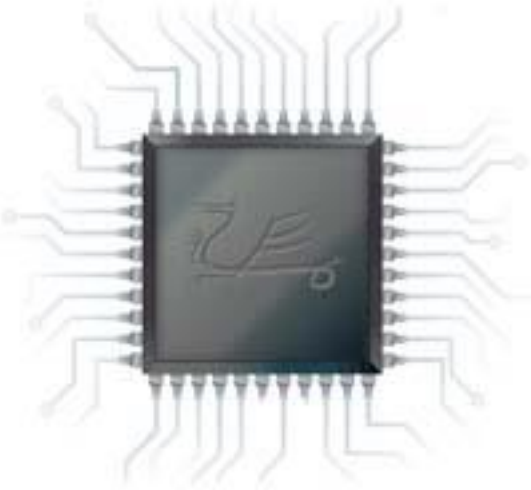
Documentation: Preserving the value of your logic IP



Simon Southwell

December 2021

Introduction

So, you've just been given a new assignment, or maybe just started at a new company, and your new objective is to add a feature to some pre-existing logic IP. The specification for the new feature seems straight forward enough, so you ask to see the documentation and you check-out the source code. The previous engineer has just left the company, and what you find is that the document has a couple of sentences for each block in the hierarchy, basically stating what is self-evident from the module names, the top-level block diagram is just blocks instantiated in the top level single lines between them, and the code has barely any comments. Of those that do exist, they state the obvious: 'add 1 to the register'. So you go back to you line manager and say, "you know I estimated a month for this work, well...".

I lost count of the number of times this has happened to me or seen it for others. As a direct result of this sort of thing, money is spent reverse engineering IP that was, previously, thought to be well understood. It may even delay a project with more incalculable costs, both financial and in reputation. It very well maybe that the IP is a well structured, robust piece of work, which will help with reverse engineering the IP but, none-the-less, unnecessary effort is now required. In the worst-case scenario, it may end up being quicker to re-implement the design than reverse engineer it. This problem is way too common.

I have, over the last few decades, gained a reputation for being good at documenting my work, and an exception to the rule amongst engineers. In this article I want to share some of my thoughts as why that might be, and why I think I am not exceptional, except in my motivation to do this aspect of my responsibilities to the same high standard as all the other aspects. I want to look only at the internal engineering documentation of an R&D department. I am not talking about documentation for external use, usually written by Technical Authors, who would normally produce excellent documentation—and are always worthwhile speaking with on constructing your own documents if they have some spare time. I want to focus on the documents for the IP that are expected to be produced by the engineers as part of a project development...the "internal maintenance specifications", or whatever they are called at your locale. I will focus on logic IP documentation for the examples, as that is what I have mostly been working on throughout my career, but the concepts should migrate to other disciplines (and I have been an embedded software engineer).

So why is this so bad? I think partly because engineers aren't so good at documentation (or think they aren't), so are not interested and would rather work on something else. But I also think that management do not always prioritise this aspect of engineering, as the consequence of bad documentation is not immediately

obvious for a given project. It's quite commonly acknowledged that "we need documentation", but this then pushes documentation towards the end of the project. This usually fails because the reasons for many design decisions have been forgotten, there is pressure to finish off the current project and motivation for starting up the next one. The documentation then suffers from those aspects we've already discussed. Can we do something about it? I think so, and I want to outline some lessons I've learned that not only help get the right sort of documentation, but actually help engineers write it, and help managers keep track of project implementation intent, progress, and match to requirements (which may change during development).

When to Start

I mentioned that IP documentation is often pushed to the back of the project and then rushed as a consequence. It should start before a single line of code is written—after the specification is complete enough to kick off development. From the specifications, some initial block diagrams can be sketched along with some initial paragraphs written to explain how the design is going to meet the requirements, what major blocks are needed and the flow of information between them. This will all be subject to change and updates, but a core of a document exists up front already, if this is done. In addition, and perhaps more importantly, this initial document can be reviewed to make sure it covers all the requirements, the assumptions made are valid and that the use cases match up to the implementation intent. For me, I have always started a design by sketching block diagrams on paper and, when sufficiently developed with pencil and eraser worn out, transferred to a drawing package of one's choosing (see *Diagrams* section below). Adding some descriptive words can then prepare the document to be reviewed. I have seen on a number of occasions that this has highlighted some issues where, perhaps, changes to requirements have not yet filtered through, or previous architectural decisions make the initial solution invalid and in need of a rethink. This all before any implementation work has started, avoiding any need to backtrack. This needs management backing to work. By allocating time for this initial documentation, having a specific well defined deliverable and date, and by scheduling for review, this will start the documentation ball rolling and a core IP document is well underway.

