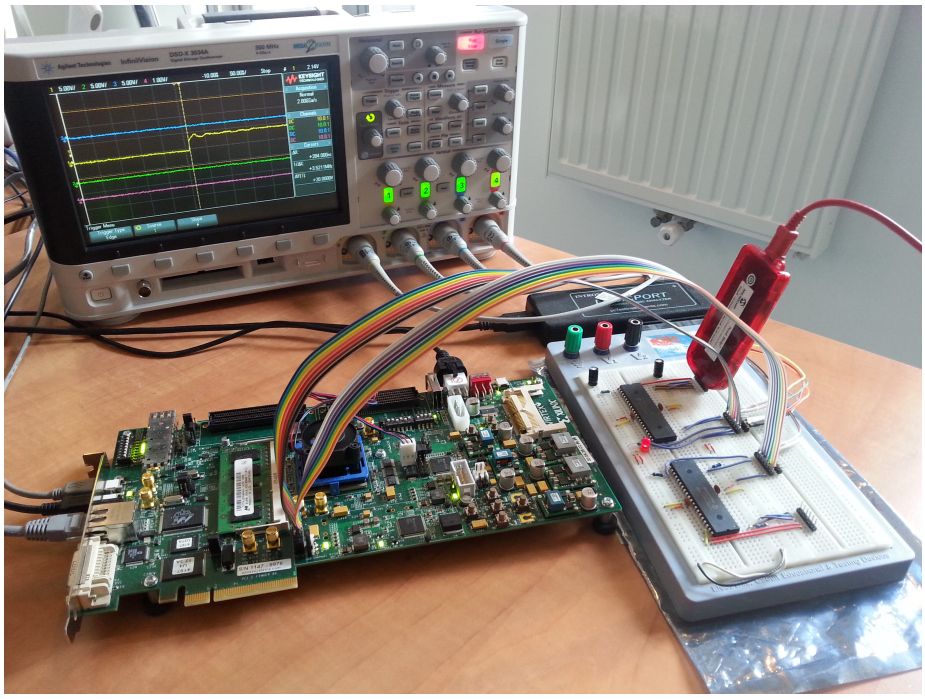
PPU, a protocol parsing unit in hardware

Remco de Wit

March 27, 2015



Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays we live in the world of the internet of things. More and more devices are interconnected and communicate with each other. Devices connected to the internet of things use wireless and wired connections and dozens of different standards to communicate with each other. Each standard uses its own defined protocol and protocols are stacked on top of existing protocols. This means there are very many different protocols and implementations of those protocols out there.

The complexity of the protocols varies greatly between them. From very simple protocols like SPI and RS-232 to very complex protocols like TCP/IP. The simple protocols can easily be implemented in a small hardware module. But the most complex ones need software and a lot of processing power to process them. Protocols can also be stacked on top of each other, like in the OSI model [18].

When designing a new product which might use protocols or protocol stacks, development can quickly become difficult. Because protocol implementation can be so difficult, most designers use libraries or pre-build solutions to implement protocols in their products. But in the internet of things world, where many small devices are used, most devices don't need fully supported complex protocol stacks, or they need to connect 2 completely different protocols. Standard solutions are too complex, may require too much processing power, or do not exist.

To find a solution for the difficult development of new protocols is the starting point of this thesis. A new method to ease the development of new protocol implementations is proposed. The hypothesis is that by using grammars the development of new protocols can be made easier. Grammars can be checked by formal methods for correctness of the protocol description, where normal programming languages cannot. For normal programming languages test cases are needed to verify compliance with a protocol description. When designing test cases, rare occurrences in protocols can easily be missed, leading to errors. Then the advantage of a grammar based approach becomes clear.

When a grammar is used for a protocol description, it still needs to be compiled for implementation. While this is possible with the modern compilers, the resulting code is often slower than a low-level code implementation. In an environment such as embedded system, this is not desired. But what if a small hardware parser, designed specifically for protocol parsing, can be used

to parse the grammar? This would allow easier integration due to the grammar description and good performance because of the hardware.

1.1 Current methods

In this section the 3 most used methods today are discussed. Also their performance metrics are evaluated. At the end of this section, the 3 methods are compared and the grammar based hardware parser is positioned relative to the current methods.

The 3 methods of processing protocols used nowadays are:

1. Software processing, the protocols are parsed on a general purpose processor (GPP)
2. Accelerators, the protocols are parsed on hardware
3. CAMs, the protocols are parsed on a special type of memory

Each of these methods will be discussed in the following sections.

1.1.1 Software processing

Most protocols are processed by software on a general purpose processor (GPP). In most systems a generals purpose processor is used for controlling the main task of a system and because modern processors are very powerful is it very easy to include tasks like protocol processing on the same processor.

Advantages

The main advantage of software processing is the cost factor. If it is done properly the development costs of software are high, since programming and testing of communications systems (which are usually considered critical) can take a lot of time. But because software does not require any additional hardware on a system with a GPP, the hardware costs are zero.

Software is also very flexible. Complex constructs can easily be implemented in software. When a program is made in a high level language it can be ported between hardware architectures if the suited compilers are available. However, the flexibility is limited by the hardware it is implemented on. If a communication module only supports 8 bit transfers and you need 13 bit transfers, both the hard- and soft-ware need extra operations, reducing the performance. In theory any protocol can be implemented in software using bit-banging [20], but this reduces the performance of the system a lot.

When libraries are used to develop software for protocol processing the time-to-market can be very fast. When libraries are not available it will take considerably longer.

If a GPP is programmed it makes no difference if a protocol implementation is included in the programming file or not. The complexity of the programming does not change and only the time to program the GPP increases. This makes it very easy to integrate a protocol into a GPP.

Disadvantages

The main disadvantage of a software solution is the low performance. For example TCP/IP processing with software requires about 1 Hz/bit in processing power [8]. Protocol parsing is not very efficient on a GPP, because for fast interfaces a very powerful GPP is required. GPPs have problems when dealing with:

- Multiple small data segments, smaller than the processors word width
- Pipeline flushes because of branches, of which there are a lot in protocols
- Low latency restrictions, if other software also runs on the GPP
- High power consumption when processing protocols (compared to hardware solutions)

The power consumption and the performance make a GPP less ideal for protocol parsing, but still the GPP is the preferred component for parsing protocols due to its versatility, ease of integration, and its low cost.

1.1.2 Accelerators

Besides processing a protocol in software on a GPP, processing can also be done in hardware. Protocol parsers made in hardware are called accelerators. Accelerators provide very good performance, because they can be tailor made to a specific situation. The specialization of hardware to a specific function allows the hardware to extract the most parallelism possible, and/or use the least amount of power during processing.

The main disadvantage of accelerators is the cost of such devices. The low sales volume and high development costs (for silicon) make them very expensive. Because accelerators are custom made for each protocol they cannot be changed or reconfigured, which gives them a relatively short lifetime. To compensate for this, some accelerators have a programmable memory to allow a slight change in function, increasing their lifetime, but also increasing the cost of a device.

The final hurdle for accelerators is the adaptation in systems. Most tasks for which an accelerator is used, are tasks that are a heavy load on GPP resources. The accelerators have to compete with the software implementation of the task, which is less expensive in terms of money. However, when looking at GPP load the cost of an accelerator is lower (obviously), but the cost of power is less obvious. If an accelerator has a high workload, the reduction in power compared to a software solution is large. But when an accelerator has a low workload, it will consume static power when idle, and the power consumption could be worse than a software solution. In practice only accelerators that provide a huge performance gain (like graphics processors) or energy savings (bitcoin mining) make it to the market.

1.1.3 CAM

Content Addressable Memories (CAM)[21] are a special class of accelerators. CAMs are normal memories, but incoming data (content) from a protocol is connected to the address lines of the memory. The CAM searches its entire

content to check if a match of the data exits in memory, and then outputs the address(es) where the matched data is stored. It is basically an indexing machine.

The address returned from the CAM can be used by the GPP to load a specific function, which executes only the required processing. For example, a TCP header enters the system. The data from the TCP header is entered into the CAM. The CAM outputs an offset address, which is used by the GPP to load a function that is dedicated to process that type of header.

This method allows for very fast parsing of protocols and it is very flexible (the memory contents can be changed). The downside of the CAM is you need to program every possible input that you want to parse into the CAM. For small protocols with only a few possible states this is not a problem, but for large protocols like TCP/IP a very large memory is required. This causes the CAM to become very expensive, because in order to search its contents quickly a CAM needs a comparator for each memory location for input matching. Therefore a CAM is expensive for all protocols, except the very small ones.

Only for very high performance or for small protocols CAMs are used. For example CAMs are used in ethernet switches for MAC address lookup, where a limited number of MAC address can be stored, and the switch gets the port number to where the data has to be send to very fast.

1.1.4 Summary of current methods

Currently the 3 most used methods for parsing protocols are software parsing and hardware accelerated parsing, by means of an accelerator or with CAMs. Software processing has the lowest cost with the fastest time to market, however, the performance of accelerators and CAMs is much better than software solutions (see Table 1.1). Software solutions are the most widely used option for protocol processing, because of the low cost and short time to market. Only when a large power or performance advantage can be achieved, accelerators or CAMs are used.

	Software	Accelerator	CAM
Hardware costs	++	-	- -
Development costs	-	+	+
Performance	- -	++	++
Flexibility	+	-	+
Time to market	+	-	+
Ease of integration	+	-	-

Table 1.1: Protocol processing solutions

1.2 Scope of this thesis

Now that the current methods of protocol processing are known, the scope of the thesis work can be determined. In this thesis work the main question to be answered is:

- Can a hardware module be designed that processes different communication protocols like a grammar?

