostojen löytäminen, epäselvä dokumentaatio, epäselvä lähdekoodi, skriptaus, versionhallinta sekä työkaluohjelmistot mukaanlukien laskentapalvelimet ja IT-tuki. IP-kehityksen tuottavuutta ei ole tutkittu kovin laajasti, joten vertailukohtana käytettiin ohjelmistokehitystä. Kun yleisimmät ongelmat oli selvitetty, 7 NSN:n työntekijää haastateltiin tarkempien tietojen saamiseksi sekä eri ongelmien vaikuttavuuden arvioimiseksi. Eniten vaikutusta koettiin olevan seuraavilla epäkohdilla: IT-tuki ja laskentapalvelimet, epäselvä dokumentaatio, oikeiden dokumenttien löytyminen ja kehitysohjelmistoihin liittyvät ongelmat.

Lopuksi ehdotettiin käytännön toimintoja merkittävimpiin ongelmiin tilanteen parantamiseksi. IT-tukeen ja laskentapalvelimiin sekä epäselvään dokumentointiin oli jo kiinnitetty huomiota, joten ne jätettiin pois. Seuraavia toimia ehdotettiin tehtäväksi: linkkikirjasto eri dokumentteihin ja esim. ohjelmistokohtaisiin linkkikirjastoihin sekä enemmän koulutusta ja ohjeistusta työkalujen käyttöön. Lisäksi tarve skriptikoulutukselle ja -ohjeistukselle tuli esille useimmissa haastatteluissa, joten myös sitä suositellaan.

PREFACE

I would like to use this opportunity to thank my colleagues and especially those who agreed to be interviewed. The open and accepting atmosphere made it easy to locate the issues and think about possible solutions.

A special thank goes to my supervisors Dr.Tech. Erno Salminen and Prof. Timo D. Hämäläinen: I don't believe that all the Master's thesis writers have the privilege to work with so experienced supervisors. I hope that I have managed to transfer even a fraction of your experience on to the pages of this thesis. Not to forget that without you two this wouldn't have been possible in the first place. My line manager Vesa E. Lahtinen helped also a lot with choosing the topic and handling the company bureaucracy connected to the Thesis.

Additionally, I would like to thank my family and friends, for not only encouraging support during the stressful parts but also for insightful discussions of productivity, lean and other related topics. In this category the special thank goes to my girlfriend Petra, who, in addition to all the discussions and input she gave, also had to live with me while I was writing my thesis.

Tampere, 18.5.2016

Lauri Juhola

LIST OF ABBREVIATIONS AND SYMBOLS

AD/DA	Analog-to-digital/digital-to-analog
ASIC	Application specific integrated circuit
DDR	Double data rate
FIR	Finite impulse response
FP	Function point
FPGA	Field programmable gate array
HLS	High-level synthesis
IC	Integrated circuit
IP	Intellectual property
LOC	Line of code
MAC	Media access control
NSN	Nokia Solutions and Networks
SoC	System-on-chip
RTL	Register transfer level
UART	Universal asynchronous receiver/transmitter

TABLE OF CONTENTS

1. INTRODUCTION

Modern integrated circuits (IC) are very complex. As an example, System-on-Chip (SoC) used in XBOX One game console contains 8 processor cores and 5 billion transistors [15]. Figure 1.1 shows a typical case of a SoC chip on a circuit board. The design teams consist of tens to hundreds of engineers and design times are measured in months or years. Therefore, SoC design is often based on re-usable building blocks called intellectual property (IP) components or blocks. Such IP components speed up the design, reduce the risks, make schedules more predictable, and ease division of work [66]. However, designing and using them is not always as straightforward as envisioned in literature. This thesis seeks to identify the issues that cause time waste and frustration in the IP design process at Nokia Solutions and Networks (NSN) and provide low effort solutions for them.

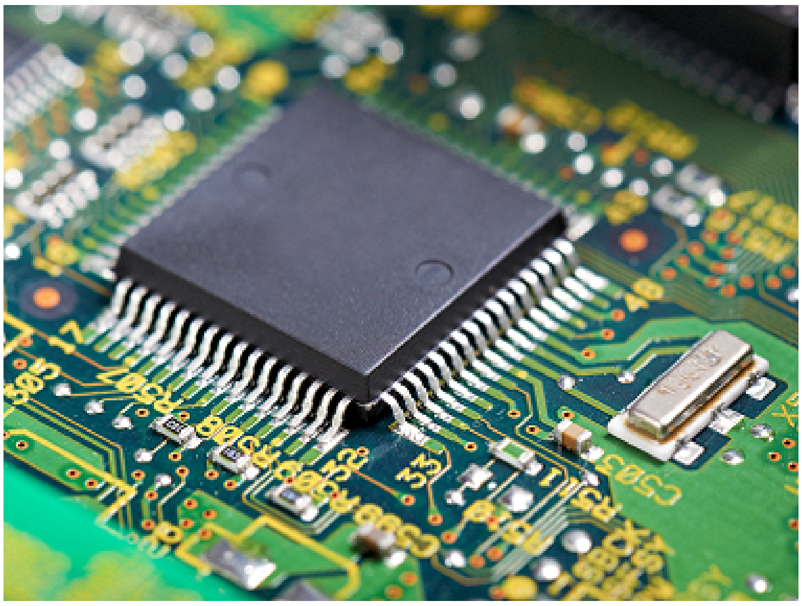


Figure 1.1 *Illustration of a typical SoC on a circuitboard. [27]*

For example, finding the right files, reading unclear code or trying to fix broken scripts can cause many issues. Although some issues sound very trivial, they seem to be difficult to solve. Additionally, these difficulties cause a lot of time waste and designer frustration in practice.

The research for this thesis is conducted in three phases. Firstly, a literature study is conducted to find out commonly known issues and solutions for them. After that, IP designers are interviewed to map which of the issues concern NSN and what is their impact on work. Based on these sources a list of issues with greatest impacts is gathered. After that, it is figured out how issues can be affected with reasonable effort and some realistic action points are suggested. Organization level process or hierarchy changes are out of the scope of this thesis.

This thesis is structured as follows. Chapter 2 introduces the basic digital design concepts and Chapter 3 provides more details about IP components, their re-use and other related topics. Chapter 5 outlines the common issues based on in literature observations. Chapter 6 seeks solutions for revealed common issues. After that, Chapter 7 introduces the outcomes of interviews and suggested action points. The thesis is concluded by Chapter 8.

2. INTRODUCTION TO DIGITAL SYSTEMS

Modern digital systems are complex, widespread, and usually well hidden from the end user. Thus, the presence of digital systems is not always obvious; they can be found from space rockets and cars as well as washing machines and toys. This chapter seeks to explain the basic terms and concepts of digital systems. The focus of this thesis lies on one specific type of digital system design unit, intellectual property (IP) and that concept is explained in more detail in Chapter3.

2.1 Integrated circuit, IC

In the heart of a modern digital system lies one or more integrated circuits (IC). They are complex electronic systems where the components are manufactured on a single silicon chip. This means that notably less discrete components are needed and, thus, this technology enables having a great number of components on a very small area. This makes it possible to manufacture very complex systems, for example, microprocessors. Figure 2.1 illustrates how IC chips physically appear. The picture on the left hand side shows how a chip is packaged to be attached to the circuit board and to the rest of the system. On the right hand side one can see a chip inside a package, namely the actual silicon die and the fine bonding wires that connect its I/O pins to pads on the package. There are also other types of packages, however, generally ICs are inside the black rectangles on the circuit board.

Figure 2.2 shows two circuit boards from mobile phones. The upper one is from an old model and the one below from a modern version. The functionality is almost equal because the newer is targeted to markets in developing countries. However, the amount of discrete components is significantly smaller in the modern version. This is due to the higher level of integration. Integration means that more and more differing functional blocks are manufactured on the same chip thus reducing the amount of chips and external components needed. Additionally, tighter integration often allows higher frequencies and

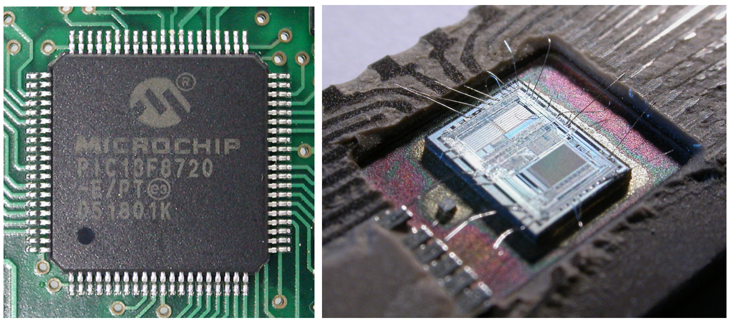


Figure 2.1 *On the left, there is an IC in a package that can be soldered onto a circuit board. [43] On the right, one can see how the IC approximately looks like when the plastic cover of the package is removed. [46] The silicon chip and the pattern on its surface are clearly visible.*

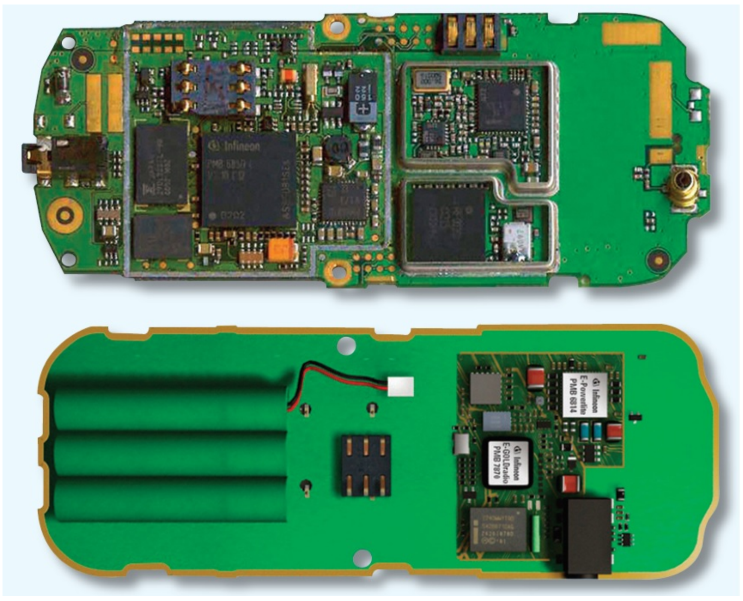


Figure 2.2 *The upper circuit board is from an old phone model and under it is a modern version with similar functionality. [9] The comparison of the two phones clearly demonstrated how integration has led to a greatly smaller amount of components.*

reduces power consumption. Since the price goes down with increased level of integration, highly integrated ICs are the trend in modern day semiconductor design [9].

2.2 System-on-chip, SoC

ICs that contain multiple functionally different digital blocks are named system-on-chip (SoC). A good example of an SoC is a microcontroller that has not only the processor but

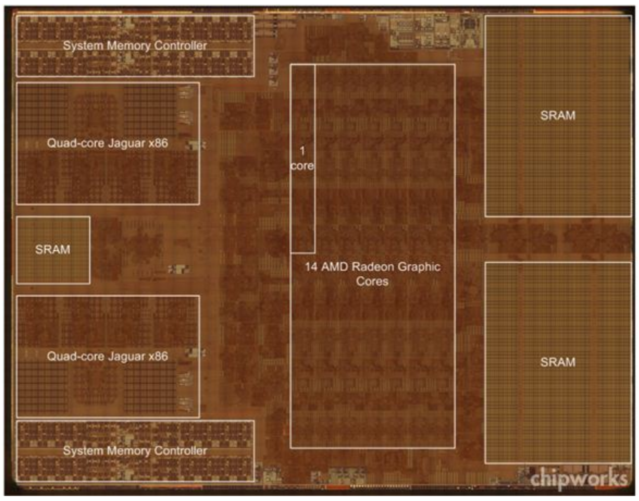


Figure 2.3 *Example of an SoC from XBOX One that has different functional blocks on a single silicon chip. [30]*

also multiple peripherals, as UART block or AD/DA converters, on the same chip. It is also possible that SoC has multiple processors, see for example [67]. Figure 2.3 shows the physical layout of an SoC used in XBOX One. In this case different functional blocks can be clearly seen as separate areas on the chip. Also the complexity of the system is visible.

2.3 Implementing digital systems

One more aspect to consider about digital systems and SoCs is their final implementation technique. Currently, all the systems are more or less ICs: logic circuits manufactured on a silicon chip. However, the division goes further and there are roughly two categories: processors and additional logic. A good example could be an Atmega microcontroller chip: on the same chip there is a processor and some peripherals [7]. The peripherals can be thought to be additional logic that doesn't process any information but takes care of some kind of more simple routine-like tasks, such as interface protocols or AD/DA conversions.

The categorization of processors is a somewhat complicated topic and, in addition, not crucial for understanding the topics of this thesis. Thus, it is not addressed further here. The division of external logic is also not straightforward. However, there are two commonly used types for implementing logic: application specific IC (ASIC) and field programmable gate array (FPGA). ASIC logic is hardwired on the silicon and the structure cannot be changed after manufacturing, whereas FPGA logic is configurable.

2.4 Levels of digital system abstraction

Due to the complexity of modern digital systems, many different levels of abstraction are used during the design. The levels can be roughly divided into the following 4 gategories:

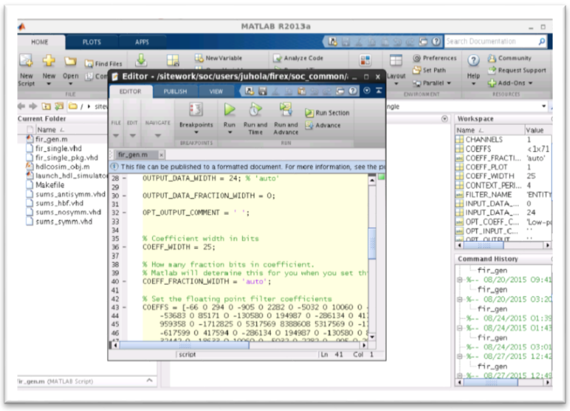
1. Algorithm level
2. Register transfer level (RTL)
3. Gate level
4. Transistor level

Figure 2.4 clarifies the division. The algorithm level describes the functions the design should execute. It does not dictate in detail how the design is implemented in the logical or physical level and usually does not consider the timing. Algorithm model usually also provides a golden reference for design verification. Register transfer level (RTL) describes the logic functionality of the bit level with clock cycle accuracy. It is a mixture of logic description and state machines. However, the RTL level design does not consider the physical implementation, hence it is mostly technology independent. Sometimes, however, the final implementation technology has to be considered already in RTL code, for example when the code instantiates technology-specific memories or interfaces.

The gate level describes the actual logic gates used to implement the logic and, consequently, it is technology dependent. Transistor level shows how the gates are actually implemented with transistors on silicon. If the design is implemented with FPGA technology, the gate level is mapped to the programmable logic elements of FPGA instead of transistors.

There is a trend in digital system design to strive for higher level of abstraction in the design process. Then, the creation of lower level models could be automatized with electronic design automation (EDA) tools. The design process automatization is discussed further in Section 2.6.

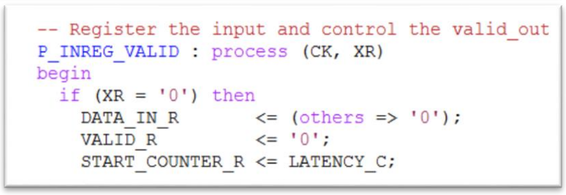
1.



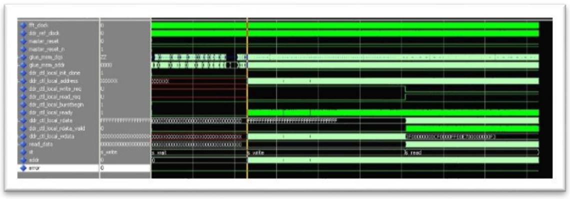
```
OUTPUT_DATA_WIDTH = 24; % 'auto'

OUTPUT_DATA_FRACTION_WIDTH = 0;

OPT_OUTPUT_COMMENT = ' ';


% Coefficient width in bits
COEFF_WIDTH = 25;

% How many fraction bits in coefficient.
% Matlab will determine this for you when you set th
COEFF_FRACTION_WIDTH = 'auto';

% Set the floating point filter coefficients
COEFFS = [-66 0 294 0 -905 0 2282 0 -5032 0 10060 0
    -53683 0 85171 0 -130580 0 194987 0 -286134 0 41
    959358 0 -1712825 0 5317569 8388608 5317569 0 -1
    -617599 0 417594 0 -286134 0 194987 0 -130580 0
```

Algorithm level:
C/C++C/Matlab is often used for algorithm level design.

2.