Layout and Content : Divide and conquer

Given that most engineers aren't motivated to write documents, how can it be made easier. For me, I would usually open a blank document and construct a set of headers as a list of top-level sections required to be written:

- Introduction
- Top Level Design
- Block A
- Block B
- Simulation Verification
- Synthesis
- Platform verification

The example list above if for, say, some logic IP targeting an FPGA. The order of the list is not arbitrary but is chosen to tell 'the story'. In the introduction, a summary of the requirements would be given and a general discussion of how the IP is/will be designed, possibly with any prerequisites and dependencies. This sets the stage for the rest of the document. The Top Level Design section is where the overall design is documented, with a top-level block diagram (and possibly other diagrams), a discussion of the overall operation and reference to which block, or blocks, each requirement is to be implemented. The introduction and top-level design sections would constitute the initial document with stated intent for review. This then gets broken down to the sub-blocks (just called Block A and Block B in the list), which themselves may reflect the top-level document structure with an opening summary paragraph, before detailing the design and any sub-blocks. The next three list items are the usual order that development might take place, with simulation testing, followed by synthesis and then testing on platform. A document is taking shape. Other developments, such as for ASICs, might have additional sections (I/O, power, DFT, power regions etc.), but the principle is the same. These sections can now be broken down into sub-sections as needed. I would normally go at least one level down adding headings for these sub-sections. In some organisations there are templates for certain documents, so the headings may already be defined. I'm not a big fan of this as a "one size fits all" approach, which often has engineers figuring out what they are supposed to put in a given section rather than writing documentation.

However it's done, having a document with the sections and sub-sections already defined up front allows them to be filled in as soon as the information becomes available. Sub-blocks can be treated like mini-IP blocks, with an initial block diagram and accompanying text. This means that each aspect of the design is documented at the point where design decisions are made, and thus these decisions can be captured. For instance, it may be that there are, say, three evident ways to implement part of the IP and one has to be chosen. All engineering is compromise—not just in technical terms, but schedule, complexity, risk etc.—and so all choices must be documented and the reason why a particular path was chosen. It maybe it didn't matter and the choice was arbitrary or reflect personal preference, but this

must be documented so that when the IP is used in different circumstances, the decisions can be reviewed for validity.

The sub-sections may need to summarise their context, which would repeat information in other sections. It shouldn't be assumed a reader of the document is going to read from the start of the document until reaching the sub-section, and so a summary at the start of the sub-section, with references to the other relevant sections, will allow a reader to get to the relevant information more quickly. If the summary is sufficiently generic, then changes to the design should not need the summary to be re-written, unless it is a major change. This document maintenance overhead is, I think, small compared to making the document usable.

Language Style

Technical documentation can be 'dry' and boring to read. Partly this can be an unavoidable consequence of the nature of material (it's not a Dan Brown thriller, after all). We talked about telling the story in the previous section, but the actual text can help engage the person reading the document. When I am writing a document, I will always take some time before writing a paragraph or two to say in my head what I want to convey as if explaining it to an engineer who is perfectly competent but who knows absolutely nothing about the design or its context—as if, for instance, they had just joined the company.

In this way the text will flow in a more natural way. I have seen many technical documents be a list of sentences that just state a fact: this, then this then this, and so on. It is really hard to concentrate (and a table might have been more useful). If writing in a conversational style, then reading should also be conversational and thus more engaging, increasing the information exchange, which is the point.