This question leads to the following related research questions:

- How does the performance of the hardware module compare to a software implementation?
- What is the cost of such a hardware module?
- What are the limitations of such a module?

The goal of this thesis is to design and implement a hardware parser processor, that can process more than 1 protocol. This hardware module will be called a Protocol Parsing Unit (PPU). The hardware parser should be targeted for communication protocols. After implementation the module is evaluated against a software implementation to determine the performance, the limitations and the hardware size. The requirement for the ability to process more than one protocol is important, because otherwise an accelerator is designed, which has been done many times already.

The ultimate goal of the thesis work is to verify if an implementation of a hardware parser leads to easier design of protocol and with fewer errors being made compared to current solutions. However, to verify this would require a lot of work and it is too much work for the allocated time.

Before a start can be made with designing a hardware parser some background information is required. In Chapter 2 all this information is presented together with information about the document itself. In Chapter 3 the design considerations are presented. Several different methods of solving the problem are discussed and for each problem a method is chosen for implementation. The implemented design will be discussed in detail in Chapter 4. The results of the implementation are presented in Chapter 5, followed by the conclusion in Chapter 6 where also future work is discussed.

Chapter 2

Background

In this chapter all information that is needed to properly read this document is presented. In section 2.1 the theory of grammars is discussed. But first the diagram definitions are explained.

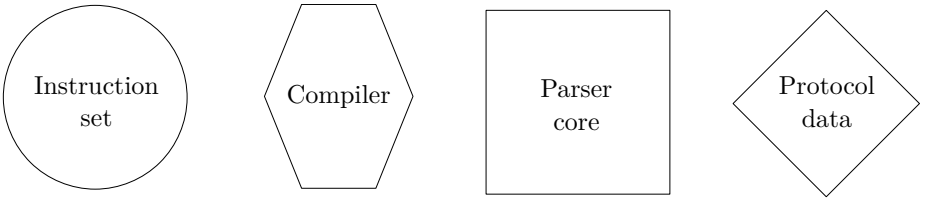


Figure 2.1: Design blocks used in this document

In this thesis several diagrams are used to illustrate high level design architectures. In figure 2.1 the different types of blocks shown in those figures are shown. Circles indicate a description or a specification. These blocks do not process or transform information, but they specify input for another block. The hexagons are conversion blocks. In these blocks an information flow is converted, but no new information is added (like a compiler performing code to byte-code conversion). Squares indicate hardware modules. These blocks indicate hardware blocks that store, process, or convert data. The diamond blocks indicate blocks with IO data from an interface or another processor. All these blocks are connected with arrows indicating the flow of information.

In this document a few risk analysis table are presented. In these tables the exposure and the impact are rated on a scale from 1 to 10. The exposure indicates the probability that during development something goes wrong and disrupt or delay the development. A high exposure means that there is a high probability that something is overlooked or that the design does not meet specifications. The impact indicates the severity of the problem if something goes wrong. A low impact means that there is an easy work-around if one solution fails.

2.1 Grammars of protocols

A communication protocol is a form of language with certain properties. If these properties are known, an efficient system can be implemented to process them. In this section protocols will be reviewed from a linguistic point of view, to find the properties of the grammar of protocols.

Chomsky [6] defines that a language can be captured with a grammar class. In the first subsection of this chapter the grammar class of protocols will be determined. The grammar classes are briefly introduced and it will be defined in which of these classes protocols are found.

In the second section parsing algorithms for protocols will be discussed. Parsing algorithms are used to define a structured way of decoding a grammar. A suitable algorithm will be selected for implementation in the PPU. Hardware restrictions will also be taken into account when selecting an algorithm. Detailed hardware limitations and implementations can be found in chapters 3 and 4.

2.2 Grammar classes

Chomsky has defined 4 types of grammar classes for languages. Each class is different from the next in terms of expressibility. However, expressibility comes at a cost of being more difficult to parse. The following classes are defined by Chomsky from the most expressible to the least:

- Type-0 grammars are unrestricted grammars. This grammar type can be used to describe a Turing machine. It is the most expressive grammar class, however, because of the decision problem [22] it is not used in practice for the creation of languages. There are no protocols to date that fall into this type of grammar.

- Type-1 grammars are context-sensitive grammars. This grammar is known by most persons as natural languages (the language we speak). Words follow each other in a specific order, however, words can be different depending on the word before or after the current word. For example 'walking' changes to 'walk' if it the previous word is 'I' and 'a' changes to 'an' if the word after starts with a spoken vowel.

- Type-2 grammars are context free grammars. In context free grammars words do not change depending on their context, but they are in a predefined order. For example in the programming language C a function declaration is always followed by a '{' independent of the name or complexity of the function. Grammars in this class can be efficiently parsed by computers, whereas grammars of type-1 or 0 cannot[19], which is why type-2 grammars are used in most programming languages.

- Type-3 grammars are regular grammars. These grammars are the last and most restrictive class of grammars. Regular grammars are the least expressive grammars, there form allows on a relatively fixed sequence of words. For example an address can be constructed with a regular grammar. A fixed sequence of the street name, street number, postal code and city name can be described with a regular grammar. Whilst the street

name, street number etc can have an arbitrary value, the overall sequence is fixed.

Type-0 grammars are of no interest for this research. Bloks [4] analysed protocols and concluded that all protocols can be described with type-1,-2 or -3 grammars. Full type-1 class protocols are not very common. Usually only a select number of fields in a protocol are context-sensitive (like checksum and length fields). The remainder of the protocol is a type-2 or type-3 grammar.

Most protocols can be described with a type-2 or -3 grammar if an extra construct is used for the few context sensitive fields. This is also true for programming languages, which use the same principle to deal with context sensitive parts. For that reason the focus is on type-2 or -3 grammars. For both grammar types different parsing algorithms are available, in the next section these are reviewed and evaluated.

2.3 Parsing solutions

The main difference in processing type-2 or type-3 grammars is in the complexity of the parsing algorithm. To process all type-2 grammars a non-deterministic push-down automaton is required. But most type-2 parsing algorithms use a deterministic push-down automaton, which have some restrictions. Regardless of the chosen algorithm all push-down automata require a stack to operate.

Based on the contents of the stack, the input and the state of the automaton a decision can be made to transfer between states. A few common algorithms for context free grammar parsing are LL, LR and LALR [1]. For implementation in hardware LALR is the most promising algorithm. This algorithm is very expressive and still has a relative low complexity (compared to LR(1)).

For parsing type-3 grammars a Deterministic Finite Automaton (DFA) is sufficient. These regular grammars are much more restrictive than context free grammars, but the algorithm for parsing them is a lot less complex. A DFA is very suitable for integration in hardware and thus type-3 grammars can easily be implemented. In the next chapter the differences between both methods are explored in more detail. The relations of algorithms and the hardware implications are described in detail and with that information an algorithm is also chosen.

Chapter 3

Design considerations

In this chapter all design considerations are discussed. A structured approach from a protocol specification to hardware design is taken. In Figure 3.1 the overall architecture of the PPU is shown. In the left block the front end of the system is shown. A protocol specification is described with an implementation language. This language can be supplied to the compiler, which in turn generates byte-code for the PPU hardware. Section 3.1 is about the language used to describe a protocol implementation. In Section 3.2 the compiler is discussed.

The core hardware is shown in the middle block. The memory and the parser core are the blocks responsible for the protocol parsing. The instruction set and the memory lay-out are key design specifications for functionality. The core hardware is described in sections 3.3 and 3.4.

The last block is the IO of the PPU. The PPU has IO modules for protocol data (Section 3.5), but also an IO module for communication with other hardware (like a GPP)(Section 3.6). The communication with other hardware is done through the API.