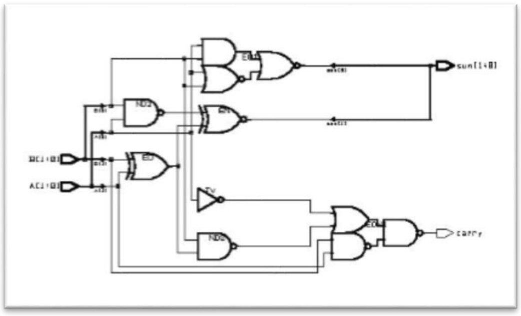


```
-- Register the input and control the valid_out
P_INREG_VALID : process (CK, XR)
begin
  if (XR = '0') then
    DATA_IN_R         <= (others => '0');
    VALID_R           <= '0';
    START_COUNTER_R <= LATENCY_C;
```



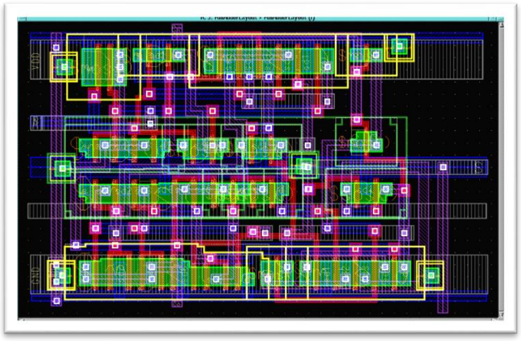
RTL level:
The logic of the design is described with some hardware description language, for example VHDL or Verilog. Usually RTL design is simulated extensively, since RTL level is clock cycle and bit accurate.

3.



Gate level:
The RTL design is mapped to gate level depending on the final implementation technology.

4.



Transistor level:
Finally, the physical layout of transistors is derived from gate level.

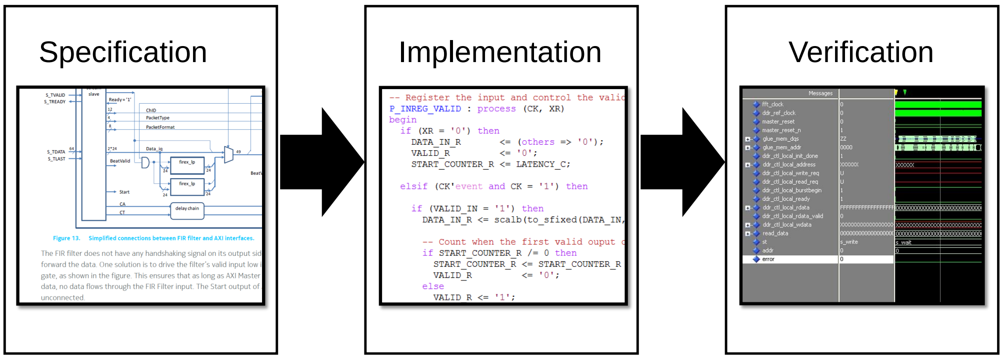
Figure 2.4 *Different levels of digital design abstraction. [44, 68, 36]*

2.5 Digital system design flow

The basic design flow of a digital design can be roughly divided into 3 steps:

1. Specification
2. Implementation
3. Verification

Specification means deciding what is the function of the respective design entity and how it is connected to the rest of the design. Implementation means the actual coding part of the design when the design is described in RTL writing by hand or by instantiating IP blocks. Verification ensures that the implementation works according to the specification. The design steps are not distinct but more or less overlapping. For example, the specification might change during the design process and both implementation and verification must adopt to the change. Equally, some parts can be verified already before the whole implementation is finished.



```
-- Register the input and control the valid
P_INREG_VALID : process (CK, XR)
begin
  if (XR = '0') then
    DATA_IN_R       <= (others => '0');
    VALID_R         <= '0';
    START_COUNTER_R <= LATENCY_C;

  elsif (CK'event and CK = '1') then

    if (VALID_IN = '1') then
      DATA_IN_R <= scalb(to_sfixed(DATA_IN,

      -- Count when the first valid ouput c
      if START_COUNTER_R /= 0 then
        START_COUNTER_R <= START_COUNTER_R
        VALID_R         <= '0';
      else
        VALID_R <= '1';
```

Figure 2.5 *The design flow of a digital system. It is important to remember that the steps are somewhat overlapping. [44]*

Usually, the design is not done by only one person but within teams. There are many ways to divide the work. However, considering the expertise each step requires, engineers ofter have their own designated areas. This all means that a lot of information has to be communicated during design handovers. Additionally, there is a current trend of dividing work between different engineers in a more flexible manner: for example, a person implementing the design is also supposed to do simple preliminary verification. This increases the need for effective communication since now it is not only information about the design that has to be transferred. Also information about the tools and other not design specific thing has to be communicated.

2.6 Design automation and high level synthesis (HLS)

The studies [14, 42] have shown that higher level of abstraction in programming languages results in higher designer efficiency. This is also true in hardware design: higher level models can be created more efficiently than lower level models. For example, the change from gate level design to RTL level design brings great efficiency improvements. [14, 66] In addition, some of the design work in the lowest levels is quite repetitive and tedious to do by hand. Thus, it could and should be automated. Higher level models are also often easier for human designer to understand and, thus, to create and modify. These are the reasons for developing electronic design automation (EDA) tools that automatize the transition to lower level model. However, the gap between the original design and the final physical implementation increases with an increasing level of abstraction [11]. Thus, more elaborate and complicated EDA tools are required to derive the lower level of design and, still, there is a need for an engineer to do some fine tuning.

Currently RTL is the de-facto industry standard for hardware design [66]. However, RTL design still requires people who know some RTL logic description languages, such as VHDL or Verilog. In addition, logic description languages are difficult to write due to the exactness, that could be partially automated. These are the reasons for developments in the field of high level synthesis (HLS): EDA tools that automatically create RTL level code from higher level programming languages such as C or Matlab code.

The biggest issues with high level synthesis are the incompatibility with current design flows and sometimes insufficient traceability. The design flow can be changed, despite the fact that the change requires a lot of effort. Traceability, on the other hand, might be more difficult to improve. Traceability means the ease of tracing the effects of code changes in the final design. It is especially important when the design is being optimized: one has to be able to know which changes are effective and which not. Of course, the effects can be seen after running the flow to the end. However, that requires time and computing power which cost money. Thus, HLS is currently only partially competitive with the traditional RTL level.

It is also possible that the designs created with higher level of abstraction loose in performance to those described originally at RTL level. Additionally, the coding style of, for example C, might affect the final results. This means, that even though a higher level language can be used, some special training and considerations are still needed.

All in all, it seems that HLS is the current trend in digital design. Many different tools are being develop: Catapult-C, Vivado, Synphony C Compiler, Simulink, just to mention a few. [62] The development has been fast in recent years and it is ongoing. Time will tell how big a part of RTL level design can be replaced by high level synthesis and how big tradeoffs are included.

3. INTELLECTUAL PROPERTY (IP) COMPONENTS

This chapter discusses the essence of intellectual property in more detail. IP is first discussed on an introductory level. Then, the concept of re-use is introduced since it plays an important role in the modern day IP design. In addition, re-use also changes quite significantly the design requirements and, consequently, affects the design itself. The concluding section covers one more important aspect of IPs: information packaging with IP-XACT.

3.1 The concept of IP

IP can be understood as a design entity or a basic functional building block of a digital system. IP can be, for example, a processor, memory controller, interface logic, and accelerator functions, see [3, 39] for examples. It is also possible that an IP contains smaller sub-blocks that are IPs themselves. These similar blocks can be used in many designs, thus, time consuming and error prone constant re-design is not necessarily needed.

A good analogy might be Lego bricks [66]. They differ in style and size depending on where they are supposed to be used and one block can be used in multiple places. The blocks have a clear, often standardized interface to other blocks, such as AMBA AXI [6] (or the lumps in Lego bricks). Figure 3.1 shows an artistic conceptual picture of an SoC containing many different IPs.

3.2 Re-use

Design re-use means that the the same design unit is used in multiple designs. The re-use is supposed to save design time since the units are designed and verified/tested only once. In an ideal case the units would not need to be modified at all but just more or less be connected to other re-used units. Naturally, this is not always the case. This section discusses the different aspects of design re-use. First the discussion is on a general level

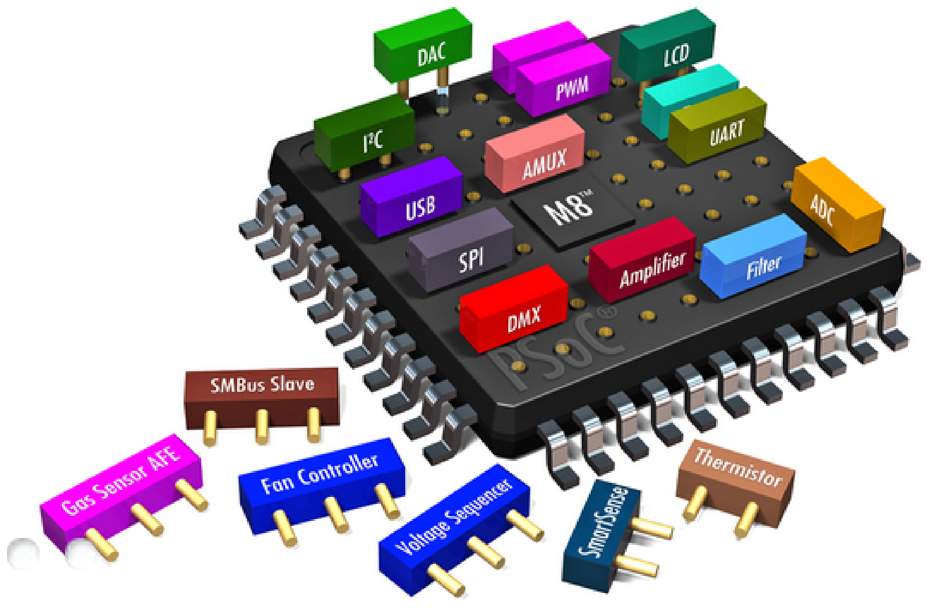


Figure 3.1 *A conceptual picture of an SoC containing many separate IP blocks. Many of the IP blocks, for example UART peripheral, are relatively common and thus using them as reusable blocks greatly reduces design time. [18]*

and can be partially applied also to the field of software development. In the end the focus is moved to IPs and the special considerations of re-use in hardware designs.

3.2.1 General design re-use

The re-use itself can be roughly divided into *blackbox* and *whitebox* reuse. Blackbox means that the user does not see what kind of logic the block contains but only the specification and connection interface. Whitebox (or glass-box) re-use is the case when the source code is visible and modifiable by the user.

In an ideal world, blackbox reuse would be the perfect case: there is no need to do any changes to the design which saves design time and, thus, money. Also the design could remain as a secret and the company espionage would not need to be considered that much. However, the blackbox re-use has its drawbacks. Experience has shown that it is very difficult to make documentation that has good enough coverage. This means that a lot of time and effort is used for communicating the details. Strict blackbox re-use also tends to lead to non-optimized design because the usage and environment of the design might change. In addition, debugging blackbox design units is rather challenging: it is difficult to find out if a bug is in the design or if the design is used wrong.

Sometimes the case is such that the re-use is supposed to happen in a blackbox manner but the code is whitebox. This might be found for example in companies that develop designs for their own use and not for sale. One of the limiting factors in this case is, again, design time. Making code general, configurable or easily modifiable is not easy. In addition to that, the design engineer cannot make too many assumptions with general code, for example that the address bus is always wider than the data bus. This complicates the design further. If the source code itself is supposed to be modified, the code has to be well structured and commented. Again, this increases the time needed for design. Additionally, one has to remember that also the verification becomes more complicated with increasing configurability.

It is also worth remembering, that re-use is not always intentional. It is possible that old design code is partially copy-pasted ad-hoc just because it seems to fit the purpose and someone remembers that such code exists. This kind of re-use is sometimes encouraged since it saves time and effort compared to starting from scratch. However, relying the whole re-use scheme on this is not advisable. This is due to the innate randomness of the method and, thus, problems arising from following the versions, for example, for bug fixes. Additionally, since the design is not designed for re-use, it might include some unwanted optimizations or some other former application spesific structures that complicate the re-use.

3.2.2 General IP re-use

For hardware design, IPs are quite ideal for re-use. This is due to the fact that IPs are mostly designed in RTL level that is essentially independent of final implementation technology. Figure 3.2 represents the ideal process of IP re-use: the IPs are designed as seemingly separate blocks and stored in an IP library. Then, IPs are taken from the library for different design in an as-needed basis in a blackbox or *hard* manner.

Even though blackbox/whitebox terms are used with IPs too, it is more convenient to describe the IP re-use by *hard*, *firm* and *soft*. Soft means that the source code is visible and possibly even supposed to be changed in some parts. A hard block is quite close to the blackbox: the thing that can be designed is where the block is situated on the final chip, for example, a double data rate (DDR) memory controller. It has such a complicated and strict timing that it is better not to touch it when it is working and it is not configurable.

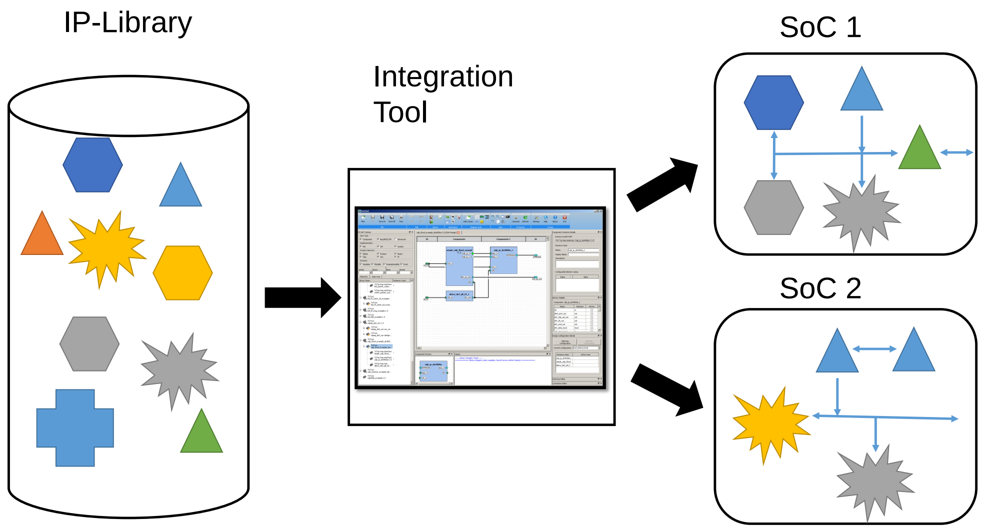


Figure 3.2 *A visualization of ideal IP re-use: the IPs are designed separately and the actual designs are then put together by simply connecting IP from the library. [52]*

Firm is somewhere between soft and hard, for example a gate level netlist that can then be integrated to the rest of the design more easily than a transistor level design. Another good example of firm IP is the NIOS processor provided for Altera FPGAs: the source code is not visible at all but the processor is highly configurable [4]. It is also interesting to note that before being synthesized to an FPGA, the processors does not physically exist.

Figure 2.4 clarifies the division between soft, firm and hard. Usually soft blocks are delivered as RTL code (part 2). Gate level netlists (part 3) do not allow source code modifications but, on the other hand, allow final implementation modification. Thus, they can be viewed as firm blocks. Finally, transistor level layouts (part 4) are not modifiable and, consequently, can be interpreted as hard blocks. The algorithm model (part 1) can be seen as part of documentation and should be delivered with all different types.

To avoid confusion in the naming in this thesis, the blackbox/whitebox terms are used when speaking about re-use in broad manner. Soft/hard re-use are reserved hardware for discussing IP re-use. In addition, the term glass re-use is not used but whitebox re-use.

Occasionally, when delivering designs to a third party, it is in the best interest of the design company to give no information at all about the design internals. In these cases the IPs can be encrypted.

There has also been discussions of platform re-use based design [47]. Platform is the next level of abstraction up from IPs. There are two different interpretations of what platforms are. The firs one understands platform as a subsystem consisting of 5-15 IPs that are connected to each other. Then, when re-use occurs, it will be used as a base for design: something might be added, something removed and something modified. However, more design data is included in platform re-use than with just single IP re-use. The second interpretation understands that a platform is simply the IP library. All in all, further discussion is out of the scope of this thesis. An interested reader can find out more from [47].

3.2.3 Hardware re-use special considerations

The re-use of software and hardware/IPs are not exactly the same. This is due to many reasons: IPs have to work well with other IPs, hardware technology changes, and has to be considered in IP design. Moreover, IPs can have the same functionality with very different types of variably optimized designs.

IPs are seldom used alone. They are usually connected to other IPs and some may require software drivers. In addition, the usage of IPs is not planar: one IP can include several other IPs with some additional logic. Thus, changes in one block quite often affect also other blocks and, in the case of very hard re-use, the changes might even make the block unusable. An example of this could be an IP that performs a change from 24 bit number to 18 bit and does the needed rounding. If this type of general IP is changed, some user blocks might not handle the change well and quite surprising effects may arise.

The hardware technology changes very rapidly and, still, the design cycle for an ASIC might be two years. Thus, when a decision is made to use design time for facilitating re-use, one has to predict the future trends. What if some new bus protocol is implemented? What if the old standard of making memories changes and, thus, also the interface? Doing really hard IPs inside a company for future re-use includes a high risk. Then, on the other hand, if IPs are designed for sale, there are always copyright aspects that support hardness. Additionally, designing IP for sale has higher volume of re-use.