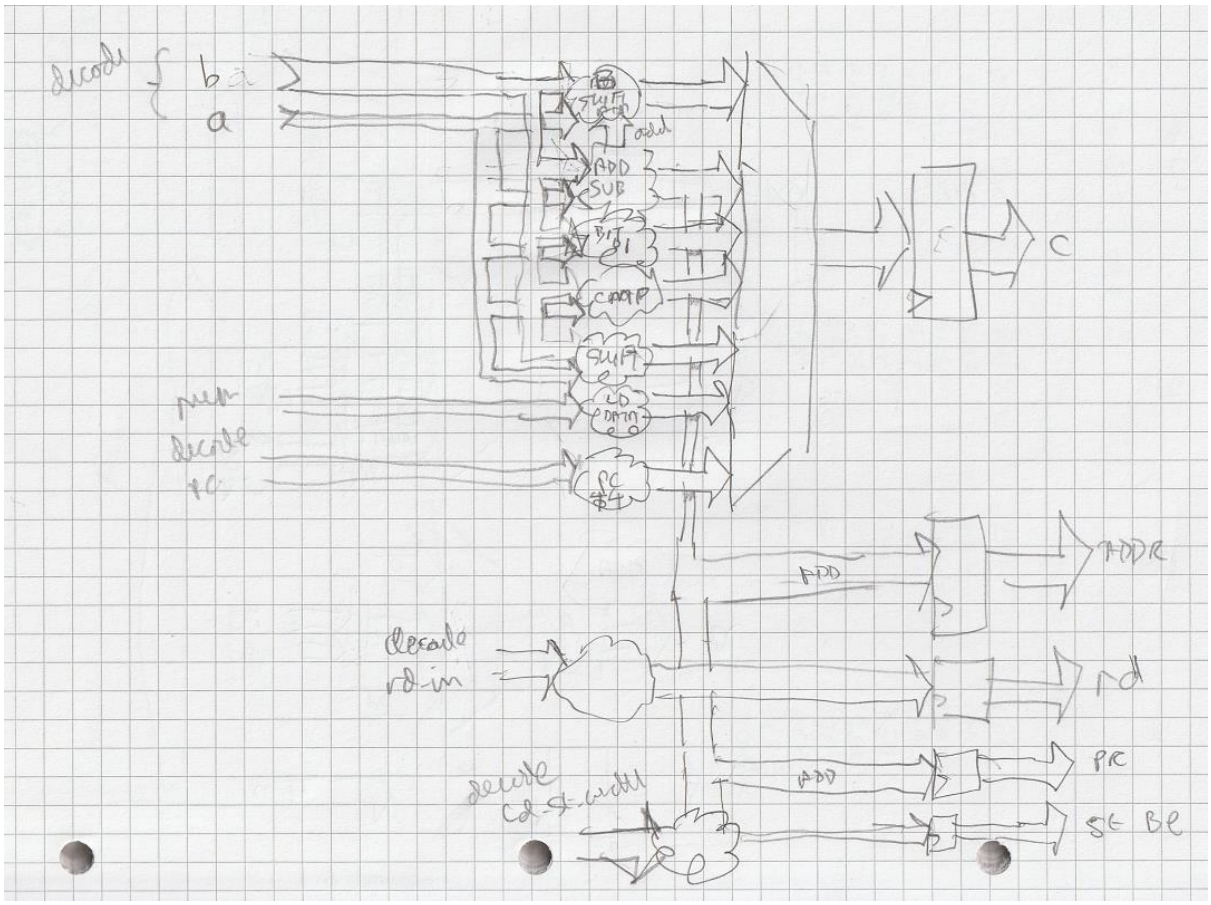
This all comes with practice of course. The more one writes the better one’s documentation skills become. Also it has been shown that language skills are improved by reading, of any description. If you find a technical document (or two) that you thought was well written, then try and emulate its style as much as possible.