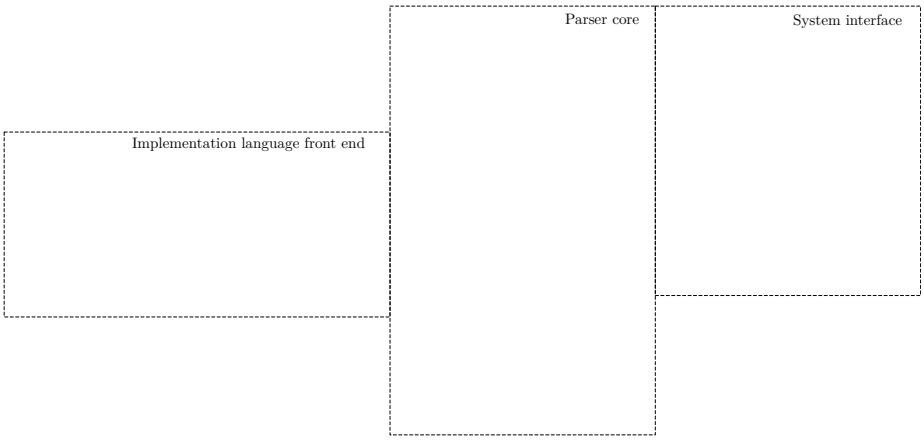


Figure 3.1: Overview of the PPU design

3.1 Implementation language

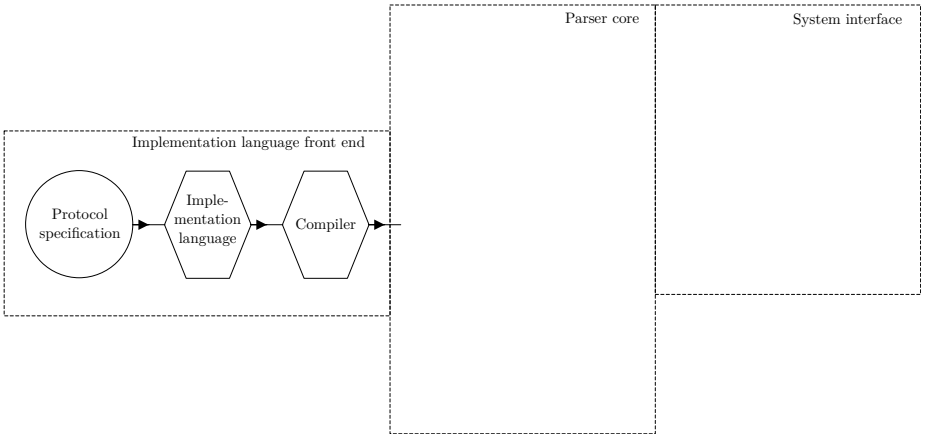


Figure 3.2: Front end of the PPU design

In a protocol specification the functionality of a system is described. This description is often written in a natural language (like English) and such a description cannot be used to program hardware. Because there are no compilers that can translate a natural language to byte-code. An intermediate language (the implementation language) is required to gap the bridge between the natural language and the compiler. For regular software this language is a programming language like C or JAVA. However, these languages are not designed for parser programming. A suitable language has to be found which can gap the bridge between a protocol specification and the compiler.

Language selection

With all different kinds of protocols and specifications a selection had to be made to choose which protocols should be supported. A starting point was to support protocols that could be described by Backus-Naur Form (BNF) [14]. BNF was chosen based on the fact that it is a language used to implement context free parsers. After a detailed evaluation BNF turned out to be unsuitable.

Several protocol related concepts cannot be expressed with BNF. A few other languages are suitable, Augmented Backus-Naur Form (ABNF) [9], Prolac and NetPDL. These 3 languages are designed for protocol description. But since only ABNF is still in use and the other 2 languages are no longer supported, ABNF was chosen for the implementation language. Thus the goal became to implement a hardware parser that can support languages written in ABNF.

ABNF operations

After ABNF was selected as the implementation language, the operations that can be implemented with this language were investigated. From ABNF a compiler must be able to generate byte code for the hardware and for that conversion the operations in ABNF must be known. For the PPU hardware the operations must also be known before they can be implemented.

The core operation of ABNF is pattern matching. In ABNF there are constructs for single value patterns matches and ranged value ones, for example:

```
zero = %x00                ; single value match
zeroToNine = %x00-09       ; ranged value match
```

Other operations for matching are available in ABNF (like concatenation), but these operations can be reduced to single or ranged value matches by a compiler.

Besides the matching operations, there are supporting operations for flow control and branches. The first is the alternative:

```
Fruit = apple / pear
```

Both *apple* and *pear* are valid fruits according to this ABNF description. What is not yet known is the state following the fruit state. This is important to know, because in ABNF *fruit* might have a different state transition based on the match. If there is no deeper level of one of the elements, the next state is the same for both alternatives. for example:

```
apple = %x00      ; apple is value 0
pear = %x01       ; pear is value 1
```

However, if there is a deeper level available to one or both of the elements, a different next state is present (eg a branch). In the example below *apple* has *red* as first matching value. On the other hand *pear* has *green* as first matching value. Both *apple* and *pear* have a different second matching value, namely *round* and *oval*. Thus a branch has occurs when either *red* or *green* is encountered on the input.

```
apple = red round
pear = green oval
```

The last operation of the ABNF language is repetition. In ABNF there is support for defined and undefined repetition. Defined repetition is a fixed amount of repetition cycles, while undefined repetition repeats for a unknown number of cycles.

In protocols defined repetition is used almost exclusively. Some protocols use undefined repetition, but they have some symbol, flag or other method to identify the end of a repetition. Unconditional repetition can be parsed with a conditional branch. This means only defined repetition needs to be implemented as an operation.

The operations defined by ABNF are:

- Single value match
- Ranged value match
- Alternative value match
- Branches
- Defined repetition

Hardware operations

The operations above are the operations which are defined in ABNF. But if actual protocols are examined some operation modifications are required to build a working parser. Ethernet is used as an example, because in an ethernet it is easy to show why certain modifications are useful.

Preamble	SD	Destination address	Source address	Ethertype/ Length	DSAP	SSAP	Control	Data	FCS

Figure 3.3: Ethernet header

The first field is the mac source field. The address of the source field is not known before runtime, but a variable that can change during runtime. Matching against such a variable should be possible, as well as storing such data. To support this, a memory is required that can be used for these matching and storing operations.

Another set of operation modifications are delayed branches and repetition blocks. In the ethernet header, the ethertype field indicates which higher layer protocol is used. Depending on the value of this field, a different parse branch has to be taken. But if a jump is taken immediately after parsing the ethertype field, all branches must include the remaining header section of the ethernet header (the DSAP, SSAP and control field). With the delayed jump a branch decision can be made at the ethertype field, but only after parsing the control field it is executed, saving memory space. This is why it is useful to have a delayed option on jumps and repetitions (which have the same problem).

Hardware actions

Not only modifications to the operations need to be made for a hardware implementation, also actions have to be defined. To understand why actions are required a quick look into Lex and Yacc [7] is made. Lex and Yacc are used to parsing programming languages and create compilers. In Lex and Yacc a construct called 'actions' is added to BNF. Actions define what must happen when the parser matches a certain string. To improve performance the PPU should have some basic actions available.

With the modifications to the operations many protocols can be parsed efficiently. But the system still has to have support for actions. Some information in a protocol may not be useful for the user. Information for flow control or other low level information can often be discarded by the PPU without affecting the user data. For example in a TCP packet only the IP address of the sender, the port number and the packet data are useful for the user. The hardware needs to know which data can be discarded after parsing and which data is necessary, so a store action is required.

Another set of actions that will be included are transmit actions. While the main focus of this work has been on data reception, the hardware should be capable of transmitting data as well. With the transmission actions included, the operations and actions required for the PPU are defined. While the number of actions is very small, these are sufficient for a working PPU device.

The following list contains all the operations and actions required for the hardware:

- operations
  - Single value match
  - Ranged value match
  - Alternative value match
  - Branches (delayed)
  - Defined repetition (delayed)
- actions
  - Store
  - Transmit

3.2 Compiler

The next step of the design is the process to translate the implementation language to byte-code for the hardware with a compiler. Although no compiler is implemented, it was not left out of scope of the project entirely. Compilation for the PPU was done by hand, during evaluation and testing. This section is for reference of what research already had be done, during the thesis work.

The compiler was planned to be implemented last, since it requires knowledge of both the implementation language and the hardware platform. One aspect of the compiler is considered, how to implement the language compilation. Since pure ABNF is not capable of describing the context sensitive parts of a protocol, an extension for ABNF is required for the context sensitive parts. Also actions like making protocol data for storage or marking data as a variable are not implemented in ABNF by default. One method of implementing these field are by extending ABNF. For example:

```
Normal ABNF
test = <length> <header> <length>*<length><data>
  How can the compiler know that the length field
  is also the size of the repetition?

Marked ABNF
test2 = @<length> <header> $<length>*$<length><data>
  @ indicates store in register
  $ indicates use from register (with same name)
```

There is a downside to this method. ABNF is very easy to read in its original form. It is also very easy to keep track of the flow levels and thus in ABNF a protocol can easily be checked manually for completeness and errors. With the addition of the store and memory notations checking the protocol will require more effort due to the reduced readability.

Another way of implementing these additions is by using a two-stage compilation. The first stage of compilation will happen with pure ABNF. Then an intermediate language will be created where a specification can be made which fields are connected or which fields need to be stored etc. This method has the advantage that the high level ABNF will remain readable, the downside is that a second level of notation and compilation is required. Which method to use is not determined, since there was not enough time to evaluate it.

3.3 Parser core

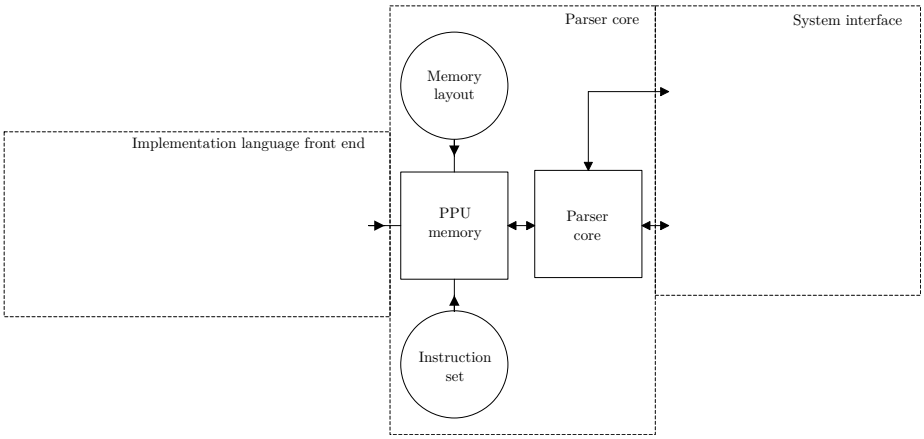


Figure 3.4: PPU core parts of the PPU design

In order to properly examine the PPU memory some information about the parser core is required. In this section the parser core basis will be determined.