Generality and/or malleability of the design also affects the area, speed and power consumption of the final IC. The effects can be estimated already in the design phase. However, this estimation supposes that there are both versions, not-for-re-use and for-re-use, available. The effects may vary based on what kind of re-usability is considered in the code. For example, good commenting and descriptive signal names do not affect performance. On the other hand, some structures are more reader-friendly than others and these can already result in different final result. The same can happen with a lot of parameters. Also the used EDA tools and final implementing technology might change the effects of coding style.

The generality does not affect only the properties of the final implementation but also the properties of the IP, such as quality, verifiability, testability and characterizability [23]. Figure 3.3 visualizes this effect: the more there are changeable parameters and the more general the design is the harder it is to verify the design and the lower the quality etc. For example, in theory the verification has to be done for all the possible combinations. The amount of combinations rises exponentially as the amount of parameters increases. Thus, verification gets a lot more difficult which lowers the expected quality.

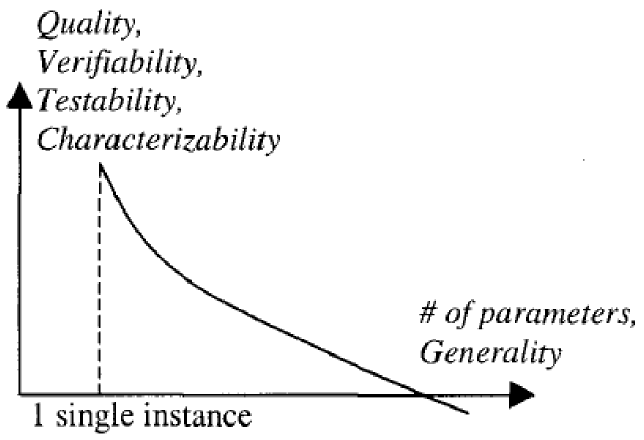
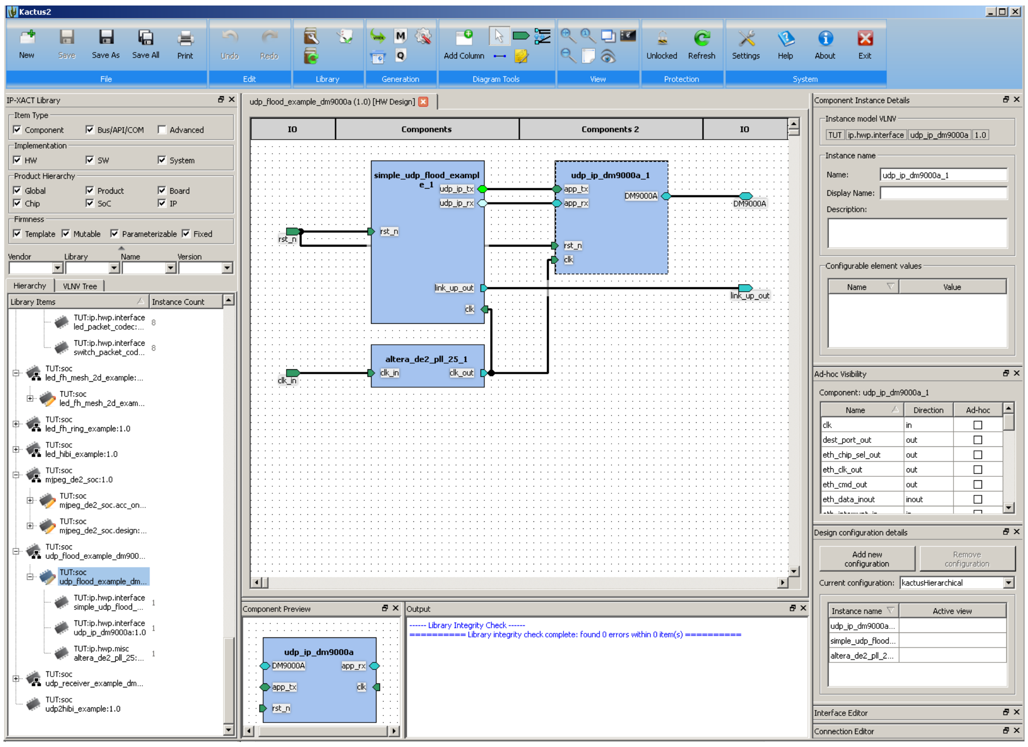


Figure 3.3 *The effects of IP generality to its properties. For example, having many parameters makes it hard to predict the silicon area and timing beforehand (characterizability). [23]*

All in all, re-use is a rather complicated topic with many variations and tradeoffs. Writing good, re-usable code takes more time and effort than writing code that just fulfills the design specifications. On the other hand, not needing to write all the code from scratch saves a lot of time. Then, the generality might also have impacts on performance. Because of all this, it is not an easy task to measure what kind of re-use saves the most money and time. Thus, the decision about how re-usable a design should be and on which level the re-use should be is to be made cautiously and with careful consideration of all the different aspects.

3.3 IP-XACT meta-data format

IP-XACT is an XML-based standardized methodology for capturing information about IPs [8]. It is used to capture metadata about the design, such as interfaces, memory mapping and file lists. Even though the IP-XACT code is somewhat clear for a human reader, it is supposed to be read by machines. Using IP-XACT makes design flow automation much easier because it is automatically created and, usually, automatically read. This reduces the human error to minimum. It can be also used as basis for automatic script creation for simulation and synthesis. Since IP-XACT is standardized, it can also greatly simplify transferring information between design teams or companies.



Component: udp_ip_dm9000a_1

Name	Direction	Ad-hoc
clk	in	☐
dest_port_out	out	☐
eth_chip_sel_out	out	☐
eth_clk_out	out	☐
eth_cmd_out	out	☐
eth_data_inout	inout	☐

Current configuration: kactusHierarchical

----- Library Integrity Check -----
========== Library integrity check complete: found 0 errors within 0 item(s) ==========

***Figure 3.4** Kactus2 is an open source IP-XACT integration tool developed at Tampere University of Technology. This screenshot clearly shows how the IPs can be used as abstraction. Left pane shows the IP library and right panes how they are connected and configured (the blue boxes).[52]*

The real value of IP-XACT becomes clear in the integration phase when different IP blocks are connected together. Figure 3.4 is a screenshot from the Kactus2 tool developed at Tampere University of Technology. It represents nicely the concept of IPs as design entities that can be connected together to create the final design. However, integration phase is out of the scope of this thesis.

4. DESIGNER PRODUCTIVITY

The whole idea of improving design flow is to improve designer productivity. With other words, smooth design flow is one of the many factors affecting productivity. Thus, discussing productivity in general is useful in the scope of this thesis and helps to perceive the big picture. Finding references of hardware designer productivity has proven to be difficult. Thus, this chapter relies strongly on software productivity research and the results are accommodated to hardware design as applicable.

4.1 Measuring designer productivity

The simplest way to measure productivity is to *compare the consumed inputs to the produced outputs* [10]. Measuring human labor productivity in a factory environment has a long history. However, designer productivity is different from this traditional case since defining inputs and outputs in not straightforward [10].

Two commonly used metrics are lines of code (LOC) per hour or function points (FP) per hour. Both of these have valid arguments supporting and opposing them and there is no commonly agreed standard. [22] For example, measurement based on LOC is rather trivial to fool by writing a lot of non-optimized code to appear effective.

4.2 Factors affecting productivity

Factors affecting designer productivity can be roughly divided into technical and soft factors. Technical factors can be fractionated further to development environment, process and product aspects. The soft factors include company and team culture, individual designer characteristics, physical working environment and project properties. [57] This is not the only way to categorize factors [57, 10]. However, the categorizations are often done to fit software purposes and this seemed the most appropriate for the IP development point of view.

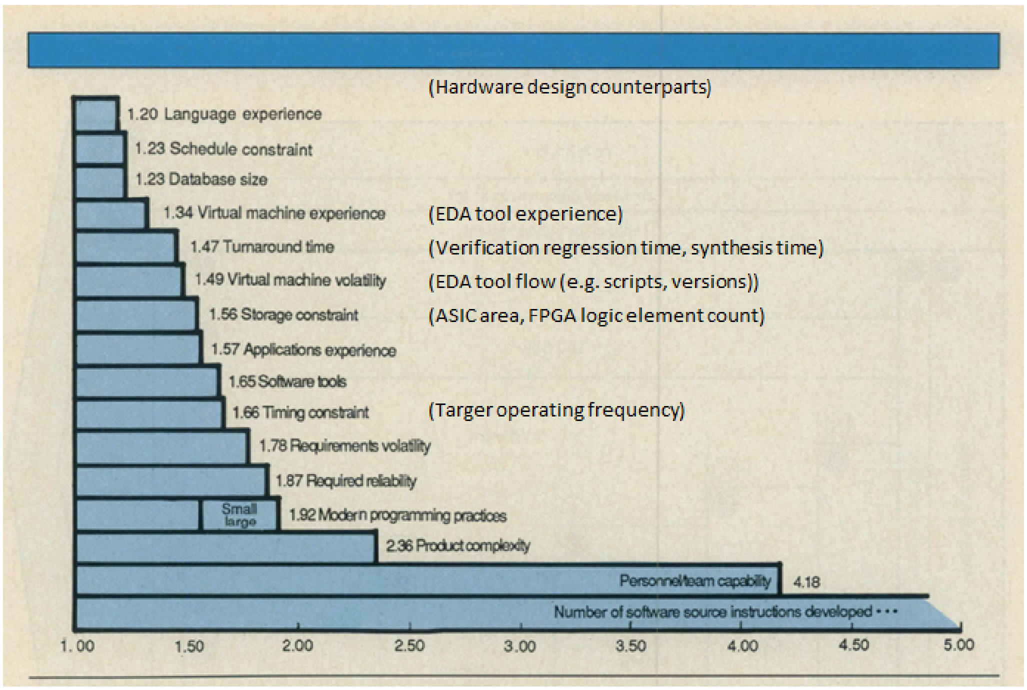


Figure 4.1 *Comparison of different factors that affect software designer efficiency [10]. Clarifications are added in order to demonstrate the connections to hardware design.*

Figure 4.1 shows the results from a study about software productivity [10]. The affecting factors division is not completely consistent with the one used in this thesis. However, one can get the idea and the figure has some connection clarifications added. The number connected to each factor means the coefficient of project duration. For example, if the programmers are not familiar with the used programming language, the development time might increase by 1.20x or if the software tools are not optimal the time might increase by 1.65x. It is worth noting that the most important factors are the team's overall capability and design's general complexity. The study is 30 years old but the issues with software development seem to have remained unchanged [57, 10, 16].

Figure 4.2 gives another aspect of productivity factors in software design. It does not only tell the negative effect of different factors but also the positive effects. The interesting observation is that the numbers are not always symmetrical. For example, if staff is experienced it might increase the productivity by 65%. However, should the staff be inexperienced, the productivity could decrease with 90%. From this one can interpreted that having qualified staff is important but the effect cannot be increased by over-educating the staff. Additionally, by looking the total numbers at the bottom one can see that the decreases total more than the increases, thus, it is easier to decrease productivity than

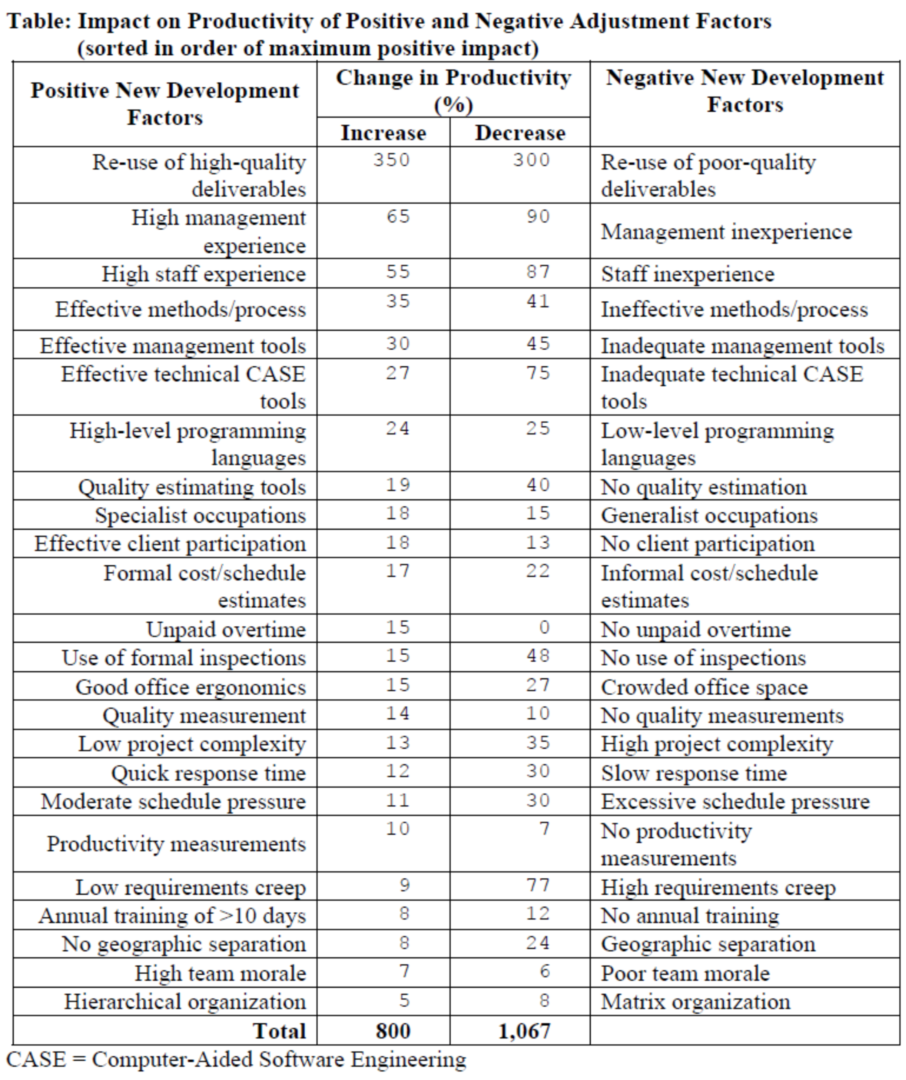


Table: Impact on Productivity of Positive and Negative Adjustment Factors (sorted in order of maximum positive impact)

Positive New Development Factors	Change in Productivity (%)		Negative New Development Factors
	Increase	Decrease	
Re-use of high-quality deliverables	350	300	Re-use of poor-quality deliverables
High management experience	65	90	Management inexperience
High staff experience	55	87	Staff inexperience
Effective methods/process	35	41	Ineffective methods/process
Effective management tools	30	45	Inadequate management tools
Effective technical CASE tools	27	75	Inadequate technical CASE tools
High-level programming languages	24	25	Low-level programming languages
Quality estimating tools	19	40	No quality estimation
Specialist occupations	18	15	Generalist occupations
Effective client participation	18	13	No client participation
Formal cost/schedule estimates	17	22	Informal cost/schedule estimates
Unpaid overtime	15	0	No unpaid overtime
Use of formal inspections	15	48	No use of inspections
Good office ergonomics	15	27	Crowded office space
Quality measurement	14	10	No quality measurements
Low project complexity	13	35	High project complexity
Quick response time	12	30	Slow response time
Moderate schedule pressure	11	30	Excessive schedule pressure
Productivity measurements	10	7	No productivity measurements
Low requirements creep	9	77	High requirements creep
Annual training of >10 days	8	12	No annual training
No geographic separation	8	24	Geographic separation
High team morale	7	6	Poor team morale
Hierarchical organization	5	8	Matrix organization
Total	**800**	**1,067**	

CASE = Computer-Aided Software Engineering

Figure 4.2 *Comparison on the positive and negative factors for productivity. It is worth noting that the factors are not symmetrical, for example the positive effect of high management experience is lower than the negative effect of management inexperience. [24].*

increase it.

In hardware design the two single most influential productivity improvements have been the development of EDA tools and IP-block re-use [29]. On the other hand, to further improve the re-use is one of the concurrent challenges.

The following two subsections discuss the technical and soft factors in more detail. However, deep analyzes are out of the scope of this thesis. An interested reader is encouraged to find out more from, for example, [57] [10] [16].

4.2.1 Technical factors

The most important technical factors for the IP development include tools, design unit complexity, amount of re-use, documenting and development practices and processes. It is vital for productivity to acknowledge the different factors. However, many of them are already discussed elsewhere is this thesis and, thus, are omitted from here. Tool related issues and solutions can be found in Sections 5.7 and 6.6, respectively. The design unit complexity refers to the concept of IP that is discussed in Chapter 3. Additionally, re-use is also addressed already in Section 3.2 and documentation in Sections 5.3 and 6.3.

Based on author's personal experiences, processes and modern development practices seem often to be despised by technical engineers and scientists. They are seen as an unnecessary bureaucratic load and means for managers to infer one's "real" work. However, there is a sensible reason behind many of these activities. For example, with processes the company management tries to minimize the effects of different personalities and skill sets by defining the way how things are supposed to be done. In addition to the common clarifying effect, this also decreases the quality fluctuations.