Diagrams

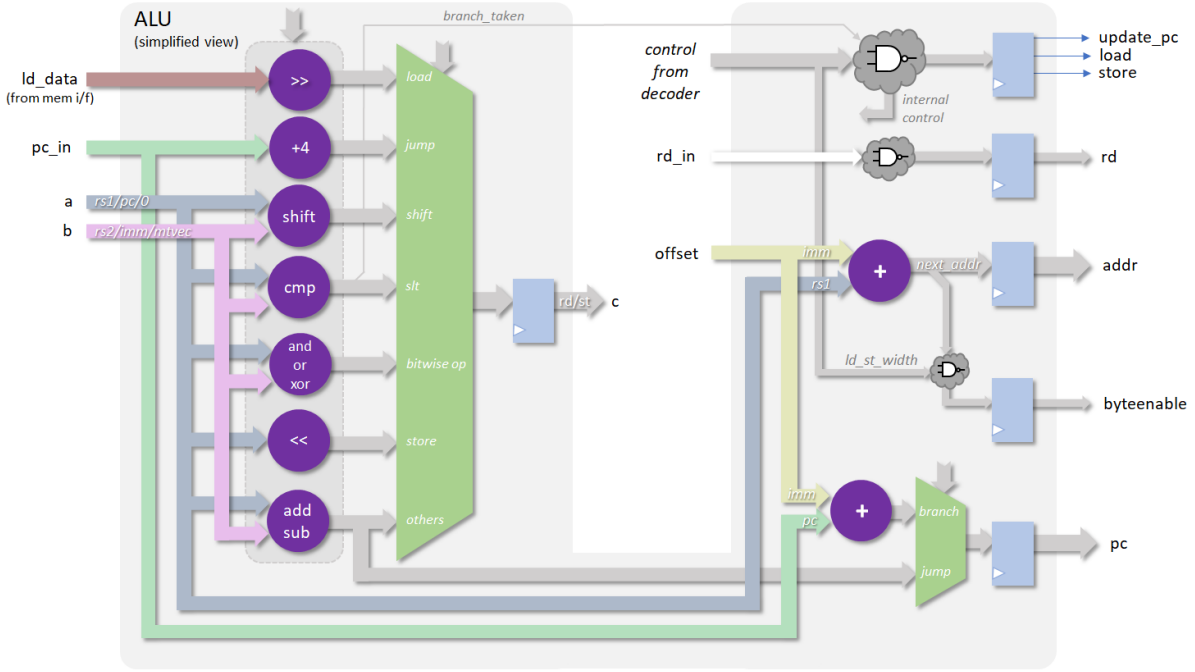
Diagrams are the most important part of a document (a thousand words and all that) but, for me, they are also key in the design process. If, as has been mentioned above, we wish to have the design intent captured and reviewed before implementation is commenced, then what better way than to sketch out the block

diagrams (and maybe waveforms), place them in the design document and add some descriptive text?

For me, when designing, I would always start with pencil and paper (I'm old school), just to get a rough idea of where I'm headed and to concentrate my thoughts. This quickly becomes messy, with much erased and redrawn, and my handwriting is terrible:



This quickly becomes unwieldy, but now that I have a basis from which to work, I can construct a full-blown diagram, such as the ALU block diagram shown below:

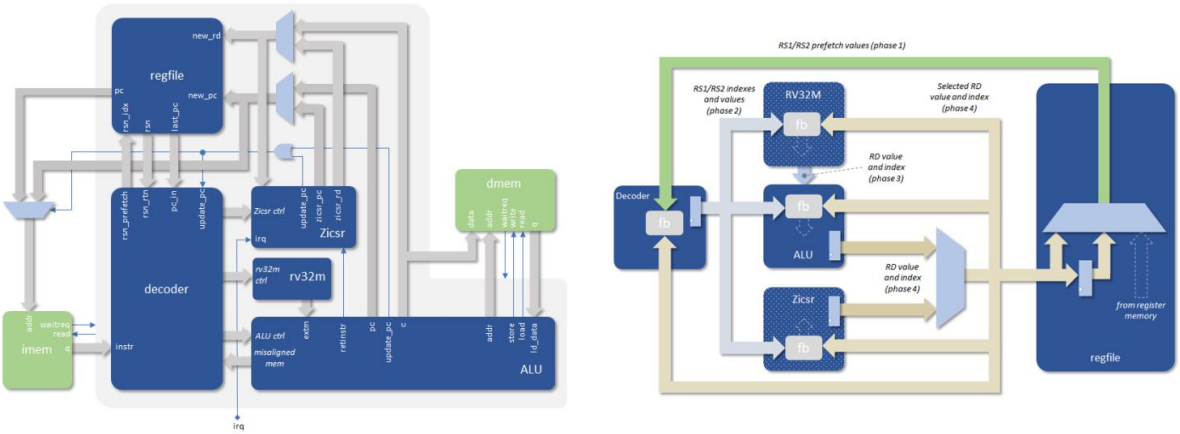


Notice that this document differs in some details to the sketch since, having now got an electronic version, I can proceed to alter it as the design is formulating to include all the features. It is at this point, for me, that I can start to see things I might not have seen before in the conception. For instance, in the above ALU design, the first drawing had a much longer timing path than the final diagram, and two pipeline phase calculations were squeezed into one. This would be hard to spot without a diagram and may not have been found until after implementation and synthesis weeks, or months, down the line.