A parsing algorithms is required to base the parser core on. The parsing algorithm should be relatively simple to allow it to be implemented in hardware. Implementing a parser that can parse all protocols that can be described with ABNF is not easy. ABNF is a rather expressive language, even with an LALR parser algorithm [1], not all protocols that can be designed with ABNF can be parsed. However, a LALR parser was the best fit since it is designed for parsing context free languages and ABNF is a context free language. An LALR parser is also much less complex than the more expressing LR(1) algorithm [1] and at the same time much more powerful then LR(0).

But there is a problem with implementing a LALR parser in hardware, which will can be described with the following example. An LALR parser can reach a state where multiple compares, jumps and look ahead data are required. When non-terminal symbols are encountered the amount of required operations is even worse. This leads to memory and resource allocation problems, due to the bursts of data. For example in one cycle the parser might only need one specific token to advance to the next state and in the next state it will require several compares, state information and look-ahead tokens. This can change again in the cycle after that. For hardware implementation such behavior is undesired. Also LALR requires one level of look-ahead and if real-time protocol parsing is required the LALR system will always lag behind a little.

A DFA [3] on the other hand is much more restrictive in terms of protocol expressiveness. To check how restrictive a DFA is, a survey was done based on the TCP/IP protocol. In this survey several types of fields were identified:

- Type/setting fields indicate which protocol is active or which settings are used. These fields can easily be processed with a both a LALR and a DFA parser.
- Address fields are compares against variable data. For instance a computer connected with ethernet can get an arbitrary IP address, which is only

known at runtime. The parser should have an option to compare this variable input data.

- Context sensitive fields like length and checksum fields are more of a problem. They cannot be parsed with a DFA, but a LALR parser cannot parse them either.
- Data fields are of no concern, a parser just needs to accept them and transfer them to the memory.

The data analysed from the TCP/IP protocol also has the advantage of being regular. All internet protocols that were checked and some simple protocols like SPI and I2C can be parsed with a DFA. It turns out that a DFA is a better candidate for implementation 3.1, because it is less complex and it is suited for the most important protocols today (the internet protocols). The LALR parser is more powerful, but also more complex. And although the LALR parser can handle more protocols, the protocols of focus can also be implemented with the much less complex DFA algorithm. That is why the decision was made to implement a parser with a DFA. To support all functionality of ABNF, some extensions were added to the DFA algorithm. The extra functionality and the requirements for this functionality are discussed in Chapter 4.

	LALR	**DFA**
Exposure	6	**3**
Impact	10	**10**
Risk	60	**30**

Table 3.1: Risk analysis for parser algorithm

3.4 PPU memory

After compilation a protocol specification is ready to be loaded into the memory of the PPU, which holds the protocol implementation. The next step of the design space is this memory. This memory has 2 aspects which determine its properties, the instruction set and the memory layout. These two properties are closely related to one another. For the memory system the parsing algorithm is important, a LALR would have different requirements than the DFA algorithm. Since in Section 3.3 it was determined that a DFA is going to be implemented, the task is to have a compatible memory system for the DFA architecture.

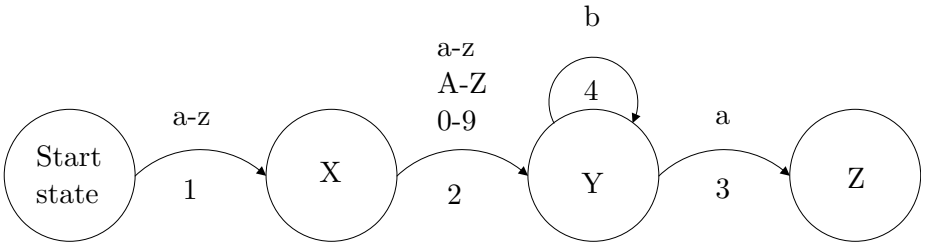


Figure 3.5: Basic DFA example

In Figure 3.5 an example state machine is given. Transition '1' is a simple "match everything between and including a-z". Translated to hardware operations this becomes: compare greater-equal 'a' and compare lesser-equal 'z'. If both statements are true go to state 'X'. These operations can either be performed in parallel or sequential.

Parallel vs sequential execution

Parallel execution has much better performance, especially when looking at transition '2'. Here multiple ranged matches require 6 operations, which could all be performed in parallel.

With branches in the protocol, performance differences are less obvious. In state 'Y' transitions '3' and '4' both have a different end state. Depending on the input data and the execution sequence, the performance can be the same as the parallel solution, or 50% slower.

Parallel computation is much faster compared to sequential computation, but parallel computation has its limits. For each compare a comparator is required. If a state has for example 40 different branches one would need 40 comparators, but then many comparators will often be idle. And if suddenly a new protocol is implemented and 41 compares are required the system would still take multiple cycles. A limit must be set if parallel processing is implemented, but the theoretical maximum throughput is increased by a factor of the number of comparators. The problem is providing all these comparators with data.

Memory architectures

The first solution for a memory architecture that is capable of suppling multiple comparators with data is to construct a memory with a very wide bus. On this bus data for all comparators is present, eg the opcodes and the compare data. For 8 comparators with a word size of 32 bits, this would result in a 256 bit memory bus for the comparator data alone, not including opcodes. While this is not a big problem, it is an inefficient method (in terms of memory utilization) if only 1 compare is required. Only 32 out of 256 bits would have any meaningful data.

The second solution was to use a variable instruction length approach. Some overhead would be introduced due to the variable data size, but no memory

space would be wasted. The performance of a variable instruction length architecture is equal to that of the fixed word length solution. The downside is that such an architecture requires a very large memory bandwidth (multiple instructions must be pre-fetched), and the instruction decoder for variable length instructions is very large and complex. If all instructions would be equal in size the problem would not be very big, but some instructions only need an 8 bit opcode, while others need the 8 bit opcode with a 32 bit parameter. Also a field indicating the length of the instructions is required, introducing another offset. This complicates things a lot. A variable width architecture that can support all of these instructions will consume a lot of power [5] compared to a sequential or fixed parallel design.

The last solution was the sequential parsing architecture. Although the performance would be significantly less than parallel parsing implementations, it would have a smaller hardware size, lower power consumption and an efficient memory usage. The hardware would also look more like a conventional processor, which might be an advantage when programming the device. In Table 3.2 the different solutions are listed together and in Table 3.3 the instruction configurations for the different architectures is given.

	Fixed width parallel	Variable width	sequential
Performance	+	+	-
Hardware size	+/-	-	+
Memory efficiency	-	+	+

Table 3.2: Architecture designs

Method	Instruction composition
Fixed sequential	opcode + parameter
Fixed parallel	opcode + parameter, opcode + parameter ...
Variable parallel	instruction size, opcode + parameter, opcode + ...

Table 3.3: Instruction sizes

The fixed width parallel solution has too many drawbacks to be a viable candidate, so the choice for the PPU architecture is between the sequential and the variable instruction width architecture. Although the power consumption is higher when implementing a variable instruction length decoder, the total energy consumption might be better off. Unfortunately there was not enough time to evaluate this. The hardware size and complexity was the key factor in determining the architecture of choice. Since a variable length instruction decoder is very complex, the more basic design of the sequential architecture was chosen. The limited amount of time available during the thesis work was also a reason to choose for the more basic design 3.4.

	sequential	parallel
Exposure	**2**	8
Impact	**10**	10
Risk	**20**	80

Table 3.4: Risk analysis for the memory

The sequential architecture determines the memory lay-out and the compatible instruction set. The instruction set has to support all the operations listed in Section 3.1. In Appendix 7.1 the instruction set is presented.

3.5 System interface

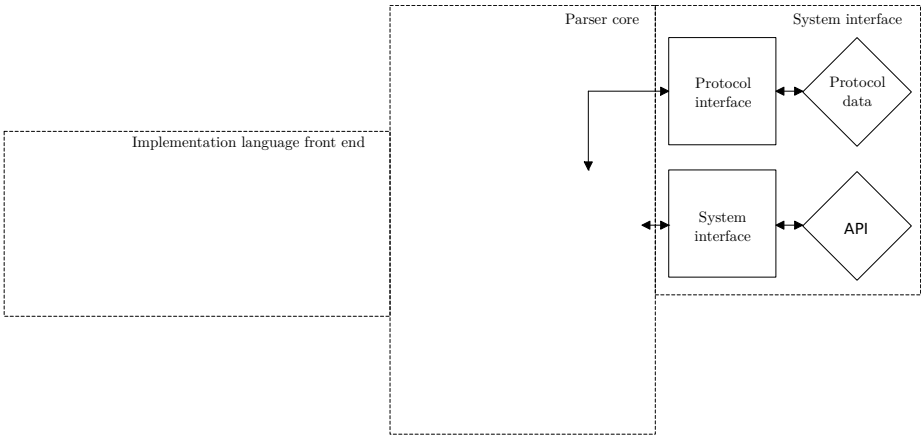


Figure 3.6: IO part of the PPU design

In the previous sections a large part of the PPU has been determined. In this section and the remaining parts will be examined, the input and output of the system. The input and output of the PPU are divided into a system interface for communication with other hardware and IO interfaces for protocol input/output. This section describes the IO for the system interface.

The PPU is in the domain of accelerators. For connection with other systems an interface is required for data exchange between the PPU and other processors. However, a single universal bus for communication between different processors does not exist. The system interface should be designed as such that it can easily be interchanged with another module. This allows the PPU to be integrated into different platforms, by only changing the system interface block. The main disadvantage of such a system is that the API requires adjustment each time it is applied to a different platform. To circumvent this problem a higher level API is required. Building a higher level API in which the lower level routines are automatically adjusted for each system bus will enable high level programmers to use a unified API that can be adapted to the different environments.