Processes are not only about the design flow but the scope is much wider. There are processes for specifications, documentations, and different human resource related topics, just to mention a few. Also the synchronization of different processes is important. For example, implementation and verification should be synced in order to achieve smooth flow.

Even though processes are important, the processes themselves should be sensible. After all, processes exist to facilitate designer work and not the opposite. Designers do not exist for the processes. Additionally, it is important to remember that everything cannot be completely controlled by processes and trying to achieve total control only leads to unnecessary complexity and general disgust against even the word process.

Modern development practices, such as scrum and lean, have been shown to increase productivity. There are comprehensive methodologies for lean product development. [35] However, the methods in such an extensive scale affect rather high level processes and, thus, are not discussed further in this thesis. An interested reader should refer to [35] or [54] for more information.

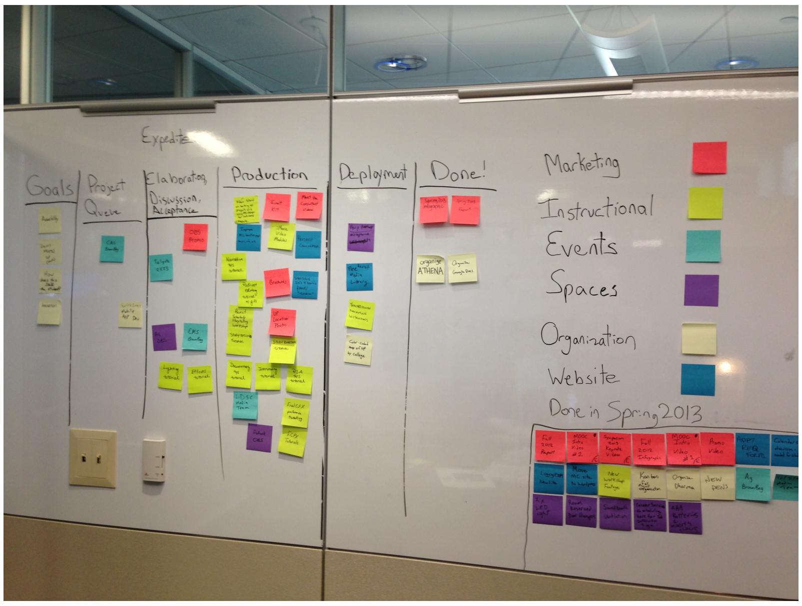


Figure 4.3 *A kanban board used in project flow management. [59]*

Nevertheless, already quite simple, low level tools, such as visual work flow boards, increase productivity [35, pp. 81–114]. Figure 4.3 shows a typical kanban board used for project flow management. This kind of visual presentation makes the complex design tasks more manageable and, consequently, reduces the overall chaos. From the board it is easy to see quickly the present state of flow and identify possible bottlenecks. When usage of visual workflow is combined with short duration and frequent stand up meetings, the effect can be enhanced [35]. Proper management also ensures that there is no unnecessary idle and that multitasking is minimized. Multitasking is a very inefficient way to work [5].

4.2.2 Soft factors

Many of the soft factors concern rather humane aspects, such as team and company culture or individual designer capabilities [57]. The comprehensive discussion of these is left out from this thesis. However, here is a short summary: people should enjoy their work in all the levels and know what to do in order to be productive. Further reading can be found from [57] or [56], for example. Also schedule is part of the soft factors. However, it is rather related to the process level planning and, thus, left out of this thesis.

The work environment is a vital productivity factor [33]. Design work needs intensive focus and minimizing the distractions is important [31]. The modern trend to use open office premises has been proven to be harmful for productivity since it is often filled with discussions and other distracting stimuli [40]. Nevertheless, the cost effectiveness, malleability, and team work synergy benefits suggest that the trend is not changing. There are various solutions for improving the possibilities of concentrated work, such as silent work rooms or commonly agreed rules to avoid loud discussions. However, the silver bullet is yet to be found.

Background noises are not the only type of distraction. Distractions can be anything from constantly appearing new emails or instant chat program message notifications to unnecessarily frequent meetings or people just appearing to your desk to chatter. Minimizing these can be achieved, for example, with shutting the chat and email programs down and somehow signaling to one's coworkers that unnecessary bothering should be avoided.

Additionally, it should be ensured that the designers work is physically comfortable. For example, physical discomfort or pain can easily harm concentration levels and, thus, decrease productivity [50]. Work comfort has quite often been taken care of by ergonomic workstations, correct lighting and preventive occupational healthcare. Naturally it is not enough to provide services but also encourage people to use them when needed and promote a healthy lifestyle in general. It is also suggested [25] that taking a nap in the middle of the working day might improve productivity.

Modern design work is done in teams. Considering this from the productivity point of view, some aspects arise: how many members does the team have and how are the people located geographically. Team sizes should not be too large or too small [60]. The closer the other team member is located to you the more efficiently you can communicate and, consequently, the more productive the team work will be [13, p. 39]. The communication in general is discussed in more detail in Sections5.1 and 6.1.

5. IP DESIGN FLOW ISSUES

From a high level perspective, IP design flow might seem straightforward: specify the design, design it, and verify it. Then, integrate the IPs to obtain a functional SoC. Of course, things are not that simple. The steps are overlapping and large amounts of really specific information has to transferred from person to person and tool to tool. In addition, the process relies heavily on EDA tools and includes many steps that are automated with scripts. Problems are lurking around everywhere.

This chapter seeks to identify the most common issues of IP design flow. The reader must remember that many of the themes are more or less interdependent of each other. For example, tool versions affect scripts and, generally speaking, unclear documentation is a part of inefficient communication. The issues are divided by issue type and not related to any specific step in design process.

5.1 Inefficient communication

Modern IP development projects are large and complicated. They are done in teams of many people and co-operation between teams is frequent. Additionally, the teams are often located in different geographical locations and in different timezones. Many of the tasks are automated by tools or scripts. Moreover, the different phases of the same design are usually done by different people. The amount of people and handovers, in addition with design complexity, create a huge amount of information that needs to be transferred. Consequently, many of the design flow issues are connected to information flow issues.

The channels and means for communication are versatile: speaking, computer chats, emails, and official documentation, just to mention a few. Then, in addition, there are ways to deliver information that are not so obvious, for example, comments in code or self explanatory code. Some parts of communication are controlled, such as official documentation. Thus, they can be affected via better processes. However, the most frequent part, person-to-person communication, is not controlled. This is a huge challenge for solving communication issues.

Information transfer can fail in many ways. Things are not understood at all or misunderstood. It is also possible that the correct information is available but transferred in an inefficient way. This wastes time. Sometimes the intention of the communication is not clear. This can be seen, for example, in writing self explanatory code; it takes more time and effort to write self explanatory code and if the developer thinks that his/hers only goal is to have a functional program, then the extra effort might feel useless. However, if the intention is understood, that is the code is easily understandable and, thus, re-usable, the extra effort could be justified.

Occasionally the problem with communication effectiveness is not in the information delivery format itself but rather in finding the essential information from the cornucopia. For example, already badly written documentation can be further worsened if the document contains a lot of non-essential information.

5.2 Finding correct files and documents

Finding correct files and documents has proven to be difficult, even one of the most frustrating issues [16]. The problem is caused by different factors. The most obvious one is the sheer amount of files due to the complexity of the projects, including the official documentation, scripts, and the actual design files. However, there is more to this: the files are usually stored in many different systems, e.g. Windows/Unix, version repository, intranet, document management system, bug database, team wiki, home directory, and project disk. The decision of which system to use can be based on various official guidelines or simply personal preferences. In addition, the search functions might be limited or nearly useless in these systems. Consequently, a lot of effort is needed to find a person who knows where something can be found.

Figure 5.1 represents a typical situation where the amount of files is big, for example tens of thousands of files for a large ASIC project. These files are stored in different systems, there is an excessive amount of folders, and the design engineer does not know how to start and how they link together. The left side depicts the in-house information, such as workstations, intranet, and emails. The right side shows the necessary 3rd party background information, such as basic theory and user manuals. Usually there are no proper links between systems, as they are tedious to maintain, especially regarding versions, and the terminology often differs. Moreover, the bottom left corner includes informal notes that are on designer’s desks or shelves and the most difficult part: tacit knowledge. The term refers to such information that is hard to transfer, but something that experienced designers "just know".

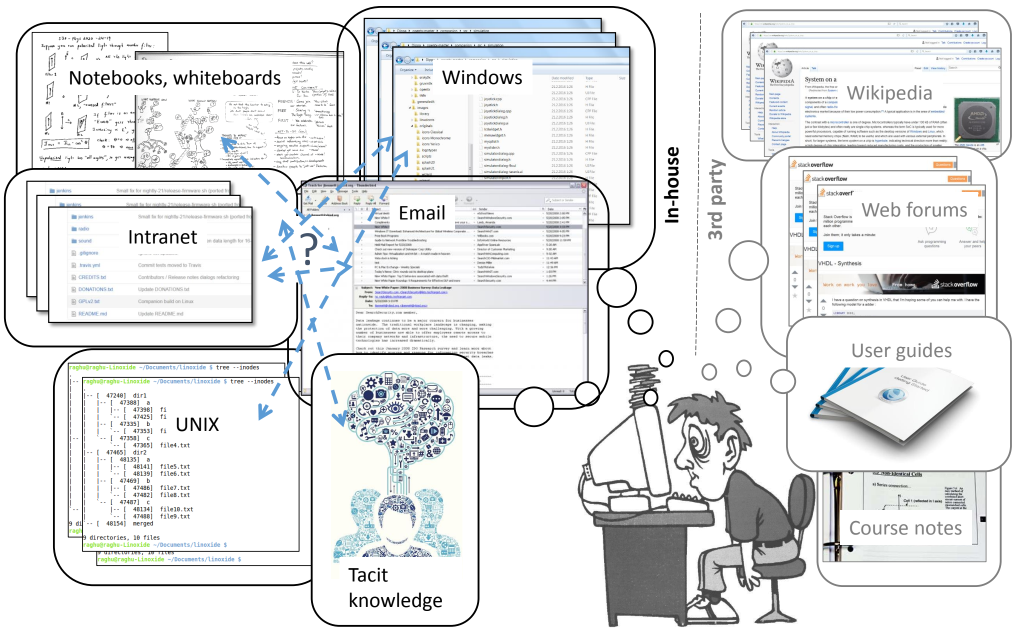


Figure 5.1 *A typical design engineer confused by files. [21]*

In addition to the official and needed files, there are always a lot of extra files stored as well. Some of them might be generated by tools and put to version repository accidentally or because the person putting them there has not really understood what the file is for. This lack of understanding also applies to legacy files: when one does not really know what, for example, an old script is used for, it is saved just to be on the safe side. Humans have limited ability to handle information [31] and, thus, finding correct files in a folder with a lot of extra files takes inevitably more time. Some files need modifications in order to be used in a different environment and, occasionally, the file name might suggest such action even though the file should just be removed. This creates confusion. Another related problem is the creation of default folders which remain unused. This is usually done automatically in order to have a generic folder structure with different designs.

Once the file is found, it is often important to be able to compare files with each other. This is relatively easy for purely textual files like source code. However, tools like Excel and Word are also widely used. Comparing two Word files is not too straightforward. Also schematic drawings are notoriously difficult to compare.

The importance of storing files logically and having clear folder structures is highlighted in re-use concept. If designs are revisited after a long time or in a very different location, it can be really difficult to find and contact a person with insights to it, sometimes even impossible. In addition to this, storing extra files for extended periods might not be efficient.

5.3 Unclear documentation

Written documentation is the most used official way of transferring information at NSN. Writing good, understandable documentation is difficult and needs a lot of effort. Tolstoi's Anna Karenina principle [61] can be applied to documenting: *all the good documents are good in the same way but all the bad documents are bad in their own special way*. Thus, writing bad documentation is much easier than writing good.

It is well known that good quality documentation is vital for efficient knowledge transfer and low quality documentation is likely to cause problems in the future, especially if the design is planned to be re-used. The re-use also sets some additional challenges for documentation since usually the design is not re-used alone but with, for example, parts of the verification environment. In this case, the documentation should contain information of how the environment is run and what should happen if everything works out well.

One colleague who is working with technical documentation put this issue quite nicely by saying that her job is *"to translate documents from Engineer to English"*. Far too often documentation is unclear and sometimes it creates more confusion than explains things. The problem with documentation quality is, in the author's opinion, due to 3 main reasons: lack of time, lack of motivation, and lack of skills. The design engineers do not have time to write well, do not want to write well, or do not know how to write well. It is also possible that there is any combination of these three.

The project schedule allocations for document writing are sometimes too low. Still, there is no time during the most hectic design phase when the details are fresh in memory and, thus, writing documents would be the easiest. Then, after the design is close to ready, a new design steps in and, again, no time for proper documenting.

In addition to the lack of time, many design engineers find documentation to be more of a burden, a necessary evil, than actual part of their work [16]. Figure 5.2 visualizes this. It shows the design flow adapted to the designer mindset where designing is the main focus, the "real" work, and documentation excessive controlling of one's work. It doesn't help the situation that documentation is not only written for designers and users but also for quality auditions. This means that the official documentation needs to follow certain formality such as ISO-9000 [28]. This formality itself doesn't make the text bad, of course. At most it adds some rather useless chapters from the colleague reader point of view. However, sometimes it seems that people resist writing good documents because of this formality. They feel like they are only writing the documents for the reviewers. Engineers seem to have quite poor understanding about quality topics in general, which adds to the general resistance against documenting.

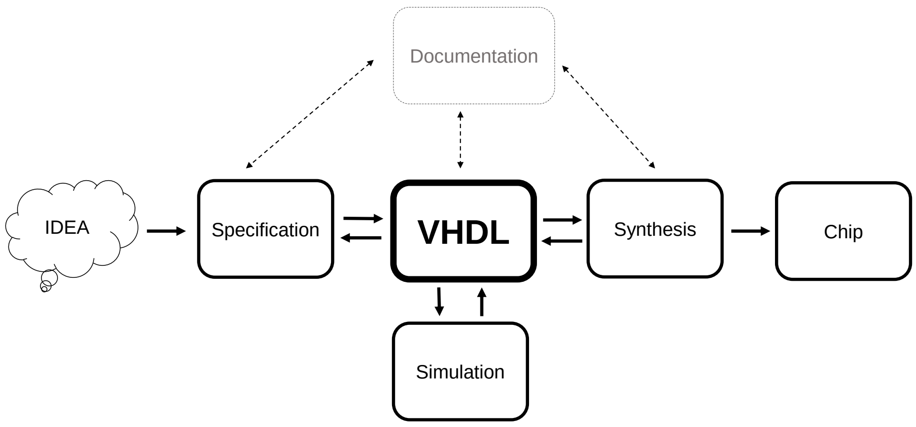


Figure 5.2 *The design flow from the designer point of view. The solid line represents the "real" work of a VHDL designer and dotted line the documentation that is often seen as excessive bureaucracy.*

The general lack of writing skills has to be considered, too. Even though technical writing is a difficult topic, it is something that can be learned. However, in order to actually learn something people need to have the willingness to learn. Conclusively, the above mentioned motivation issues also become important here.

The writing of documents is not the only aspect that takes time in documenting. The reviews can also be rather painstaking. It is also possible, that due to the organizational reasons, one person is burdened with too much reviews. This might lower the quality of the reviews and make them even useless.

5.4 Unclear code

Martin Fowler, a British programming expert, once wrote [65]: "*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*". Code readability has a big impact for design quality, re-usability and designer efficiency [31, 34]. It is also shown that software developers actually use more time searching for correct spot in code and reading code than actually writing it [31].

Re-use highlights the importance of clear code. At NSN there is a lot of soft IP re-use. This means that the code is most probably used again at some point. Since technology develops, modifications are often necessary and, thus, code is also modified by other designers.

Even if the code is not designed to be re-used, the importance of clear code remains. First of all, not all the re-use is intended and the code can be re-used without prior planning. Second, the code is not read only by designers but also by many others: back-end designers, verification engineers, FPGA prototype designers, and even the ones writing software that use the IP. Thus, the effort invested in clear code is likely to pay back already in the first time of use.

Code can be unclear in many different ways. For example, signal/variable naming might not be sensible, commenting might not cover enough or it can cover unnecessarily much, or code might not be structured well. Using methods of object oriented programming has proven to be efficient [41]. However, hardware description languages, such as VHDL, do not have many of these features. Despite that, VHDL is not unclear by definition and following certain rules of thumb already provides great improvements.

The reasons behind the lack of clarity are similar to unclear documentation with some differences. Code is the main product of designer's work and it has to be done. Still, writing clear code usually includes following official guidelines or at least giving a lot thought for something else than pure functionality. This, again, results in resistance.

5.5 Scripting

Scripts are widely used in IP design and also cause problems. Most of the scripts are intended to work as-is or with minor modifications, but quite often they don't and it is not clear what modifications are actually needed. Thus, a lot of time is wasted with bad scripts even the scripts would be functional per se.