I tend to use colour in my diagrams to add information by, say, grouping blocks together, or identifying a common attribute (e.g. a clock domain). I do also spend time trying to make the diagrams look 'attractive' with the use of rounded boxes, shadowing etc. This takes only a little time to do initially, and then diagram items can be copied and pasted from earlier documents. I do this as I wish to engage the person reading the document. As discussed before, technical documentation can be a dry read. Making the diagrams attractive and engaging is as important as the style of the text to tell the story. I was accused once of spending my time 'moving boxes around the screen' (not by my managers) but, for me, it is when I am putting together these pre-implementation diagrams that I start to 'see' the design, its flaws, and develop it to be on a sounder footing before writing code. Therefore, I am designing and documenting in parallel and getting nearer to the final design than if I were to start coding from the off. The colours and styling I personally use I don't believe is essential in a document's diagrams, and square empty boxes and shapes with standard arrows conveys the important information. But, hey, no-one's complained about it and, indeed, I've been told they convey a sense of

professionalism. That may seem egotistic but if, for example, one is writing a proposal for a new idea that one has to convince your company to spend time and resources on, then the diagrams can give subliminal messages as to the idea's credibility, maturity, and the belief in it from the proposer. Trust me, I've seen this work.

When adding the text to go with the diagram one major rule I have is there should be nothing on the diagram that is not referenced in the document's text. This may be collectively (e.g. describing a multi-signal interface) or explicitly. The other way around is okay, with the text describing more detailed aspects of the design than is shown in the diagram. Another alternative is to separate different aspects of the design into two separate diagrams. It is a difficult call sometimes, but it is important that a diagram is crammed with too much detail, so splitting it may be the way forward. Below shows two aspects of the top-level blocks of a RISC-V core, one showing the major connections between blocks, and the other the pipeline bypass feedback paths.



As can be seen, the five blue blocks are the same, but with different aspects of the design shown. Since the bypass paths need special attention in their description they were separated. In any case, the original diagram would be overcrowded and unclear if the bypass paths were included. There are no hard and fast rules and a judgement call needs to be made about what to include and what not to include in a given diagram.

Code Comments

Ask ten engineers about how code should be commented and you'll get ten different answers. I have even heard (from a talented and very senior engineer) that he doesn't comment very much as they get out of date. He did also said that one tends to add detailed comments on bits of logic that went wrong and needed fixing, forgetting about the rest. I think the first goes too far, and the second is very true.

Without diving into the comment wars, I think we can address these two viewpoints with compromise. Comments must be accurate and kept up to date. My take is that, when modifying code, update the comments first then make the change. It doesn't matter that it may change again, just use this same procedure. Comments should be as minimal as possible whilst still conveying the operation. Obvious statements can be dropped ('add A to B and store in C'), but the function of even the 'obvious' logic should have brief descriptions. It may be obvious now, but what about two years' time?

One way to reduce the amount of comments is to make reference to the documentation that described the logic. This way, when changes are made and (as we've recommended) the documentation updated before implementing the changes, then the comment that references the document's sub-section remains accurate.

The 'Customer'

When I worked for Hewlett Packard in the 1990s, the customer was prominent in the working philosophy. They were seen as partners, and the aim was to give them the best possible deliverables to help their businesses succeed. But there was also this idea of 'internal' customers. That is anyone who was a recipient of one's deliverables. These customers should also be treated as partners, and the best possible delivered to them as well. Thus the chain of quality is maintained right through to the external customer. So who are the customers of the internal design documentation?

The first customer I want to mention (and these are in no particular order) is the engineer who comes after you. It is very likely that your designs will be maintained, modified or enhanced by another engineer whilst you are working on another (more exciting new) project. Or you may have even left the company. Doing a good job with the documentation reduces the amount of support you'll be required to give, and less interruptions will make you more productive. If you have left the company then you may very well be that 'next engineer', picking up somebody's IP. What would you want to see with the IP you have just been handed, designed by an ex-employee? That is what you should aim for with your documentation efforts.