3.6 Protocol interfaces

For the protocol data an interface module is required. Some interfaces have single ended signaling, but others have differential, open drain, or other bus signaling. To connect all these different types of interfaces to the PPU a conversion module is required to translate these bus signals to signals for the PPU.

Two possible solutions were investigated, direct attached IO or with separate IO modules. In the direct attach method all IO signals would be connected

directly to the PPU. Only a small piece of hardware to translate interface signals to internal signals is required. This would allow the PPU to be very flexible, even multiple interfaces on the same IO pins could be implemented with this method. However, the load on the PPU core will be very high in such an architecture. Besides parsing the PPU would also become in charge of signal level reading. For instance with a simple RS-232 connection, the core would need to keep track of time, signal levels and signal level change, just to be able to decode the signals on the bus.

The other method is to use separate IO modules which can translate and buffer incoming data from connected interfaces and transfer this data to the PPU. The advantage of this method is that a lot of overhead interface signaling is handled by the specific modules. Verification if an interface is compliant with a standard is easier, because software delays are no longer an issue. The mayor downside is flexibility, with software controlled IO pins pretty much any protocol could be connected to any pin configuration. With this method the supported interfaces are fixed to the ones embedded in the module and the pin assignment is usually also static (a pin mux/demux could be used for re-arrangeable pin assignment).

	direct attached	**separate modules**
Exposure	7	**4**
Impact	4	**4**
Risk	28	**16**

Table 3.5: Risk analysis for IO interface

A combination of the two modules is also possible by using a dedicated processor for the IO decoding. However, verifying if the module meets an interface specification would still be a problem, especially for high speed interfaces. All things considered (see Table 3.5), the architecture with separate modules for the protocol interfaces proved to be the best option for implementation in the PPU.

3.7 Conclusion

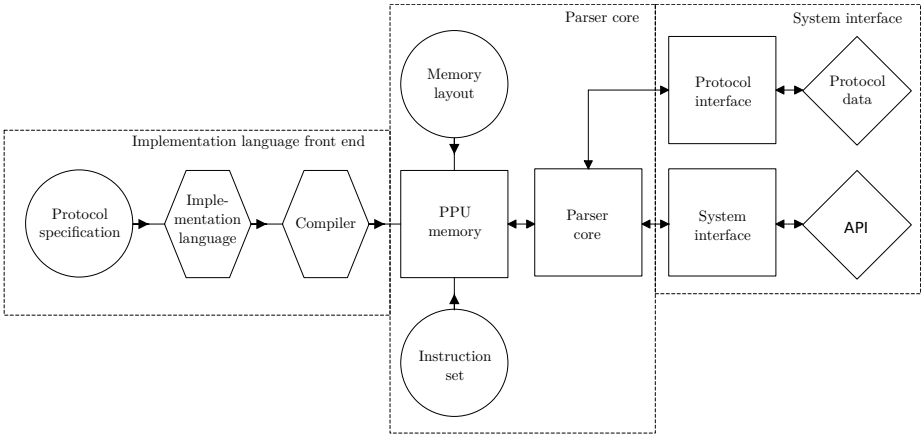


Figure 3.7: The PPU design

For the hardware design of the PPU a lot of different aspects were evaluated and decisions were made for the implementation. Figure 3.7 shows the global design used to specify the system. The implementation language is the bridge between the compiler and the protocol specification. ABNF is the language chosen for the implementation language. The language has the right properties for a grammar based parser, but it will require some extensions to support protocols. The focus for the PPU will be protocols that have a regular grammar.

The implementation language is converted into byte-code by a compiler. However, the compiler could not be finished and is left as a future project. During this thesis work the task of the compiler was done by hand.

The parser core is where the protocols are actually processed. The protocol grammar is loaded into the memory of the PPU. The memory architecture is a single issue memory (one instruction per clock cycle). The parser core loads the protocol specification from the memory and compares it with input from the protocol interfaces. For parsing, the core has the following instructions available:

- Single value match
- Ranged value match
- Alternative value match
- Branches (delayed)
- Defined repetition (delayed)
- Store
- Transmit

The IO of the PPU is handled by two modules, the protocol interface modules and the system interface module. The protocol IO is handled by separate IO

modules for each interface. The modules translate interface signal levels into signal levels for the PPU core. They also buffer incoming data if the PPU is busy. The system interface module is the connecting element between other processors (like a GPP) and the PPU. The system interface module will be designed such that it can be easily changed with another module. With this design the PPU can be made compatible with new systems with relative ease.

Chapter 4

Hardware design

In the previous chapter the design considerations of the hardware architecture were made. In this chapter the hardware for the chosen architecture will be described. Detailed design information will be presented and specific design choices for select components are described.

4.1 Hardware specification

Almost all hardware specifications are listed in 3.7. Here the most important specifications and modules for the hardware are repeated:

- DFA based parser core
- Memory for protocols
- Memory for variables
- Supported instructions:
  - Single value match
  - Ranged value match
  - Alternative value match
  - Branches (delayed)
  - Defined repetition (delayed)
  - Store
  - Transmit
- IO interface modules
- System interface module
- Multi protocol support

The multi protocol support was not discussed in the previous section. This was done on purpose, because it is more related to the hardware design, than the architectural design. However, some design considerations must be made, before it can be implemented. In Section 4.2 the multiple protocol processing

architecture is evaluated. In this section a system integration interface is also chosen. While it is discussed in the previous chapter a selection still has to be made for to implement one specific interface for the design that is implemented. In Section 4.3 the actual hardware implementation of the various parts of the PPU is described. After that section is a short recap on the compiler followed by a conclusion of this chapter.

4.2 Architecture design

In this section several possible architectures for multiple protocol processing are described. After the part on multiple protocol processing a section is devoted to the system integration. In this section a hardware platform is chosen which will be used for actual implementation of the PPU. This hardware platform will have a direct consequence for the system interface and it will be determined which interface is required.

4.2.1 Multiple protocol parsing

An important aspect of the PPU is the ability to parse multiple protocols simultaneously. Two methods for multiple processing were investigated:

- Thread switching, use time division multiplexing to process multiple protocols
- Multi core processing, use multiple parsing units to process multiple protocols

The main advantage of thread switching is that is requires a minimal hardware adjustment to incorporate it into a design. For thread switching the thread state must be saved and an arbiter is required to determine which thread is assigned processing time. The disadvantage of this method is that the total available processing resources are shared between protocols. If the processing requirements exceed the systems capabilities, data loss will occur on one or more protocols. To circumvent this problem the clock speed of the PPU cores can be increased. However, this cannot be done without consequence. If a certain clock speed is exceeded pipelining must be used to increase the clock speed further. When pipelining is used extra hardware is needed and a lot of care must be taken to divide the processing steps in equal parts.

Multi core processing is another method for allowing parallel processing. By replication the execution hardware for each attached protocol the processing power is easily increased and the performance is maintained. However, replicating the hardware increases the hardware size and the static power consumption.

For the PPU both multi core and thread switching are incorporated and pipelining is not (see Table 4.1. In the basic design, for each protocol a parse core was required. With thread switching a core is able to process multiple protocols on a single core. However it does not enhance the processing capabilities of the system. With many connected interfaces this will lead to performance issues. To increase the performance extra cores can be added. A dual core design doubles the peak processing power, compared to a single core system. By incorporating both methods designers can easily adjust the PPU to their design criteria. This makes the design flexible and ready for different processing environments.

	Multi core	**Thread switching**	Pipelining
Exposure	**5**	**3**	7
Impact	**2**	**4**	2
Risk	**10**	**12**	14

Table 4.1: Risk analysis for multiple protocol switching

4.2.2 System integration

The design of the system integration interface will be for the largest part independent from the rest of the PPU. This allows the PPU to be more easily adjusted if it needs to be implemented for another bus. To show the capabilities of the system the PPU will be designed with an interface for the AMBA 2 bus[2]. This bus was chosen because it is connected to the LEON3[10] processor. This processor is a soft-core which is free of charge for educational goals and experience with this soft-core was at hand.

4.3 Hardware implementation

In this section the implementation of the architecture is discussed. In Figure 4.1 the top level overview of the system is shown. This global structure is used to describe all parts and components in detail in the following subsections. In the first part the PPU cores are described. In the modules seen in the figure, the protocol data is processed. The IO interfaces are discussed in the next part. To ensure compatibilty of the IO interface with the PPU cores, the IO interface properties need to be determined first. Both the PPU cores and the IO interfaces are controlled by the arbiter. The arbiter governs the data flow inside the PPU. It controls muxes and enables parts required for parsing. The last element that is discussed is the AMBA interface. This unit is required for system integration. It connects the PPU to a host processor and allows programming of the PPU together with data transfer.

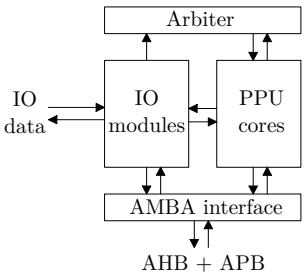


Figure 4.1: High level architecture

4.3.1 PPU core

The PPU core is an implementation of a programmable sequential parsing DFA (see section 3.3). A DFA drawn in figure 4.2 is used to illustrate the basic requirements the core must be able to process. The transition from state '0' to '1' requires the match of the character 'a'. Thus a comparator is required to

match the input and a state controller is required to keep track of the state the DFA is currently in. Most state transitions are from one state to the next, to simplify the design the state controller will automatically increment to the next state and these transitions do not have to be programmed.