Scripts suffer from many of the same issues as code. However, scripts also have their own peculiarities. Scripts are not compiled but interpreted, and hence all errors are detected at runtime. The syntax of scripts is often really strict [2] and UNIX shell script languages that are almost the same but not really [55], these complicate fast visual identification of the problematic spots.

Broken paths within scripts are common cause for problems. Scripts often refer to shared folders or environmental variables and long concatenated paths are not rare. This makes understanding the dependencies difficult. Another thing, which constantly breaks scripts, are wrong tool settings. Tool switches can be strongly version dependent and, thus, scripts might get outdated every time some tool version is changed.

When scripts inevitably stop working, the user is usually in trouble. Scripts are often poorly documented and the help is not done properly. Additionally, since the errors happen at run time, the failing script might have done something and is then just frozen in some strange state. If error messages appear, they are usually worse than those of compilers.

The following example illustrates a few common problems in scripts: insufficient header, unclear variable naming, no printing of messages and long distance between variable declaration and actual usage. In addition, there is a typo in the path (should be: work) that will break the script.

```
#!/usr/binExampleShell
##############################
# Version: new
##############################
set PATH_1=$BASE_DIR/../../$USER/workk/foo/vhdl
.
... 100's of lines in between
.
cp PATH_1/source_1.vhd source_1.vhd
```

Problems with scripts are sometimes caused because they are planned only for one's personal use to start with. Then, suddenly, some colleague needs to do the same job and asks for help. The script is shared and then the new user makes some modifications and possibly shares the script further. This leads to a situation where there is a collection of undocumented scripts that is a terrible mess.

One must also remember, that when a design is re-used, also scripts are usually re-used and it is possible that the environment changes considerable. This is likely to cause problems if scripts are written in unclear style and possible modifications are not considered.

5.6 Version control

During IP development, version control is used almost daily by the designer. It enables effective sharing of information, tracking changes, and reverting to old versions if bugs creep into the code. Figure 5.3 visualizes the version control dependencies and usage. It shows different version types with different colors, dependencies with arrows, and additional information about reasons for some versions. As one can see, the system gets complex rather easily and, thus, the utilization of version control is not problem free.

Some of the issues with version control are similar with other IP development issues. For example, insufficient, unclear, or missing commit messages are in the same category with unclear documentation or code. Version control is also linked to the file finding issues. Too often designers include unnecessary files to version control that cause confusion later. This can happen by accident or by lack of knowledge. Additionally, the opposite is possible too: not including files in version control when they should be there. For example, scripts can be easily forgotten to be included or updated, and the only copy resides in the designer's personal directory.

Some file formats have proven to be difficult for traditional, text based version control systems [49]. These include, for example, frequently used Word, Excel, or pdf files. The problems are related to the large size of the files and difficulties in comparing different versions. Thus, every version has to be stored as a whole and not only the changes to older versions.

The version control itself also contains some fundamental issues [20]. For example, how to solve merging of two files that have changed a lot or to decide which branch to choose. Additionally, there are hazardous occasions when someone has modified an old tag. A tag is meant to be a frozen release, so that error reports and synthesis reports, bug fixes, and documentation can refer to a known set of files. Changing the content of tagged version leads to extremely harmful mismatching copies depending on when codes are checked. Thus, it should never be done, no matter how well one knows what one is doing.

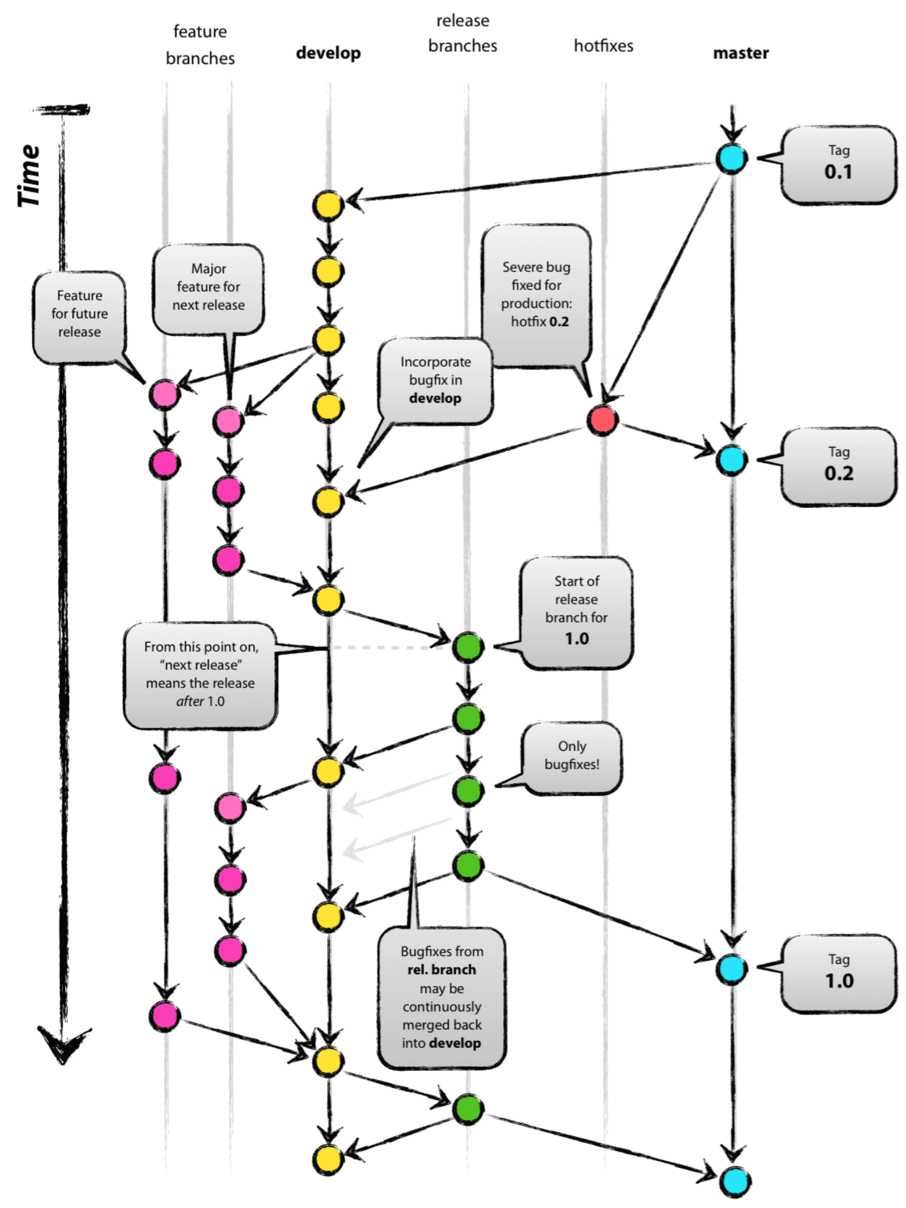

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Figure 5.3 *A visualization of version control. [19]*

Many of these version control issues within one company can be managed with proper processes, guidelines and project management. However, some issues are related with sharing information with collaborators or customers. When there is proprietary information included and everything is not for sharing, more caution is needed. However, this thesis concentrates on IP development process inside the company and, thus, this last aspect is out of the scope of the thesis.

5.7 Issues with tools

IP design relies heavily on different tools. The code is written with some text editor, preferably supporting VHDL, the code is simulated with sophisticated EDA software and usually some trial synthesis is run. In addition to that, there might be some usage for additional linting or other code analysis tools. EDA tools are an important technical productivity factor. Fighting with tools causes frustration and a lot of resistance against trying new ones. On the other hand, proper tools increase productivity [10, 24, 57].

Re-use is a common situation for tool problems to emerge. The related scripts are likely to be outdated and some of the created files may not be compatible any more with different tools or tool versions. However, the re-use specific tool problems usually arise only once in the each respective project and, in the case of strict black box re-use, they might not arise at all.

Many of the EDA too issues originate from licenses. EDA tool licenses are expensive. The cost might be tens of thousands of euros per one user per one year. Hence, companies only invest in a limited number of them. This might cause unnecessary waiting, especially without proper license management system. Additionally, licenses themselves don't always work and the license error message tend to be unclear. These problems might be impossible to solve without the help of the IT department or the vendor. Communication with 3rd party vendors can be time consuming and frustrating. Additional issues are caused by the variety of offered licenses, for example for evaluation purposes that might be restricted in design size or time usage, or academic vs. commercial institute.

Another problem with licenses is that some companies only give a Windows possibility whereas others only for Linux. This interrupts the work flow and forces the designer to unnecessarily copy files between systems. Also some unexpected problems may arise with using two different operation systems: once the writer used around two hours for debugging a script's error message that seemed to come from nowhere. In the end, with the help of colleagues, the reason was discovered: at some point the code file had been edited in Windows environment and, consequently, all the line endings were automatically changed. A quick *dos2unix* command fixed everything.

EDA tool vendors make an understandable effort to maximize their business. One way to achieve that is to offer a huge variety of special software for many different purposes. These specialized programs can offer extra features useful for designing. However, this offering also has a darker side. By dividing features to different tools rather than providing less but more comprehensive versions the customer company is forced to buy more expensive tools. This division also wastes the designers time, because design has to be transferred from one tool to another.

Another method the vendors use to improve their business is trying to create so called *vendor lock* [64]. Vendor lock means that the customer is, more or less, forced to use tools only from one vendor. The lock can created by using, for example, vendor specific file formats and by not providing proper import-export functions. Even though extreme cases are rare, the tools from different vendors tend not to work well together. These things have been tried to fix with standards, such as IP-XACT [8]. However, quite often vendors try to make their own extensions to standards that can be compared to the vendor specific file formats.

In addition to the vendor lock aspirations, the EDA companies often try to ensure that the programs are not used without licenses. This has lead to, for example, node locking. In modern agile development world, this makes things rather difficult if a license if tied to the certain network card MAC address.

If problems arise with proprietary tools, the user can rarely do much. It depends a lot on the company but sometimes the support is not too good. User sends a ticket through some reporting system and then just waits which is waste of time.

Different tool versions have also proven to be problematic. Occasionally the files generated by earlier version are not compatible with newer versions. The versions may also differ in the switches used to specify the program functions in command line. This affects scripts and can cause surprising errors that are difficult to debug. Even if the tools used in a single project could be harmonized, different departments of the company are likely to have version differences. This is due to the difficulties of switching from one version to another. Additionally, it is rare that one tool or version fits for all purposes.

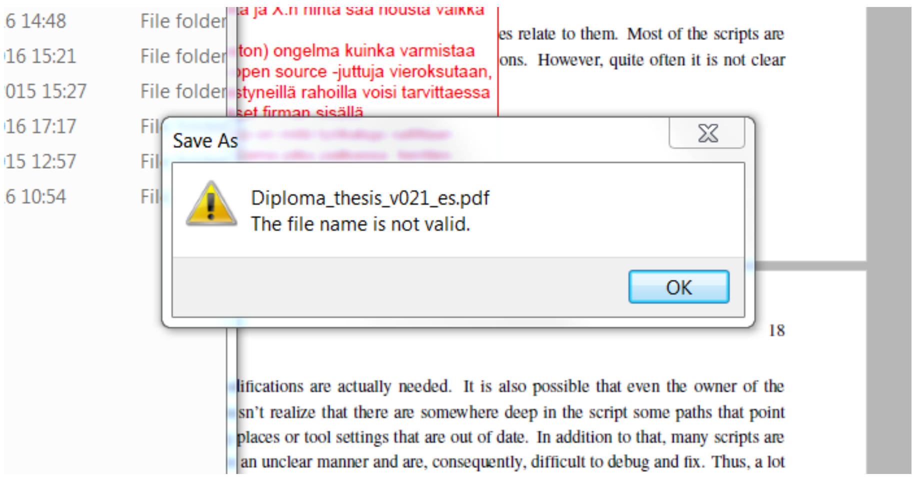
Many of the EDA tools are used through different computational facilities. Consequently, also they have an effect in the smoothness of tool usage. Unnecessary waiting in queues is often wasted time from the designer point of view as is slow computation, too [10].

There are also many smaller issues with tools. The following list gives some good examples:

- Unclear and non-informative error messages

- False warnings that complicate especially the analysis of log files
- Different text editors may break indentations that work nicely in some other
- Automatically generated names can be so long that some tools cannot handle them anymore
- Many tools do not accept all the characters, for example, in the file names
- Tools may require admin rights to the computer which may not be possible because of company policies

Figure 5.4 shows an example of non-informative error message that the author got while writing this thesis. It came when trying to save a version with comments in a pdf-software. The message is clear: there is a problem in the file name. However, there is no information about how to fix the name: is it too long, does it contain wrong characters, what are the actual wrong characters etc. In the end, the accepted name was *thesis*.



Diploma_thesis_v021_es.pdf
The file name is not valid.

Figure 5.4 *A typical example of a non-informative error message. The name is wrong but there is no hint why it is not accepted.*

Even if everything else is working well, the tools themselves can be difficult to use or otherwise worthless. Poor usability decreases productivity [10, 24] and causes common frustration. The same applies for remote computational facilities. The usability issues can rise from strange operating logic or badly designed graphical user interface, for example.

5.8 Summary of the issues

The most common issues of the IP design flow can be categorized in the following way:

- Inefficient communication
- Finding correct files/documents
- Unclear documentation
- Unclear code
- Scripting issues
- Problems with version control
- EDA tool related problems (including IT-support and computational platforms)

Some of the issues are a bit overlapping and, for example, unclear code problems have a lot in common with script related problems.

6. SOLUTIONS TO GENERAL ISSUES

This chapter discusses solutions for IP design flow practice issues. The focus is on solutions that might be implementable rather easily, for example, by paying a bit more attention or following simple guidelines. Organization level process or hierarchy changes are out of the scope of this thesis.

The division of the sections follow loosely the division of issues in the previous chapter. Some of the topics are discussed in more detail than others. This is due to the fact that some issues have more low level enhancement opportunities than others. The space given for different topics does not reflect the importance of the topic.

6.1 General person-to-person communication

Unlike written documents, person-to-person communication happens mostly in an informal, spontaneous, and ad-hoc manner. It is strongly affected by one's personality. Trying to improve such communication might be difficult since feedback is easy to take too personally. Still, it is worth the effort to try to enhance and encourage richer person-to-person communication, it might not be effective with all the individuals but the overall effect should be positive. This could be done by different kind of trainings or guidelines. Further discussion of these is out of the scope of this thesis.

Sometimes the problem in information sharing is not about how information is shared but, more or less, about what information is shared. Sharing unnecessary information might cause overflow hiding the important bullet points underneath. On the other hand, not having enough information lowers the quality and ends up wasting everyone's time. One solution of this could be simple check-lists of information that needs to transferred, for example, between implementation and verification.

The better people know each other the easier the communication becomes and the threshold for asking help or express one's opinion decreases. These are important aspects of teamwork. Also, the commonly used teleconference calls get more efficient if one has met the other person face-to-face at least once. Thus, the company should pay some attention to team building and to make people familiar with colleagues working on different sites if they work in the common project. Moreover, common, project barrier breaking coffee breaks could be encouraged.

There are also other methods for lowering communication threshold. Intranet and chat program profiles should always include pictures and finding people could be enhanced with office seating maps.

The comfort of using a certain information channel affects the easiness of communicating. Some people like to talk face-to-face, some to chat via a program, and some to write emails. Also timing is critical: people seem to receive information better when they actually have to need for it. These aspects can be considered by everyone. However, these are really important from team leader point of view should they want to communicate better with their team.

Physical distance effects how the people communicate. It is easier to take contact with someone sitting next to you than in another office space. The effect is further amplified when the city or even country is changed. The writer's personal experience supports this idea. It has also been noticed by Robert P. Colwell, the head architect of Intel's Pentium 4 processor. At Intel they actually even went that far that they designed the seating arrangement based on the processor final floorplan. [13, p. 39] Some part of the physical distance effect can be explained by not knowing the people personally or cultural differences. However, it still exists even in smaller scale and should be considered when making design team allocations and local seating plans.

Additionally, people distance matters in communication. This means adding intermediate people between the actual communicating parties. This effect is already known from the children's broken phone game in which the whispered message changes completely during the round. This has already been noticed in software industry where designers are often in direct contact with customer.