Another customer might be the embedded software engineers, if you are working on a CPU based logic design (which is quite likely). The programming registers are the view that the embedded software engineers see when writing drivers and control code for the logic and system in general. Any block with its own registers should have some sort of data sheet—which might be auto-generated (see *Tools* section below). This should explain the registers and their bit-fields

(where applicable). If the block, from a programming point of view, is self-contained, then the relevant sub-section can just have some accompanying text to explain how to initialise, use and shut-down etc., the registers. More likely is that, for a given system function, registers from multiple blocks will need to be configured and used. Often the documentation does not include this information, making it hard for the software engineer to work out how to use the logic—I know, I've been an embedded software engineer. Therefore there may need to be sections in the documentation for the 'use cases' for each top-level function, describing how to set the logic up for various operations. If, as should have happened, the logic engineer worked with the software engineer to define the use cases and what the software/hardware interface should be like, up-front, then this can be captured (and reviewed) before implementation. I have too often seen logic developed, simulated, synthesised, and delivered without reference to use-cases with the software, making the logic extremely difficult to use. The logic engineer will claim that all functionality has been tested and works but, having not considered the wider system usage, particularly software, the design gets a reputation as being difficult to use. Even if the logic engineer designs the interface without reference to the software, if it is documented up-front and reviewed, it will at least open up a dialogue between the two disciplines and catch potential use hazards.

Whether the programming guides should be separate documents from the internal logic design is open to question. My feeling (and experience) is that maintaining two documents with common information, thus duplicated, is a difficult, error prone task and can lead to misunderstandings. Organising a single document with sections for usage alongside sections for logic design allows common information, like the register maps and details, to be documented just once and referenced from the separate sections.

The last customer, and the most important, is yourself. If you haven't left the company, IP that you created 6, 12, 24 months earlier may suddenly need your support and your head is full of all that has happened in between. In this case not only might you have forgotten what you did, but also why. I personally have refactored old code because it looked more awkward and complex than it needed to be, only to find out why I had done it that way when tests started to subtly fail. I'd put nothing in the comments beyond stating what it did, and nothing in the documentation explaining the particular need for the added complexity (it added a small amount of complexity to save a lot of complexity on a timing critical path). In these cases the rules of being the 'next engineer' are very similar, so write good documentation for your own sake, if for no-one else's.

Tools

Regarding what tools to use I'm easy. Wikis are now becoming the norm for documentation as collaboration is so much easier and cross-referencing to other documents and their subsections a breeze. I tend to use wikis, with Confluence my most recent professional experience and Xwiki for my own project documentation. Both have drawing packages available of varying sophistication. I tend to draw in PowerPoint because I'm familiar with it (so work fast) and because I have ready-made slides for presentations. In a wiki I capture the images as .png files and add to the wiki page as images. To ensure that a future engineer can change or update them, the source PowerPoint document is attached to the page and uploaded each time the original has a modification. If Visio (for diagrams), or Microsoft Word or any other tool is preferred, than that is fine by me. Content is the key.

Wikis and word processor applications usually have the ability to be saved as PDF files or, at the very least, the PC can 'print' to a PDF file. It is usually this method I would use for distributing documentation in a package or github repository. Being able to read PDF files is ubiquitous nowadays.

Auto-generation of documentation is also something that is widespread. I have been involved writing scripts for generation block registers datasheets from a common register description source (e.g. spreadsheet or JSON), often used as the source for generating the HDL that implements them, avoiding the pitfalls of duplicate information drifting apart. The software world has had Doxygen for many years now, and this now supports VHDL, which can even be integrated into the commercial Sigasi HDL IDE. I believe Verilog support is also being developed.

Conclusions

Bad documentation affects the value of a piece of IP. Inaccurate documentation is even worse and may cause misunderstandings and avoidable defects in implementation. No documentation at all sits somewhere in between—at least you can't be misled. Not properly writing documentation costs money, because effort is required to reverse engineer or re-implement IP or cause an unplanned support overhead.

In this article I have summarised my approach and experiences to help make it easier to generate good documentation by starting early, getting reviewed, having good diagrams as part of the design process, and adopting a divide and conquer approach. Telling the story, and using conversational language also makes the document more user friendly. We looked at who the customers were, placing

ourselves in the position of those who will use the document (including ourselves down the line) and being thoughtful as to their needs.

I have not put forward any hard and fast rules as to how a technical document should look, or dictated which tools should be used, but presented just my approach as someone who has written a lot of documentation and who has been praised from many quarters as to its quality. I would expect anyone (who's got this far—congratulations) to adapt what I have said here to their own local needs. The important point, for me, is that it should not be optional and not quickly cobbled together at the end of a project, or you devalue you IP.