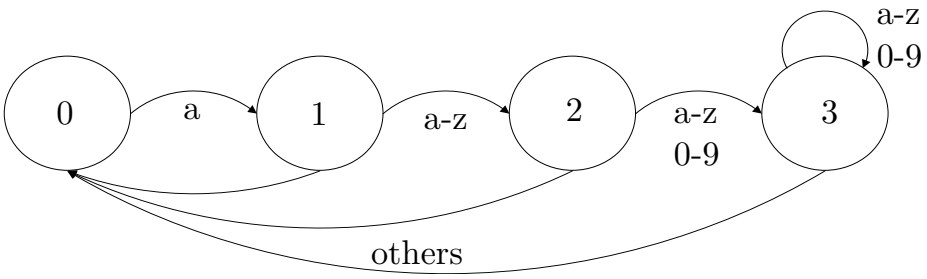


Figure 4.2: Basic state machine diagram

For the second transition a ranged match is required. The value should be equal or greater than 'a' and less or equal to 'z'. In the sequential machine these 2 operations cannot be computed simultaneously (because of the memory bottleneck), thus and intermediate state is required. First equal or greater than 'a' must be matched, followed by equal or less than 'z' (or the other way around). This could easily be done by introducing an intermediate state, however, would not work with the 3rd transition without introducing complex extra states. For these transitions the state controller must remain in the same state, and wait until the comparator has executed all the required operations. After all the required operations are completed, the state controller can decide if the transition requirements are met and transition to the next state.

In the last state the state machine should stay in the same state, this conflicts with the auto increment functionality of the state controller. For this purpose an extra state needs to be added. In this fourth state a non conditional jump is made to the third state.

There is one last functionality required for this DFA to work. If data is fed into the machine that does not match any transition in the current state, the DFA should return to its start state.

State transition implementation

The core functionality is designed so it can easily be mapped to hardware. A state is equal to a memory address, each address contains one instruction and the related parameter. The instruction field is 8 bits and the parameter field is 32 bits. The output of the memory is fed into an instruction decoder, which will decode an instruction and drive the comparator and the state controller. The comparator receives input data and the parameter to be matched and outputs the result to the state controller. The basic core is shown in Figure 4.3. A short description of the program of the DFA in Figure 4.2 is used to provide a more detailed look into the operation of the PPU core.

- When activated the state controller loads the start state and sets the memory to the start state address.

- The memory outputs the first instruction, compare equal, and the parameter value ‘a’
- The instruction decoder flags a single match to the state controller and sets the operation of the comparator to “equal match” and the parameter value ‘a’
- The comparator will compare the input data with the parameter and signal the state controller if a match or a mismatch has occurred.
- If the state controller receives a mismatch the next state will be the start state, else the state controller will increment the protocol memory address by one and the cycle repeats.

For ranged compares the sequence is a little bit different. The first operation is a multiple compare instruction, this signals the state controller that a ranged compare is coming. In this instruction the comparator is fed dummy data and it does not load new input data. The next instruction looks like a normal compare instruction, but the result is stored in the state controller. All other compares are executed (up to a maximum of 8) and after all the compares are executed the state controller decides if a match has occurred or not. If more than 8 compare are needed, the selection must be broken down in several states of 8 compares, this can be done by the compiler.

Jumps are also implemented with a special instruction. When a jump instruction is encountered the state controller is flagged that a jump is executed and a jump vector is supplied to the state controller (the jump vector is the parameter of the jump instruction). Instead of incrementing the state address the jump vector will be loaded.

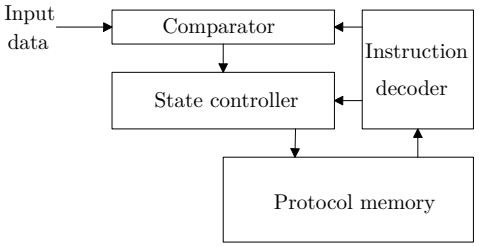


Figure 4.3: Basic PPU core

To support more than one protocol per PPU core, each protocol state must be stored. This is done in a special memory for this purpose. This memory is addressed by the arbiter (which will be discussed in subsection 4.3.3) and used to retrieve the correct state information. When the state controller is activated, it will require 1 clock cycle to load the correct state address and address for the protocol memory. When a protocol is deactivated the state controller needs another cycle to store the state information. This causes an overhead of 2 clock cycles every time a switch between interfaces takes place.

With these added components the module is also ready for multi core implementation. For the multi core implementation to work, the state memory and the protocol memory need to be shared between cores. This means that the state memory needs to be a dual port memory when using 2 cores and a

higher order port memory when using more cores. Multi port memory is required, because a protocol state needs to be coherent between PPU cores. Since the protocol memory is basically a read only memory (it is only written during programming), duplicate single port memories or a multi port memory can be used for the protocol memory. With the state memory included the PPU core schematic now looks like fig 4.4.

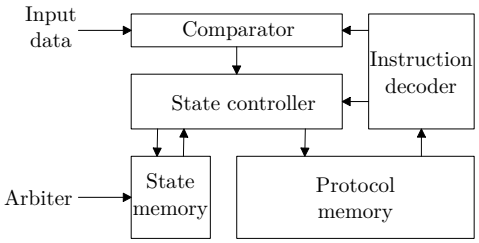


Figure 4.4: PPU core with multi protocol support

Core instructions

With the state memory the PPU core is not yet complete, but the overall structure is nearly finished. Four classes of instructions are still required for a functional model. Delayed branches, repetition, and transmission instructions are still needed. Delayed branches are implemented in almost the same manner as direct branches, the only difference is that there is a separate branch setup instruction (with the branch address) and an extra instruction for branch execution. The branch address is stored together in the state memory until a branch is executed.

Repetition works slightly different, first a setup instruction is executed which loads the repetition count, which is also stored in the state memory. When a repetition start instruction is executed, the current address is stored too. The PPU continues normal function until a repetition stop instruction is encountered. The repetition stop instruction will load the address of the repetition start instruction. But when the repetition count is zero, the state controller will increment the address and load the next instruction.

The transmit instruction loads the parameter from the protocol memory and outputs them to the IO modules. Also a flag is set to signal the IO modules that transmit data is coming and to signal the state controller to increment the state address.

One last instruction that is required is the store enable/disable instruction. This is used to mark data that needs to be transferred to the host processor. By marking data, not all data needs to be stored in the main memory, reducing the required memory space and bandwidth.

The last part added to the PPU core is a register memory. This multiport memory is used to store runtime variables. The compare, transmit and repetition instructions can work from this memory. Data can be written by the host processor and incoming protocol data can also be stored in that memory. Read memory access is controlled by the instruction decoder and write access is controlled by the state controller.

With these last adjustments the PPU core is completed and the resulting schematic looks like fig 4.5. In this figure the programming interface is not

shown, this is for clarity of the figure. The instruction-set is listed in appendix 7.1.

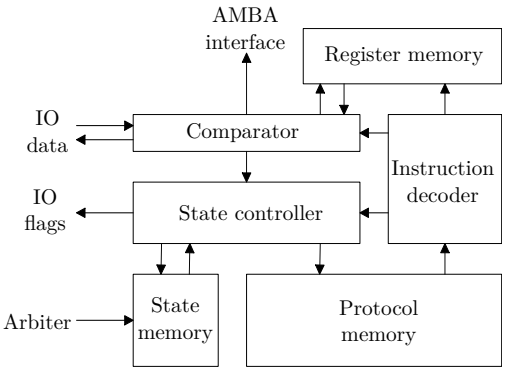


Figure 4.5: Block diagram of implemented PPU core

With all the elements in place the PPU core is capable of processing most DFA protocols. The limit for DFA protocols is their size. The combined size of all implemented protocols must fit inside the protocol memory. If the protocols exceed the size of this memory there is no support for fast swapping of the protocol memory contents (it does not work like a cache). This may pose a problem when implementing very large or a large number of protocols.

4.3.2 IO interfaces

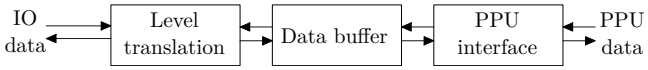


Figure 4.6: IO module overview

The IO interfaces are responsible for the IO of the protocol interfaces. Each interface is connected to its own module. Each module manages the IO level translation and buffers incoming data before it is send to the PPU. The basic architecture of the IO modules is shown in Figure 4.6.

An important function of the IO modules is to provide the arbiter (see Section 4.3.3) with information of processing requirement. The IO modules indicate if they require processing. If they do, they will notify the arbiter and wait until they are granted processing time. Basically the PPU is a slave to the IO modules.

The data transmission handshake is fairly easy (see Figure 4.7). The interface sets a ready signal to notify the PPU core that it is ready to receive new data. The PPU core will assert a TX signal indicating the transmitted data on the data bus is valid. The IO module can now load the data and transmit it over its protocol interface.

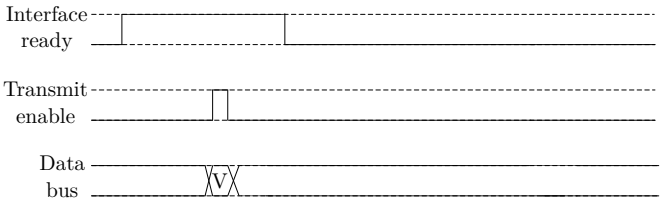


Figure 4.7: Transmit operation waveform

Reception of data is different (see Figure 4.8). When the IO module has received data after turn on, it can assert the ready signal. The PPU core will respond by asserting the newData signal and setting the dataLength bus. If the IO module does not have the requested amount of bits ready, it will send dummy data and de-assert the ready signal. The next clock cycle the PPU core will repeat the dataLength signal and the module can wait until it has sufficient data. When it has sufficient data it can assert the ready signal again and transfer the data to the PPU core. The data sizes that can be requested by the PPU cores are : 1,2,3,4,8,16,24 or 32 bits of data. These numbers are chosen such that with a maximum of 3 instructions all bit sizes between 1 and 32 bits can be received. Not every module needs to support all these data sizes, but the protocol implementation should take this into account.