6.2 Making files easier to be found

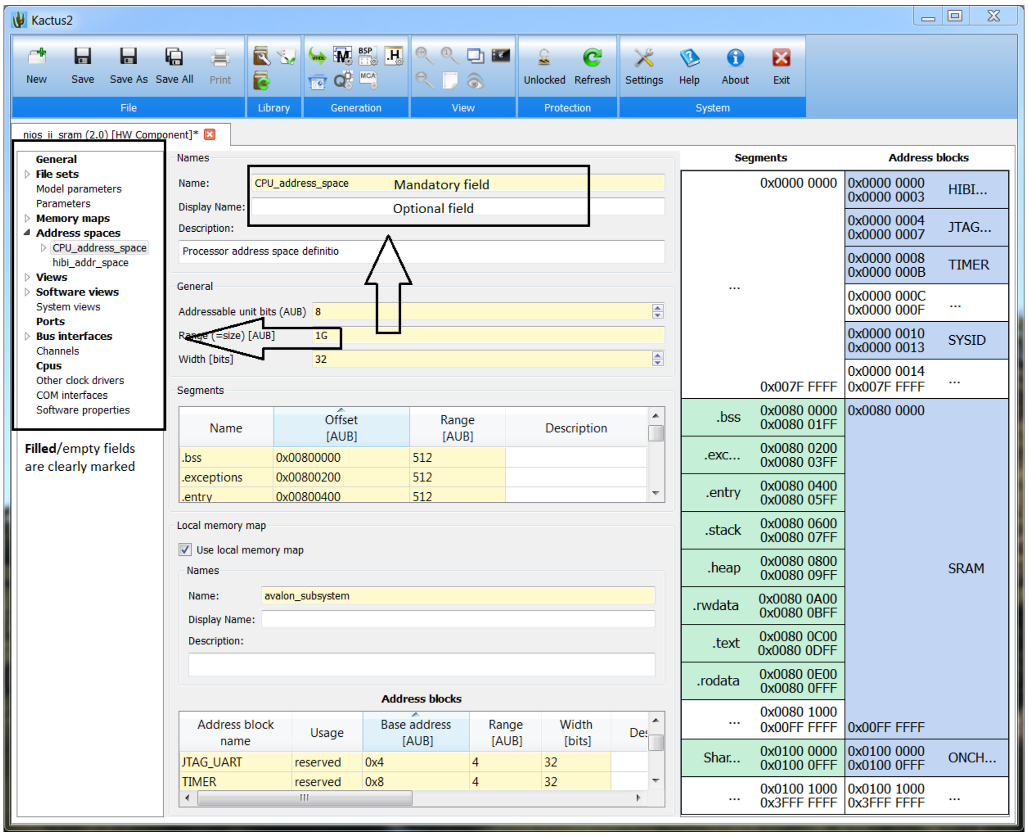
Reducing the number of files eases file browsing. Therefore, the amount of files should be reduced if possible. This can be achieved by periodically going the files through during design and check if all the files are actually relevant and up-to-date. After that unnecessary files should be deleted. If deletion seems too permanent, legacy folders can be created in order to store, for example, old scripts that might be needed but are not vital for everyday work. Then the files would be out of sight, thus, decreasing confusion. The same should be done with special care before handovers because the other designer might not know which files are important and which not. Extra care should also be considered when placing files in version control.

Also too complicated folder structures should be avoided. Empty folders should not be generated just for generality purposes. If default folder structures are really needed, the situation should be clarified somehow. Figure 6.1 shows an example of graphical solution from Kactus2 tool. Default sections are created based on IP-XACT schema, however, the ones actually containing information are written in bold text. This idea can be extended to folders which clarifies situation a lot. Additionally, the compulsory fields are marked with yellow and possible errors with red.

In addition to the folder structure and amount of files, the file names also have an effect for finding files. The names should be descriptive and follow some format consistently. For example, if design specification of a FIR filter is named *projectx_firfilt_design_spec.doc*, the verification plan should be *projectx_firfilt_verif_plan.doc* and not be *verif_plan_FIR.FILT-projectx.doc*. To further ease the file navigation, one should always use the filename extension, such as .vhd, .txt or .pptx. The filename extension enables also simple double click opening of the most of the files.

The filenames and folder structures should be thought already in the beginning of the project before it gets too much out of hand. Making changes to folder structures and even filenames that are in version control is time consuming and may lead to difficult situation. An example of this kind of situation is tree conflict in Subversion SVN that is almost impossible to solve. However, not everything can be known in the early phase and, therefore, this should not be overdone but allow some flexibility.

Sharing files by email, chat programs or other unofficial ways is a sure way to chaos. Files should always be stored in an official, common system and, then, only the link should be shared. In addition, the link should be included in the file headers. By sharing only the links and emphasizing the link usage, one can ensure that the file is backed up and that people can easily bu sure that they have the latest version. Also, additional document list files with links can be created if needed.



Names

Name: CPU_address_space

Display Name:

Description: Processor address space definitio

General

Addressable unit bits (AUB) 8

Range (=size) [AUB] 1G

Width [bits] 32

Segments

Name	Offset [AUB]	Range [AUB]	Description
.bss	0x00800000	512	
.exceptions	0x00800200	512	
.entry	0x00800400	512	

Local memory map

☑ Use local memory map

Names

Name: avalon_subsystem

Display Name:

Description:

Address blocks

Address block name	Usage	Base address [AUB]	Range [AUB]	Width [bits]	Des
JTAG_UART	reserved	0x4	4	32	
TIMER	reserved	0x8	4	32	

Segments		Address blocks	
...	0x0000 0000	0x0000 0000 0x0000 0003	HIBI...
		0x0000 0004 0x0000 0007	JTAG...
		0x0000 0008 0x0000 000B	TIMER
		0x0000 000C 0x0000 000F	...
		0x0000 0010 0x0000 0013	SYSID
	0x007F FFFF	0x0000 0014 0x007F FFFF	...
.bss	0x0080 0000 0x0080 01FF	0x0080 0000	SRAM
.exc...	0x0080 0200 0x0080 03FF		
.entry	0x0080 0400 0x0080 05FF		
.stack	0x0080 0600 0x0080 07FF		
.heap	0x0080 0800 0x0080 09FF		
.rwdata	0x0080 0A00 0x0080 0BFF		
.text	0x0080 0C00 0x0080 0DFF		
.rodata	0x0080 0E00 0x0080 0FFF		
...	0x0080 1000 0x00FF FFFF	0x00FF FFFF	
Shar...	0x0100 0000 0x0100 0FFF	0x0100 0000 0x0100 0FFF	ONCH...
...	0x0100 1000 0x3FFF FFFF	0x0100 1000 0x3FFF FFFF	...

Figure 6.1 *Bolding is one way to help designers distinguish which options have information and which not. The figure is from Kactus2 tool [51]. The reader can see also other ways to enhance clarity with color usage.*

Figures can be used to ease file navigation, too. If one has a Powerpoint presentation, it is more informative to not only show the link but also attach a screen capture of the location in the respective system. Additionally, the version control ID numbers and other relevant information can be highlighted in the figures. This helps especially with clumsy user interfaces that this kind of software too often has. On the other hand, also good solutions exist, for example the SVN web interface.

Occasionally, the issue is not finding a specific file but, moreover, knowing if a file should be found. For example, one might want to know if there is already a script for compiling all files or if it should be created. Another thing is to know if some file needs to be modified after tool updates or similar changes. An additional readme.txt file could be added that includes all the vital metainformation about the other files. The information about how the file should be modified can be included in the header of the file itself. Additionally, using standardized packaging methods, such as IP-XACT, help to solve this aspect.

6.3 Documentation readability

Documentation is a really widely studied topic [26, 58] and mostly out of the scope of this thesis. However, in this section some aspects will be discussed in a quite general manner. Few experience based "food-for-thought" type of ideas are presented instead of exhaustive guidelines.

The section is divided in separate subsections covering different aspects of writing. First, the motivation to write is discussed so that it would not be considered just as a necessary evil. Then, the content of documents is addressed, subdivided into text, abbreviations, and figures and tables. Additionally, the concept of visual bookmarks is introduced and document scope is discussed. In the end, the documenting part is concluded in its own subsection.

6.3.1 Motivation to write

The reasons for writing documentation are clear, at least. They should be cleared also for the writer so that he/she finds producing good quality documentations useful and important. In other words, the writer should consider documentation as actual information sharing and not just bureaucratic burden.

Motivational issues are rather complicated psychological phenomena and, thus, out of the scope of this thesis. However, their importance should be acknowledged and some strategic moves done to change the situation. As a rule of thumb one could think that the harder people are pushed the more they will push back. At least if they do not understand or agree with the reasons of pushing. Also making rules that cover everything is not possible.

6.3.2 Text

It has been shown [26] that the style of written text affects how easily the reader can receive the message. This style includes everything: chapter division, used words, and sentence structure, just to mention few. The following quotations give an example of a typical piece of scientific text and then another piece that contains the same information but is formulated in a better way.

Original text [12]:

> Review of each center's progress in recruitment is important to ensure that the cost involved in maintaining each center's participation is worthwhile.

Improved text:

> Review the recruitment process to ensure cost-effectiveness.

As one can easily see, shorter sentence and the use of active voice makes it clearer. In the following list some good rules of thumb are presented. Some of the rules are based on experience some are extracted from [26]:

- Avoid long sentences
- Avoid using perplexing (=complicated) words
- Avoid using unnecessary acronyms and abbreviations
- Avoid using passive
- Place the already known in the beginning of the document/section and the new in the end of the document/section
- The verb should be as soon as possible after the grammatical subject

Another good documentation habit is to write summaries. The essential information is often scattered in a large amount of text, thus, summaries enable fast information acquisition when there is no need for deeper understanding or just quick recap if the reader is already familiar with the material. Numbered lists, bullet lists or tables usually fit this purpose well.

In conclusion, engineers must to pay attention and focus in order to write good text. Of course, it can become almost automatic with experience but that needs a lot of work. One important aspect of writing is that people are blind for their own mistakes. The writing skills are best improved with writing a lot and getting good, constructive feedback.

Different quality processes include documentation reviews. However, the focus of these reviews might be more on the technical content and details but not that much on the quality of the text. Hence, the reviews ensure that all the needed information is in the documents somehow but not that it is communicated in a good manner. Thus, additional informal style reviews are advised.

6.3.3 Abbreviations and acronyms

Let us start with fictional yet representative example:

> PWAA is very common in FTC. Occasionally PWAA are about the IU factor. In addition to IU practical problems include the fact that TLAs are HP and DG, especially apparent in TT and create TCs. It does not help if one uses ETLAs. In conclusion, PWAA is common in FTC and it appears in the form of HP, DG, TC, and IU especially in the case of TT...[1]

Abbreviations are used to avoid repeatedly writing long words or combinations of words. Sometimes they may improve readability. However, quite often they don't since understanding abbreviations and, especially, acronyms is not straightforward. It takes time from the reader to learn what the seemingly random combinations of letters mean. This reduces the reading speed, causes confusion, and they get easily forgotten. Additionally, they make the text unaesthetic by creating clutter. Not to forget, that acronyms are often hard to pronounce. This is highlighted in the modern day multinational working environment where accents are diverse and teleconferences frequent.

The same acronym can mean multiple things in the different context. This complicates the search for the correct meaning, especially if one is not familiar with the context. For example, in telecommunication PA is used for power amplifier. However, it can also mean personal assistant, public address system, synthetic polyamides, or Pamela Anderson, just to mention a few [63].

[1]Acronym explanations for readers not familiar with the topic: PWAA (Problem with Abbreviations and Acronyms), FTC (Field of Telecommunication), IU (Impossible to Understand), TLA (Three Letter Acronym), HP (Hard to Pronounce), DG (Difficult to Guess), TT (Phone Conversation, before telephone talk), TC (Text Clutter), ETLA (Extended Three Letter Acronym).

Nevertheless, some terms are really long and writing them all the time is not really effective. This means acronyms and abbreviations cannot be discarded completely. Consideration should be used and if a shortening of the term is really necessary and what are different options. The abbreviations are easier to understand if more letters from the source are included. For example, *downlink decimation filter* could be abbreviated to *DDF*. A better option would be *dl_dec_filt*, since dl is already established abbreviation for downlink. When used in speech, one could simply say the whole word. Saying the whole word is further encouraged because in English the pronunciation of letters depends heavily on the surrounding letters. For example in the case of *decimation*, the "c" is pronounced in the original word as "s" but in abbreviation *dec* as "k". Thus, if the abbreviation is clear in written form it might not be clear anymore when said out loud.

If abbreviations are necessary, established abbreviations are a good choice. Of course, there can be only a limited number of these and one must really carefully consider the audience and think if the abbreviations are also clear for them. Sometimes it might be a good idea to create a commonly agreed list of abbreviations. For example, in signal naming *address* could always be shortened as *addr*, *channel* as *ch*, *read* as *rd* and so on. The list of agreed abbreviations must be easily available for all the users.

Figures, tables, and signal names in code are exceptions for the rule to vigorously avoid abbreviations. There is simply not always enough space use full terms. Nevertheless, the explanation should included somewhere, for example in figure caption or legend.

If abbreviations and acronyms are used, they should be always explained at least when first introduced. This is self evident in scientific articles and different types of theses. However, this is often forgotten in technical documents and presentations, and code files.

6.3.4 Figures and tables

Good figures make documents much more understandable [48]. For example, complicated digital system's structure and timing can be presented much better using visual information than text. Additionally, figures can be used as visual bookmarks as mentioned in Subsection 6.3.2. Although figures can clarify things, they can also make things more complicated and even misleading.

Figure 6.2 shows a good example that captures the structure and functionality of a system that executes a mathematical function. The system is completely imaginary and probably not realistic.

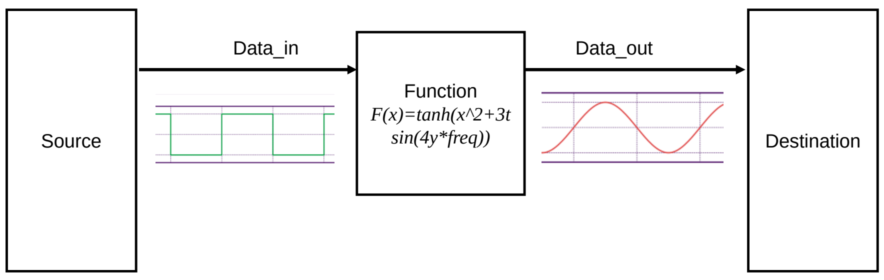


Figure 6.2 *A visualization of an imaginary system. Note that the figure doesn't only show the structure but also functionalty of the system*

Making high quality figures is time consuming and the figures have to be updated and maintained. However, the figure quality can be improved already by following some simple guidelines. Additionally, one must remember that the factors that make high quality pictures is also very situation dependent. Also the importance of feedback cannot be underestimated.

The following list itemizes some common rules for drawing good figures:

- Follow western reading order: left-right and top-down
- Do not use red for highlighting important details, use it for showing undesired aspects
- Show only the essential and situation appropriate information
- Remember to visualize functionality, not only structure

Tables are useful for capturing textual and numerical information. Table 6.1 provides some simple guidelines that clarify the tables considerably [53, pp. 26–27]. In the upper section there are more generic guidelines and in the lower section some advice how to format the table.

As always, reviews can be used in improving table and figure quality. It does not take too

Table 6.1 *Basic table guidelines*

Guideline	Example#1	Example#2
Ordering	First things first	-
Categories	Split into e.g. 3-5 lines	-
Link to text	Reference and exaplantion	-
Number to right	w/ thousand separators	5 175
Text to the left	Like this	-
Nothing is centered	-	-
Bounding lines	Not around every cell	-

much time to show one's figure to a colleague and ask how clear it is for he/she and then think about improvements.

6.3.5 Visual bookmarks

When people read documents, they quite often have to return to a certain spot. Thus, the document should include visual bookmarks to help browsing the text in a fast pace. Visual bookmarks comprehend something that breaks the monotonic text blocks with something different, for example, a figure, a separate list of things, a code block, or a table. This precipitates the finding of information since people seldom remember page numbers but often remember things such as "It's was right next to that picture". Experience has shown that every second or third page is a good pace for the visual bookmarks.

The idea of visual bookmarks can be extended from documenting to, for example, Powerpoint presentations. Quite often companies and institutions have a template that is used in every presentation. When people then need to find a certain presentation it might take time since one has to read the titles in order to know what it is about. If there was a picture in the title slide, the correct slide could be found much faster, since people tend to remember to figures. Figure 6.3 clarifies this idea. It is rather extreme and a lot can be achieved already by simply adding some related figure next to the title.

6.3.6 Documentation scope

When writing documentation one must always remember the target audience. Describing everything in the document does not serve any purpose; the reader can be expected to check some details from the code. For example, the port widths are in this category.

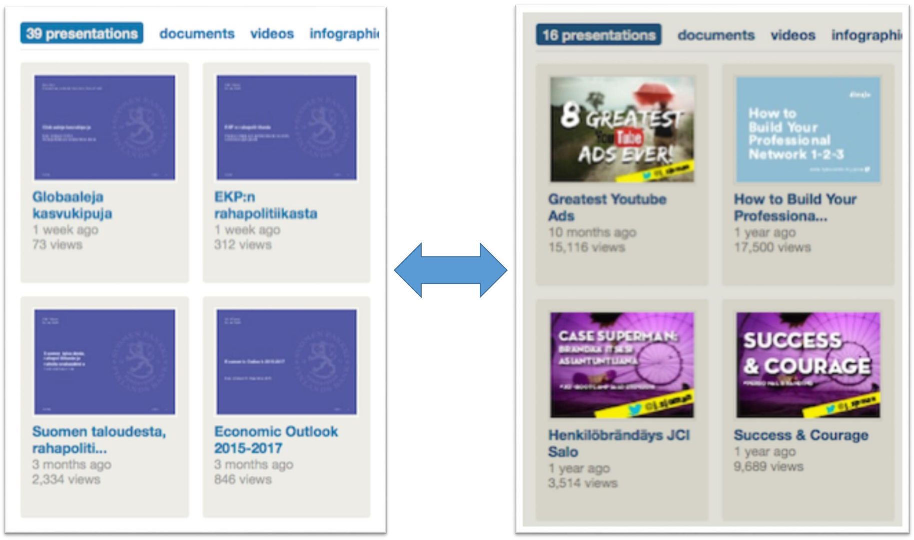


Figure *6.3 An example of using visual bookmarking with Powerpoint slides. On the left the same monotonic template is always used which makes it hard to identify individual presentations quickly. On the right more elaborate backround is added and the presentations can be identified by a quick look. [32]*

Also the level on which things are discussed should be adjusted. If information is too specific, it is possible that most of the readers cannot extract anything from the document. On the other hand, if the documentation is completely on Wikipedia level, why is the document existing in the first place. One must also remember, that low level details are often troublesome to keep updated.