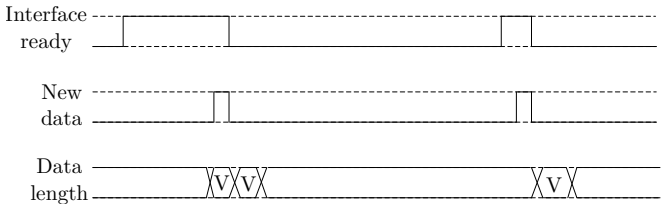


Figure 4.8: Receive operation waveform

The last requirement for an IO module is buffering. Every module needs some buffer to allow some delay between receiving data and transfer to the PPU. The size of the buffer is depending on speed of the interface and the data rate, a fully utilized ethernet connection will require more buffering than an I2C connection which is only used every now and then.

One limitation of the IO modules is that they do not support interrupts. While it is not difficult to implement this, a problem is connecting the interrupts to the output. Because different system busses can be connected to the PPU a good universal way to handle interrupts was not implemented.

4.3.3 Arbiter

The arbiter is used to assign interfaces to a PPU core and controls the multi core and the multi threading support (by time division multiplexing). The multi threading support allows multiple protocols to run on a single PPU core, as long as the combined data rate of the protocols does not exceed the bandwidth of the PPU core. If it does require more processing power, or if two protocols are active at the same time, the arbiter will utilize more cores (if available).

Arbitration algorithm

In the subsection on the IO module, it was stated that the IO modules request processing time if they need it. The request can change every clock cycle, therefore every clock cycle the arbiter computes for each available PPU core if an interface can be assigned to a core. From this requirement it can be concluded that the algorithm implemented for the selection of an interface must be fast (linear or loglinear time). The algorithm must also be able to scale, because multiple PPU cores and interfaces can be connected to it.

Several algorithms were evaluated, but the wavefront algorithm [12] (an algorithm designed for matrix switches) was the most suitable. This algorithm finds a solution in linear time and scales linear. It is both fast and flexible and it requires only basic gates, making it small and cheap. The downside of this algorithm is that every interface will be assigned a fixed priority, determined during synthesis. The algorithm implementation could be adjusted for dynamic priority selection, but this would require at least double the hardware and the latency.

The wavefront algorithm output is one-hot encoded. If this output is used in the PPU, a large and wide bus is required throughout the device. To reduce the bus width a one-hot-to-binary converter is added. The drawback of this converter is that it adds an extra delay and the output of the converter is 0 when no devices are requesting data, but also when the first entry is. Thus the value 0 cannot be used anyone as an interface index.

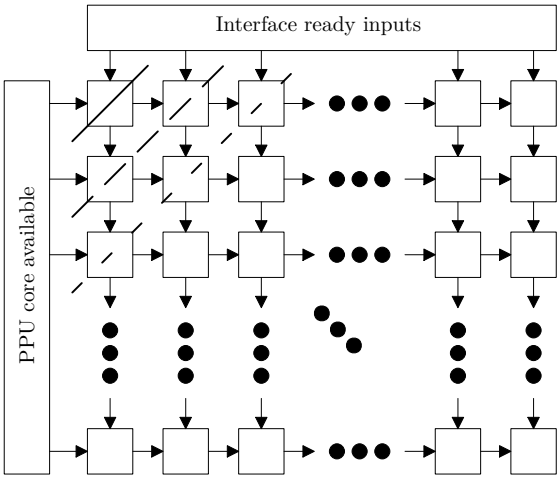


Figure 4.9: Wavefront algorithm

Wavefront algorithm

The wavefront algorithm is explained in Figure 4.9. The algorithm first checks to see if column 0 (highest priority interface) requires attention and if row 0 (first PPU core) is available (solid line). If the interface requires attention and the core is available then the two will be connected and the row will be closed. This means no other interface can be connected to the first PPU core. When the first core is already in use, or the first interface does not require processing,

column 0 and row 1 and column 1 and row 0 are compared (dashed line). The next step is column 0 and row 2, column 1 and row 1 and column 2 and row 0. This continues until column N and row M have been processed. The algorithm has a time complexity of O(N+M), but it could be improved to O(N).

The improvement is done by using a wrapped wavefront algorithm and although it is faster, it will also be more complex to implement and the priorities are not very clear any more. Also because the number of PPU cores will likely be much lower than the number of attached interfaces. The speed gain obtained with the wrapped wavefront algorithm will likely be minimal.

4.3.4 AMBA interface

The last part required to make it all work is an interface to a general purpose processor. All data flagged by the PPU for storage needs to be stored somewhere. An interface to the main memory is needed for storing data. Also the PPU needs an interface to program the memories and settings. To accommodate these requirements an AMBA 2 [2] interface has been added to the PPU. On the AMBA interface the Advanced High-performance Bus (AHB) is used for a DMA interface. The Advanced Peripheral Bus (APB) is used for programming the PPU.

DMA

The DMA unit is used to transfer all incoming data from a protocol to the system memory. To separate the data of different protocols, the user can program the memory range where the DMA unit may store its data per interface. Five registers are available for this purpose, “Read start” and “Num words” are used for read actions, “read start” indicates the address from where data will be read, “num words” indicates how many consecutive locations from the address are read.

“Write start”, “write current” and “write end” are used as the cyclic memory boundaries for DMA write actions. The DMA unit will start writing data to the “write start” address. After each write the “write current” address is incremented and to the location of this address data will be written. If the end address “write end” is reached the next write will be the start address. If a single memory location is used all three addresses are the same. The DMA unit does not have any flow control for the memory write locations. The software on the general purpose processor must read the data before it is overwritten.

Programming

The PPU is programmed over the APB interface. This low speed interface is sufficient for programming the device. The programming of the PPU is different from normal APB devices. Normal APB devices are programmed with memory mapped registers, with each register on a specific memory location. The memory and registers of the PPU are not memory mapped to the APB. This was not possible, because the size of the memory, the registers and the number of attached peripherals can change depending on the configuration.

To program the PPU a custom command sequence is required at the base address of the PPU as depicted in Figure 4.10. This allows each module to have

a specific programming sequence. For instance the protocol memory requires a size command, then an address command followed by a sequence of instruction and parameter data. The sequence for programming can be different for each attached module and that is why this implementation was chosen.

Aside from the base address the remaining address space of the APB device is used to program the AHB DMA controller. Each interface is assigned 8 32 bit registers. In 5 of these registers the DMA configuration is stored (see Section 4.3.4), the other 3 are reserved. This allows a maximum of only 7 interface devices attached to one PPU unit, but this can be expanded if required by adjusting the APB device address space. However, for the first version of the PPU support for 7 interfaces was sufficient.

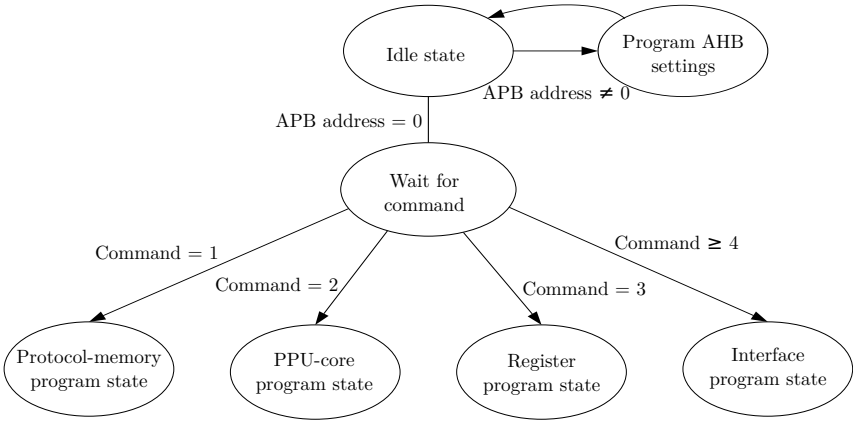


Figure 4.10: Program flow of program-interface

4.4 Compiler revisited

After the hardware design the compiler could be revisited. First a start was made to make some basic rewrite rules for a compiler. These rewrite rules are listed in this section, unfortunately further development of the compiler has not taken place due to lack of time. Single value matches, data store start and stop instructions and unconditional jumps can be rewritten directly to byte code. Ranged matches and conditional jumps require some rewrite rules. Ranged matches (up to 4 ranges) require the following construct:

```
Original:
  <ASCII> = a-z
  <ASCII> =/ A-Z
  <ASCII> =/ 0-9
Rewritten as:
  <ASCII> = <multipleCompare> <firstValue>
                <otherValues> ..... <otherValues>
```

Branches can have a maximum of 8 different branch locations. If more than 8 different branch values are required a selection decomposition must take place.