When it comes to documentation of the same topic but for different audience, one should decide whether to reduce the amount of files by making documents more comprehensive or enhance the finding of relevant information by making more but very specialized documents. This issue can also be solved partially with software. For example, FrameMaker has a conditional text option which enabled to writer to write everything in one document and then easily choose which parts to forward to which person [1].

6.3.7 Conclusion on documenting

There are some things to keep in mind when writing documents:

- The text is written for the reader

- The reader is likely to know less about the subject than the writer
- Do not overestimate the reader's background knowledge
- Do not underestimate the reader's ability to absorb new information
- Remember to use visual bookmarks
- Write summaries
- Unexplained abbreviations and acronyms complicate the text unnecessarily
- Good figures and tables make documents much clearer
- Naming in the text, figures, tables and design itself needs to be consistent
- Feedback is important part of improving all aspects of documentation

In the end, the question is not only about designer skills but also about time allocations. If the documents need to be written in a hurry, the quality goes down accordingly. Sometimes time is given but in the same time new design tasks are pushed in and designer might prioritize the design.

When thinking about documentation, it is strange that in the era of Youtube and Facebook the official information sharing is still based on printable documents. This was understandable 50 years ago but concurrently the technology enables much more intuitive and more user friendly solutions, such as, html-documenting or videos. Maybe something better could be developed from this point of view.

6.4 Code clarity

This section is divided in 4 subsections: naming and structure, improvement practices, "Hello world" examples and conclusions. Naming and structure focuses on explaining how clear VHDL code could be written. Improvement practices is more about how to implement the clear structures in daily work. "Hello world" examples introduces a method for facilitating easy re-use and other kind of design transfers. In the end, conclusion gathers most relevant information together.

6.4.1 Naming and structure

High quality code should be self explaining and even self documenting [17, 34]. In VHDL, this is achieved with careful division of entities and processes, proper naming, and good commenting. The naming should be descriptive and consistent. Abbreviations are usually needed. They should follow the commonly agreed naming conventions and be explained. Capital letters should be avoided at least in signal names but they might be useful with generics. [37]

Readable code is not only about the naming of different pieces but also about the visual appearance of the code. Indentations make code clearer. The assignments <=, =>, and other operators should be aligned. Also the length of one line should not be too long. [37] This is especially easily achieved with VHDL because dividing one long line to two shorter ones is rather easy.

The code should also be structured in a clear way [34]. For example, it affects clarity if some condition is implemented with separate if-clauses or with if-elsif-else structure. The following code example elaborates the above mentioned case. One must remember that the coding style cannot be chosen completely freely because it might affect the later implementation of the code.

```
a) Four separate if-clauses (not good)

if (MODE = 1 and CHOICE = 0) then
   next_state_r <= wait;
end if;
if (MODE = 1 and CHOICE = 1) then
   next_state_r <= run;
end if;
if (MODE = 2 and CHOICE = 0) then
   next_state_r <= run;
end if;
if (MODE = 2 and CHOICE = 1) then
   next_state_r <= wait;
end if;

b) Mutually exclusive if-else branches (good)

if (MODE = 1) then
   if (CHOICE = 0) then
    next_state_r <= wait;
```

```
 else
    next_state_r <= run;
 end if;
else
-- MODE = 2
   if (CHOICE = 0) then
    next_state_r <= run;
 else
    next_state_r <= wait;
 end if;
end if;
```

Further improvements to code readability can be achieved by separating functionally different code segments within on file from each other by empty lines. Additionally, instantiations can be organized in the same order in code as they are in block diagrams. It is also important to avoid including too many entities or functions in the same file.

6.4.2 Improvement practices

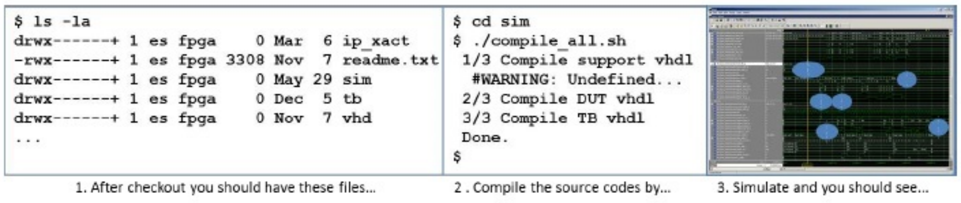
Linter tools are also handy when it comes to checking that coding style is good. Robert Colwel described a situation in his book [13, p. 42] where they were trying to decide common coding style together with many experienced programmers. After following the fighting for some time, Colwell realized that the agreement cannot be reached. He started thinking about the solution and, at some point, he figured out that there might be one. He reasoned, that during their studying and working life, computer engineers are conditioned to take feedback from computer without getting angry about it. This is mostly due to the endless compiler errors. Thus, Colwell created his own linter tool that actually implemented his vision of good coding. The other engineers accepted the solution without resistance. Apparently, when the criticism comes from computer, it is easier not to take personally.

If code quality needs to be improved, reviews are a very useful tool. In addition to providing a valuable way to get feedback, code reviews often reveal bugs in the design. It is of paramount importance that the reviews are done already in early design phase, so that the code can actually be changed.

6.4.3 "Hello world" examples

The easiness of IP reuse can be accomplished by using "Hello world" examples. It refers to an easy startup script that, for example, runs a simple test case. Also some low level documentation is provided to show how simulation should be running. With this kind of example the user knows that he/she has all the files and the basic settings are correct. When the user then starts to do modifications the inevitable debugging becomes easier with the knowledge that basic things are working.

Figure 6.4 shows an example of this kind of easy startup. As one can see, very little space and user time is used with this method to give a lot of information and clarity.



```
$ ls -la
drwx------+ 1 es fpga    0 Mar  6 ip_xact
-rwx------+ 1 es fpga 3308 Nov  7 readme.txt
drwx------+ 1 es fpga    0 May 29 sim
drwx------+ 1 es fpga    0 Dec  5 tb
drwx------+ 1 es fpga    0 Nov  7 vhd
...
```

1. After checkout you should have these files...

```
$ cd sim
$ ./compile_all.sh
 1/3 Compile support vhdl
  #WARNING: Undefined...
 2/3 Compile DUT vhdl
 3/3 Compile TB vhdl
 Done.
$
```

2 . Compile the source codes by...

3. Simulate and you should see...

Figure 6.4 *One possible way to do easy "Hello world" startup for an IP [45].*

"Hello world" examples should be done in a very late phase of design when no more changes are expected. When used correctly it can even substitute a part of official documentation.

6.4.4 Conclusions of code clarity

The following list gathers thing so be considered with clear code:

- Pay attention to the division of entities and processes
- Name everything wisely
- Remember to comment
- Code structure matters

All in all, as a rule of thumb, company guidelines should be followed. If company guidelines are just not good one should try to change the situation or think if his/ hers ideas are actually any better. There is a difference between personal opinions and facts that actually have proven effect on code clarity. However, the amount of guidelines has to be small enough that following them does not get too complicated. *"Don't sweat on the small stuff"*, as they say.

6.5 Scripts

Improving the scripts is an effort that is likely to pay back. Most of them are written in-house which enables straightforward deployment of improvements compared to, for example, proprietary tools. Good scripts have a lot in common with good code, for example commenting and variable names. In this section the focus is on script specific aspects, such as path concatenations and error messages.

6.5.1 Different scopes of scripts

Scripts can be thought to be used in two different purposes: automation and abstraction. Automation means that one uses scripts to do a series of simple tasks one could do also by hand separately. The abstraction means that the script is used to allow someone perform a task one would not be able to perform oneself. In this case, the script is written by someone else and it, in a way, creates a level of abstraction from the user point of view. However, also other classifications are existent. For example, one could think that automation type of script doesn't use parameters whereas abstracting one uses. However, the first system is utilized in this thesis.

Scripts have naturally quite different requirements depending on their purpose and lifespan. For example the commenting is not that crucial, if a script is most probably going to be used only in a certain project by a certain individual. However, if the scripts is redistributed it should be modified to suit this new scope. Additionally, if the use seems to extend beyond a few designers, version control utilization should be considered. One must be careful when creating guidelines because designers feel that sharing a script causes them a lot extra work

There is also one aspect against the heavy use of abstraction scripts: the more people use scripts the less they learn to do things themselves. On the other hand, experience has shown that debugging a borrowed script is a great teacher.

6.5.2 Good script writing practices

There are some common guidelines good scripts should follow. For example, a header should be included and not only a pro forma header but a good, updated header. Too often one sees such a header that it is there only because it has to due to guidelines or company policies.

Another good common practice is to make the script give informative prints such as how it is progressing. For example, "Step 4/5, reference files are created.". This makes the almost evident debugging easier. The script could also inform the user with prints if some task has a bit longer execution time. With that print it is much easier to know if the execution is stuck or not.

Sometimes it is useful to have a summarizing print. For example, "2 files were added to IP-XACT.", i.e., to confirm what user expects to happen. If there should have been more than 10 files, this should raise some suspicion. Probably many other types of good prints can be thought of, too. One must also remember that the visual appearance of prints affects the clarity. Thus, indentations and other similar methods should be deployed.

However, should prints help debugging and give the user ideas of the script operation, they can also be useless and even cause disturbance from more important things. At least when the script is used many times a day. To help with this, there could be an additional switch to disable the prints partially or even completely. It depends on the script how many levels of verbosity is appropriate.

Following example illustrates a helpful and informative set of script execution prints:

```
>>./run_test.csh -verbose
1/5 Checking that needed files exist
2/5 Compiling files
   ..
   Compilation completed successfully!
3/5 Running elaboration (this may take several minutes)
   25% of the elaboration completed
   50% of the elaboration completed
   75% of the elaboration completed
   20 files added to filelist.
   Elaboration successfully completed!
```

```
4/5 Saving files...
   Filelist saved!
5/5 Task completed. Exiting.
```

Naturally, a good script has a help print. However, this is usually implemented by using only one switch, for example, –help. Multiple switches could be enabled. In addition, help print is also something that should be kept updated and that should be designed carefully and with time.

Log files are also an option for showing and storing information of script execution. The information stored in log files should be considered carefully. Some tag words, such as warning and error, can be used to facilitate search and the words should be told to the user. The visual appearance of log files is also important.

The experience has shown that many of the issues with scripts are connected to the broken paths. Thus, each script should automatically check if environmental variables are set correctly and that the paths and files actually exist and can be accessed. The existence check could also be done for the referred files. Occasionally, paths are created by concatenation of many parts. This complicates the debugging greatly since it is difficult for human eye to detect errors in very long strings to find the root cause. In addition, the error can be really small, just one letter extra/missing or wrong in wrong case.

Even though the broken paths are almost inevitable, the debugging can be made a lot easier. Should a path be broken or a file non-existing, the script could tell how broken the path is, for example, that 80% of the path is ok. The script could also do guessing about the correct path with autofill feature.

This kind of elaborate scripting might be too complicated for some designers. It is possible that the most difficult parts could be in some common folder and individual user could call them from there. Of course, environment changes would break this and, to fix it, the scripts would need to actually check if the common script exists. Of course, some designers could be just provided with training.

Broken paths are not the only headache, but sometimes a path directs to an existing but old version of the design. This is difficult to detect in the early phase if the user has both versions in his/her personal folder. One way to do check this would be manually renaming the folder containing the old design. After this the path will be broken if it directs to the old version.

Once there was a case, where a script was used to create another script that would package an RTL design with IP-XACT. To collect the file names, it just took the content of the whole folder without checking. As expected, this caused issues: once the folder contained temporary files created by Emacs, ending .vhd~. It took long time to find the root cause. The same could happen with temporary backup files, such as design_old.vhd.

Sometimes scripts assume that the user makes some changes before using the script. These changes could be described already in the header. If some variables need to be set, they could be located already in the very beginning of the script. One could also use environmental variables that are defined outside the script so that the setup can be the same for all the scripts. A good place for defining this kind of variables is, for example, the script that sets the tools and licenses.

Making the script ask for user input is not a good way to increase abstraction. It is true, that then the user doesn't need to modify the script itself. However, such script is difficult to run in automatic batch mode. This greatly limits the use.

6.5.3 Conclusion on scripts

The following list summarizes some good script writing practices. It should be considered especially if the script is used by multiple designers:

- Proper header is very important
- The coding style of scripts is also important
- Prints make following the execution easier
- Prints can be enabled with a switch if they are not always needed
- Log files are also useful
- Paths and file existence should be checked

Additionally, the most commonly used scripts should be regarded as tools in a workshop. This means proper maintenance and some kind of catalogue of them.

An interested reader can find more information of scripts from, for example, [2] or [55].

6.6 Managing the EDA tools

Problems caused by using proprietary tools can be affected by changing the tool. The tool can be changed to another proprietary tool or to an open source alternative. If another proprietary tool is chosen, it is likely that similar problems emerge. Open source solutions are not that straightforward either. First, there is no alternative for all the tools. Second, if an alternative exists, the support might not be on the needed level. However, with the money saved from licenses the company could already finance its own support and development for the open source tools.

The open source tools are not completely free from license issues; the issues are just different from proprietary ones. For example, some licenses do not allow use for commercial purposes and some dictate, that if the code is used as a part in a bigger project, the whole project needs to be complete open, too. [38] This might not be what the company wants.

If the decision is to use proprietary tools, there are things a company can do to enable smooth usage. A real time information of license usage is a good example. If one really needs a license quickly one can ask a colleague if his/hers task can wait and, thus, he/she can free the license. Sometimes the program is just forgotten open. To solve this, automatic emails can be send to users that seem to reserve licenses for a long time.

Controlling the choice of used tool and tool versions is easy, at least in principle. The company could just dictate which tools and versions to use in which project. The tools can be set automatically in some project management software or by a common tool setup script. However, considering the resistance every dictation causes, unnecessarily comprehensive policies should be avoided. Additionally, same people usually work in several projects which complicates this.

Occasionally the problem is not the unwillingness to use correct software or version but rather the difficulties in finding the information. This is related to the common issues in information flow. Moreover, the naming of tool environment scripts might be inconsistent. This complicates the designers search for the correct version.

Companies should really think carefully which tools to buy. With some tools the problems are clear but benefits questionable. Also people that work with tool everyday should be asked about opinions. The marketing personnel from the vendors are good and decision makers busy and distant from floor level design; there is usually 1–2 organization levels between the important decision makers and the actual users that understand the best what is needed. This is a bad combination.

7. INTERVIEW RESULTS AND ACTION POINTS

This chapter describes the interviews. They were conducted in order to find out more about the issues at NSN and also to provide some idea of their impact to actual work. The first sections describes the interviews. After that the results are discussed. The last section focuses on the possible actions.

7.1 Conducting interviews

Seven IP developers were interviewed based on volunteering, thus, the sample group might not be statistically representative. Yet, the interviews were also about figuring out possible issues and, thus, it was good to let people speak if they wanted to. The interviewed group contained both male and female members and three different nationalities. The work experience in the field ranged from 1.5 to 25 years.

The interviews were quite loosely structured and were based on informal discussion. The author wrote down things that came up during the discussion and sometimes asked additional, elaborating questions. Thus, the interviews were not always identical. The duration of a single interview was between 30 minutes and 1 hour. In general, the more experienced the interviewee the longer the interview.

The topics that were discussed in all the interviews were: inefficient communication, finding correct documents, version control issues, unclear documentation, unclear code, tool related problems, IT related issues (computational platforms and IT support), and script re-use related problems. The interviewees were also asked to rate the impact in scale 1–4 where 1 means that there is no impact at all and 4 that there is a big impact.

Before the interviews, a summarizing presentation was delivered about the most common issues. The presentation raised some discussion and the results of the discussion were taken into account when designing the interview topics. Most of the interviewed people participated in this presentation.

Not all the gathered information is published here, since it contains information that is company confidential. However, the results in the following section contains most of the most essential aspects and notations.

7.2 Results

Figure 7.1 represents the result of the interviews. It is derived directly from the interview data. The maximum score only came up twice in the scoring; once for inefficient communication and once for IT related issues. The minimum score come up more frequently: the only two items that did not get one at all were IT related issues and unclear documentation.

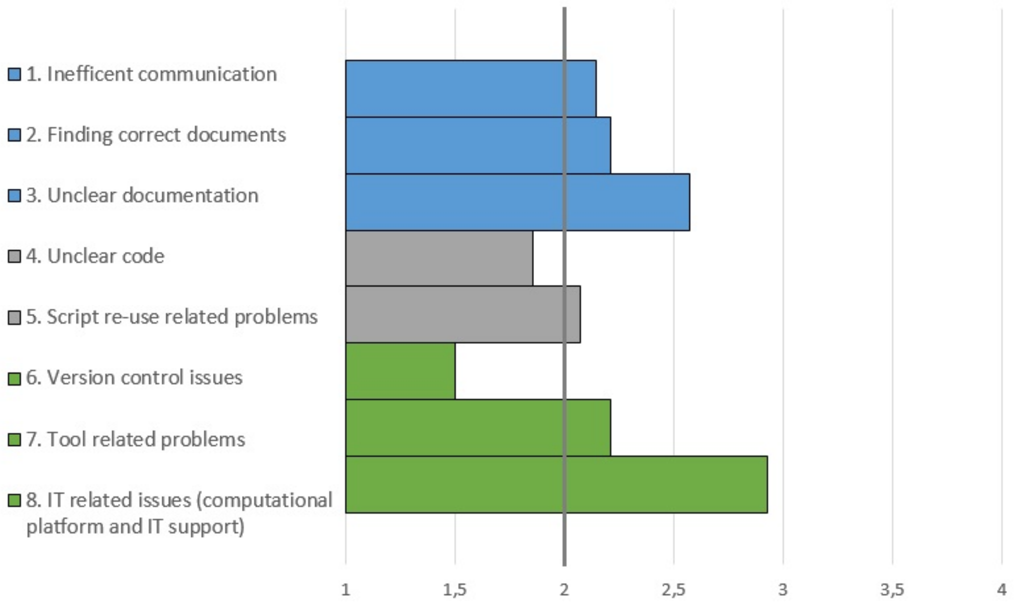


Figure 7.1 *Reported average impact of each issue. Colors denote 3 categories that can be seen partially interdependant.*

The interviewee also had a lot of comments for different topics. The following part will give short summaries of the comments.

Inefficient communication: Person-to-person communication is working well and it is easy to find the correct person for asking a specific question. However, some of the interviewees felt that the communication overhead was quite big and people should spend more time reading documents instead of asking. On the other hand, it was also mentioned that some understand things generally better through discussions than reading. The impact of this aspect was evaluated to be average.

Finding correct documents: Finding documents had an average impact score. Documents were usually found by asking others for the link or following links from other documents. This means that the search of documents increases the communication overhead. However, people don't need to look for new documents too often and saving links of the needed documents was found to be a good way to handle this issue.

Unclear documentation: Unclear documentation was the second biggest issue. It was felt that documents were mostly done for process purposes and they were not clear; they didn't contain relevant information in simple format. However, many of the interviewees felt that the situation is currently being paid attention and that it is getting better all the time.

Unclear code (VHDL): Issues with VHDL clarity did not have much impact. There were some complaints about the lack of clarity in computer generated codes. Actually, during the writing of this thesis before the interviews, the NSN coding guidelines were updated and clarified which got a lot of positive feedback and might have an effect for the low impact score.

Script re-use related problems: Script issues scored average and higher than code clarity. Within the scripts a clear division could be seen based on the experience level of the interviewee. The people with long experience mostly wrote the scripts themselves and did not have much issues with re-use. The new people, on the other hand, were asking for script training or guidelines and had problems with re-use. It was also suggested that scripts could be gathered in a centralized manner and to have more commenting and such things to enhance re-use.

Version control issues (fundamental usability): Version control was the most problem free area. Three issues were specifically named: getting access rights quickly, managing the versions of supporting common blocks or verification environments, and changes made to tagged releases.

Tool related problems: Tool related problems scored average. The most important single issue with tools was the lack of simple instructions. The user guides provided by EDA companies are usually really comprehensive whereas the everyday use quite limited. Also the need for more trainings came out as a conclusion.

IT related issues (computational platforms and IT support): IT related issues had the highest score. Because of the heavy computation included in the IP design, the impact of even small issues is big and might prevent working entirely, for example server crashing momentarily. It was felt that the issues were not solved quickly enough and that the used ticket system was complicated.

7.3 Conclusions and possible actions

Corrective actions should focus on the items with most impact: IT related issues, unclear documentation, finding correct documents or tool related problems. It is also worth considering, that even though the script problems did not score high, the need for script training and guidelines came up in many of the interviews.

The base for the actions described here can be found from the Chapter 6. When thinking about possibilities, the author has considered especially two things: the ease of implementation and other ongoing actions at NSN. The ease means that no big process level changes are needed but that solutions concentrate on actions that can take place on the team level. The other ongoing actions mean different kind of projects that aim to same kind of improvements as this thesis. Some of the projects have started during the making of this thesis. This thesis could be used as a reference to further highlight the importance of these actions.

The two biggest issues, IT and unclear documentation, have already been widely noticed and many different actions are ongoing to make some corrections. Planning some new actions while the effects of previous ones are not clear would be rather useless. Thus, the focus should be placed on finding correct documents, tool related problems and inefficient communication.

Finding correct files: One rather straightforward action for easing the search of documents could be some kind of link library. There are several different link libraries at the moment for different projects but it could be useful to have one library that contains links and descriptions of different libraries. Also links for not-design-related documents, such as tool instructions and different guidelines could be added. The library should be regularly updated and some easy broken link reporting system should be added. It is possible that such library already exists and in that case it should maybe be more advertised.

Tool related problems: Most of the tool related problems were connected with bad or missing low level instructions and lack of training. This means that instructions should be written and trainings held. More research would be needed to get prioritized list of different tools requiring this. When writing the instructions and preparing the trainings one should consider carefully the guidelines for clear documentation. Bad training is as good as no training at all.

Inefficient communication: With communication the highlighted area was communication overhead. It could be improve with better processes and using better development practices. However, those are out of the scope of this thesis. The author's suggestion is to wait until some results from better documentation and file finding are available and see if these affect communication overhead. After that it can be decided if more actions are needed.

Scripts: Due to the demand that came up in the interviews, some script training, template and guidelines could be developed.

8. CONCLUSIONS

During the literature study, the following common sources for issues were identified: inefficient communication, finding correct files/documents, version control usage, unclear documentation, unclear code, EDA tool usage and IT in general, and scripting. Based on these results 7 employees of NSN were interviewed. The interviews showed that the biggest issues at NSN are related to IT support, document clarity, finding correct documents and EDA tool usage.

Since there are a lot of ongoing actions at NSN related to IT and documentation clarity, the author decided to focus on improving document search and EDA tool usage. The recommended actions are creating a functional library for document links and providing training and better instructions for EDA tools. The need for script training and guidelines was mentioned in most of the interviews and, thus, they are also recommended to be arranged. Additionally, this kind of interviews should be organized periodically to follow how improvements are working and to see if new issues are arising.

BIBLIOGRAPHY

[1] Adobe, "Adobe framemaker homepage," 2016. [Online]. Available: https://www.adobe.com/products/framemaker.html

[2] I. D. Allen, "Unix/linux shell script programming conventions and style," 2013. [Online]. Available: http://teaching.idallen.com/cst8177/13w/notes/000_script_style.html

[3] Altera, "Altera ip components," 2016. [Online]. Available: https://www.altera.com/products/intellectual-property/overview.html

[4] ALTERA, "Nios ii processor overview," 2016. [Online]. Available: https://www.altera.com/products/processors/overview.html

[5] American Psychological Association, "Multitasking: Switching cost," 2006. [Online]. Available: http://www.apa.org/research/action/multitask.aspx

[6] ARM, "Amba specification," 2016. [Online]. Available: http://www.arm.com/products/system-ip/amba-specifications.php

[7] Atmel, "Atmega328p datasheet," 2015. [Online]. Available: http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-328p_datasheet_complete.pdf

[8] B. Bailey and M. Grant, *ESL Models and their Application*, 1st ed. Springer US, 2010.

[9] J. Blau, "Talk is cheap," *Spectrum, IEEE*, vol. 43, no. 10, pp. 14–15, 2006.

[10] B. W. Boehm, "Improving software productivity," *Computer*, vol. 20, pp. 43–57, 1987.

[11] M. Casale-Rossi, "The heritage of mead & conway: What has remained the same, what has changed, what was missed, what lies ahead," *Proceedings of the IEEE*, vol. 102, pp. 114–119, 2014.

[12] K. Cobb, "Scientific writing lecure 1 slides," 2008. [Online]. Available: web.stanford.edu/~kcobb/writing/lecture1.ppt

[13] R. P. Colwell, *The Pentium Chronicles*. John Wiley & Sons, Inc, 2006.

[14] P. Coussy, M. Meredith, D. D. Gajski, and A. Takach, “An introduction to high-level synthesis,” *IEEE Design & Test of Computers*, no. July, pp. 8–17, 2009.

[15] CPU World, “AMD XBox One specifications.” [Online]. Available: http://www.cpu-world.com/CPUs/Jaguar/AMD-XBox%20One%20APU.html

[16] R. A. Crabtree, N. K. Baid, and M. S. Fox, “Where design engineers spend/waste their time,” Department of Industrial Engineering, University of Toronto, Tech. Rep., 1993.

[17] Cunningham & Cunningham, Inc, “Self documenting code,” 2014. [Online]. Available: http://c2.com/cgi/wiki?SelfDocumentingCode

[18] A. D. day, “Psoc.” [Online]. Available: https://armdeveloperday3rd.files.wordpress.com/2012/11/psoc.png

[19] V. Driessen, “Vizualisation of version control,” 2010. [Online]. Available: http://nvie.com/posts/a-successful-git-branching-model/

[20] M. Ernst, “Version control concepts and best practices,” 2016. [Online]. Available: https://homes.cs.washington.edu/~mernst/advice/version-control.html

[21] D. FIlippov, “Computer user,” 2016. [Online]. Available: https://blog.jetbrains.com/pycharm/2013/06/vim-as-a-python-ide-or-python-ide-as-vim/

[22] M. Fowler, “Cannotmeasureproductivity,” 2003. [Online]. Available: http://martinfowler.com/bliki/CannotMeasureProductivity.html

[23] D. D. Gajski, A. C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, and P. Briacaud, “Essential issues for ip reuse,” *ASP DAC*, no. January, pp. 37–42, 2000.

[24] A. P. Goldberg, “Producing production quality software,” 2005. [Online]. Available: http://www.cs.nyu.edu/artg/Producing_Production_Quality_Software/Fall2005/lectures/Lecture-14-GroupPractices.pdf

[25] J. R. Goldschmied, P. Cheng, K. Kemp, L. Caccamo, J. Roberts, and P. J. Deldin, “Napping to modulate frustration and impulsivity: A pilot study,” *Personal and Individual Differences*, pp. 164–167, 2015.

[26] G. Gopen and J. Swan, “The science of scientific writing,” *American Scientist*, 1990.

[27] A. Hesseldahl, "System on chip," 2012. [Online]. Available: http://allthingsd.com/20120102/global-chip-sales-down-on-thailand-flooding/

[28] ISO, "Iso-9000 standard," 2015. [Online]. Available: http://www.iso.org/iso/home/standards/management-standards/iso_9000.htm

[29] ITRS, "Itrs report 2005, design," ITRS, Tech. Rep., 2005. [Online]. Available: http://www.itrs2.net/itrs-reports.html

[30] D. James and C. Young, "Xbox one soc," 2013. [Online]. Available: http://www.chipworks.com/about-chipworks/overview/blog/inside-xbox-one

[31] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenence tasks," *IEEE Transactions on Software Engineering*, vol. 32, pp. 971–987, 2006.

[32] K. Kortesuo, "Organisaatio, anna väljyyttä otsikkodioihin," 2016, in Finnish. [Online]. Available: http://eioototta.fi/kategoria/diaesitykset/

[33] S. Y. Lee and J. L. Brand, "Effects of control over office workspace on perceptions of the work environment and work outcomes," *Journal of Environmental Psychology*, vol. 25, pp. 323–333, 2005.

[34] R. C. Martin, "Uncle bob's clean code tutoria videos part 1." [Online]. Available: [Nokiainternal]

[35] R. Mascitelli, *Mastering lean product development*, 1st ed. Technology Perspectives, 2011.

[36] B. Mehta, "Adder transistor level layout," 2001. [Online]. Available: http://pages.cs.wisc.edu/~bsmehta/555/project/layout_fulladder.png

[37] NOKIA, "Vhdl coding guidelines," 2016. [Online]. Available: [Nokiainternal]

[38] Open Source Initiative, "Licenses & standards," 2016. [Online]. Available: https://opensource.org/licenses

[39] OpenCores.org, "Opencores ip components," 2016. [Online]. Available: http://opencores.org

[40] J. H. Pejtersen, H. Feveile, K. B. Christensen, and H. Burr, “Sickness absence associated with shared and open-plan offices – a national cross sectional questionnaire survey,” *Scandinavia jounal of work, environment & health*, vol. 37, pp. 376–382, 2011.

[41] J. L. Popyack, “Object oriented programming: Advantages of oop,” 2015. [Online]. Available: https://www.cs.drexel.edu/~introcs/Fa15/notes/06.1_OOP/Advantages.html?CurrentSlide=3

[42] L. Prechelt, “An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program,” Fakultät für Informatik Universtät Karslruhe, Germany, Tech. Rep., 2000.

[43] A. Raschka, “Pic18f8720 microcontroller,” 2006. [Online]. Available: https://commons.wikimedia.org/wiki/File:PIC18F8720.jpg

[44] Raymond, “Modelsim wave view,” 2010. [Online]. Available: https://fftonde3.files.wordpress.com/2010/07/wave_20100713_result.jpg

[45] E. Salminen and T. D. Hämäläinen, “Teaching system-on-chip design with fpgas,” fPGAWorld 13, September 10-12, Copenhagen, and Stockholm.

[46] I. Sameli, “Intel 8742,” 2003. [Online]. Available: http://www.flickr.com/photos/biwook/153056995/

[47] A. Sangiovanni-Vincetelli, “Quo vadis, sld? reasoning about the trends and challenges of system level design,” *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[48] SHIFT, “Studies confirm the power of visuals in elearning,” 2014. [Online]. Available: http://info.shiftelearning.com/blog/bid/350326/Studies-Confirm-the-Power-of-Visuals-in-eLearning

[49] Stackoverflow, “Version control for docx and pdf,” 2011. [Online]. Available: https://stackoverflow.com/questions/3298525/version-control-for-docx-and-pdf

[50] W. F. Stewart, J. A. Ricci, E. Chee, D. Morganstein, and R. Lipton, “Lost productive time and cost due to common pain conditions in the us workforce,” *The journal of the American Medical Association*, vol. 290, pp. 2443–2454, 2003.

[51] “Kactus2 memory visualisation,” Tampere University of Technology, Department of Pervasive Computing. [Online]. Available: http://funbase.cs.tut.fi/#kactus2

[52] "Hardware design view of kactus2," Tampere University of Technology, Department of Pervasive Computing, 2016. [Online]. Available: http://sourceforge.net/projects/kactus2/

[53] Tampereen teknillinen yliopisto, "Opinnäytetyön kirjoittaminen tampereen teknillisessä yliopistossa," 2014.

[54] S. Torkkola, *Lean asiantuntijatyön johtamisessa*. Talentum, 2015.

[55] Ubuntu, "Ubuntu manuals: checkbashisms." [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/en/man1/checkbashisms.1.html

[56] Vincit, "Vincitin blogi." [Online]. Available: https://www.vincit.fi/blog/

[57] S. Wagner and M. Ruhe, "A systematic review of productivity factors in software development," State Key Laboratory of Computer Science, Institute of Software, TU München, Tech. Rep., 2008.

[58] R. Waller, "What makes a good document?" University of Reading, United Kingdom, Tech. Rep., 2011.

[59] R. Wetzel, "Kanban board system," 2013. [Online]. Available: http://sites.psu.edu/ryanwetzelleadership/2013/06/13/leadership-communication-strategy-kanban-board/

[60] S. A. Wheelan, "Group size, group development, and group productivity," *Small Group Research*, vol. 40, pp. 247–262, 2009.

[61] Wikipedia, "Anna karenina principle," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Anna_Karenina_principle

[62] ——, "High-level synthesis," 2016. [Online]. Available: https://en.wikipedia.org/wiki/High-level_synthesis

[63] ——, "Pa," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Pa

[64] ——, "Vendor lock-in," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Vendor_lock-in

[65] Wikiquote, "Martin fowler," 2014. [Online]. Available: https://en.wikiquote.org/wiki/Martin_Fowler

[66] R. Wilson, “Socs: Ip is the new abstraction,” 2011. [Online]. Available: http://www.edn.com/electronics-news/4368295/SOCs-IP-is-the-new-abstraction

[67] W. Wolf, A. A. Jerraya, and G. Martin, “Multiprocessor system-on-chip (mpsoc) technology,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systmes*, vol. 27, no. 10, pp. 1701–1713, 2008.

[68] W. Zhang, “Adder gate level schematic,” 2001. [Online]. Available: http://esd.cs.ucr.edu/labs/tutorial/adder_sch.jpg