For instance an 8 bit word can be broken up into a 3 bit and a 5 bit sequence. The 3 bit sequence can be evaluated first, followed by the remaining 5 bits (which can in turn be broken up if it is required). Another option is to load the 8 bit value and program ranges. This allows the compiler to use binary search (with 8 options at a time) to find the correct branch address.

```
Original:
  <ethertype> = <IP> / <ICMP> / <IPsec>
Rewritten as:
  <ethertype> = <multipleCompare> <firstValue>
                    <secondValue>....<lastValue>
                    <firstBranch> <secondBranch>
                    ...<lastBranch>
```

Repetition also needs to be rewritten.

```
Original:
  <repetition> = <X>*<X><repetitionData>
Rewritten as:
  <repetition> = <repetitionStart> <repetitionData>
                      <repetitionEnd>
```

Something that has also got to be considered is the limitation of certain IO modules. Some modules only support specific data sizes (see 4.3.2). With options the compiler must be notified which protocols support which data sizes. The compiler specification is far from complete. A lot of work is needed in this area to make a working compiler, but some basic design decisions have been made.

4.5 Conclusion

One of the core features of the design of the PPU is flexibility. The goal of the project was to create a hardware module that can process a wide variety of protocols like a grammar. The hardware design reflects this idea.

Protocol interface modules are required to connect physical interfaces with the PPU. The design and connection for these interface is made such that the number of connected interfaces can easily be scaled. To cope with a large number of interfaces multiple parser cores can be added to increase performance of the system. This is also designed to scale.

The main bottleneck for large systems will be the system interface unit. However the limitation of this interface is very dependent on the system it is integrated on. For this first design the PPU is connected to the AMBA 2 interface. But also here flexibility was important, the system interface unit is designed such that it can easily be replaced by another interface, to allow the PPU to be used in very different systems.

Chapter 5

Results

After the hardware design the hardware was tested. To test the PPU a few protocols were selected to be implemented. The selected protocols are:

- I^2C
- SPI
- ethernet
- PWM

For these four protocols an IO modules was designed and first tested in simulations (see 5.1). After the simulations the hardware was implemented on a FPGA and measured (see 5.2). With the data from both the simulations and the hardware implementation a conclusion can be made about the PPU in 5.3.

5.1 Simulation

During the implementation of the PPU a lot of simulations were made to test and verify the functionality of the PPU. Every module is tested separately with its own testbench. After modules were verified they were connected to test the functionality of the PPU core, and the AMBA interface combined with the PPU core. Final tests were done with each interface module connected to the complete PPU.

The LEON3 processor was not used in simulations. This restricted the simulations somewhat, but even without the LEON3 many simulations could be run. The largest problem was testing larger protocols, because they could not be loaded from software, thus they had to be written in byte-code entirely by hand.

For each of the 4 implemented protocols extensive testbenches were created. For PWM, I^2C and SPI this was not a problem. Ethernet with the TCP/IP stack was another matter. Because ethernet is very complex a testbench where many use cases were tested was not feasible. Therefor for ethernet only a limited number of tests were written, excluding many aspects of TCP/IP. However, the results from the simulation proved that the design works, and gives and estimation of the performance of the system.

5.2 Hardware implementation

After the simulation the four protocol modules were implemented in hardware. With the hardware implementation several tests were conducted to evaluate the performance of the PPU with different protocols. For each protocol the load on the CPU was compared between the PPU and the CPU load. The load is measured in C instructions for the CPU. All C code lines were counted as a single instruction, except for for loops which were counted as 3 instructions and function calls are not counted as they can be in-lined. The implemented protocols and their results are described in detail below.

5.2.1 PWM results

While PWM [23] is not a communication protocol, it was used as an early test module. It also shows that the PPU can be used for different functions other than pure protocols. For the PWM test a very basic protocol was implemented. The PWM module has to generate a 16 bit sine wave PWM with 16 steps. This protocol was compared to a software solution with a standard PWM peripheral.

A GPP needs to program the protocol into the PPU memory. This takes $3 + 5 * protocolSize$ C instructions. The protocol size for the PWM protocol is 34, so 173 instructions are required for programming. Another 5 instructions are required for PPU core setup and the interface setup. For the PPU implementation a total of 178 C instructions are required to implement this protocol. After this initialization, no more attention for the GPP is required.

For the software solution with a dedicated PWM module the load is very different. To program a standard PWM peripheral somewhere in the order of 5 settings need to be programmed [13]. After this initialization a PWM values needs to be programmed after each completed PWM cycle. This will only take one instruction (not including polling or interrupt overhead). This method clearly requires less setup instructions, a PWM signal can be generated with only 6 instructions. However, as time progresses the software solution has to keep updating the PWM value. The break even point is after 173 PWM values (almost 11 sine wave cycles). When more than 173 values are required the software solution poses a higher overall load on the GPP than the PPU would. The software solution has the advantage that the values of the PWM cycle can be changed more easily, but if this happen less often than once per 11 cycles, the software solution still has a lower performance compared to the PPU solution.

Another thing that was learned from this test is that a simple protocol is not very efficient in terms of PPU core load. The PWM module receives a transmit operation and will remain idle until the next PWM cycle starts. But for every operation a start and stop cycle are needed. For the entire protocol the efficiency measured in useful cycles of the PPU core is only 1/3.

5.2.2 SPI results

SPI [24] was the first real communications protocol implemented in the PPU. In the SPI protocol the role of the PPU was to be a slave. The protocol and hardware for a slave device are less complex than a master device, which is why a slave function was chosen as a first communication test protocol.

Protocol description

The protocol that was implemented in SPI was a very simple 'toUpper' protocol. The PPU was programmed only to accept "hello" as input string. The output would be "HELLO" with 1 character delay between input and output. While this was a very simple protocol it would test the receive, transmit and branch operations of the PPU.

A second test was also done with the SPI protocol to send variable length data. In this protocol a set of random data bytes was send, all with a different length. This was used to test the ability of the PPU to receive variable width data. No load measurement was done with this protocol, but data lengths of 1-32 bits were tested, and they could all be parsed without problem with at most 3 clock cycles.

Load measurement

The software load for a toUpper program is very low, but again in time the PPU will be more efficient. To run a similar program on a micro controller only 5 instructions are needed for setup. The runtime load is much higher than of the PWM test, not including polling or interrupt overhead. The software needs to perform a compare on the received data, load the next transmit data and increment to the next data point. After incrementing to the next data point, it has to check if the end is reached and loop back. The total load in C is 7 instructions per correctly received byte and 3 instructions per incorrectly received byte.

The PPU fares better, it has an initial higher load of 63 instructions for setup, but afterwards no CPU cycles are needed anymore. This means that after only 9 bytes of correct data or 21 bytes of incorrect data the PPU is more efficient compared to the software solution.

An advantage of the PPU is that the load of both the SPI and the PWM protocol is very low, and that both protocols can run together without any problem.

5.2.3 I^2C results

I^2C [15] is the third protocol tested on the PPU. For this protocol a master module was implemented. I2C is a protocol with more checks and timing issues than SPI has.

Protocol description

The protocol on the I^2C connection is a little more complex, a connection is initiated by the PPU and the address for the slave is send. After the slave acknowledges the address command, 5 bytes are written from registers. Then the PPU stops and restarts communication, again an address is written, but this time with the read command. The bytes read from the bus and then stored in the registers and the cycle starts over.

Load measurement

The difference between this protocol and software is just like the other two. On initialization it requires much more CPU cycles, but during runtime it is much more efficient. The software takes 5 cycles for each byte (send or receive) and 3 more for each acknowledgement. To load the protocol into the PPU memory takes 123 cycles. After cycling through the protocol twice the PPU implementation is already more efficient.

5.2.4 Ethernet results

Ethernet together with the TCP/IP stack is one of the most used protocols today. It is also one of the most complex ones to implement. For an ethernet connection a separate PHY IC is required to translate the differential ethernet signals to single ended signals for the FPGA. The connection between the FPGA and the ethernet PHY is a GMII connection. This connection has its own protocol so basically the PPU handles GMII as low level protocol layer instead of ethernet. Fortunately this makes no difference for the PPU.

Performance

For ethernet an ARP request and an ICMP (ping) reply was implemented. However, comparing the performance of these protocols between a software and a PPU implementation is very difficult. In the software implementation a full TCP/IP stack is present. The full stack has many options that are checked where the PPU implementation does not. The reason that the PPU implementation does not have the full stack, is time. Implementing a full TCP/IP stack takes a huge amount of work and there simply was not enough time to do it properly.

However, an estimation was made based on the results of the tests run. Software can parse about 120000 ping packets on a 2.66 GHz Pentium-4 processor [11]. Based on the measurements done on the PPU a comparison is made how many packet could be parsed using the PPU. By using the same packet size as in the paper, running a simulation yielded a 108 cycles per packet receive load. This load is calculated with 24 supported ethertype fields, 24 IP protocols and 16 ICMP types. The only assumption made (which was not included in the simulation), is the inclusion of a checksum offload module.

Based on this testbench the PPU requires 140 clock cycles to parse an incoming packet and send the reply. The Pentium-4 processor requires 22166 cycles per receive and response (on average). The PPU is much more efficient, but a Pentium-4 processor is a very old processor. A modern high end Intel Core i7-4790K processor performs about 44 times better [16]. The performance of a modern CPU would be around 515 cycles for parsing a ping packet and sending the reply. This would still favor the PPU as it requires about 3.5 timer less clock cycles for the same amount of work.

But the absolute performance of the Intel processor is much higher, since the PPU runs at a much lower clock frequency. Also when no offload module for the checksum is available, the performance of the system would reduce very much. This is because of two factors. The first and obvious one is that the GPP in the system has to calculate the checksum twice (send and receive) before the reply

can be transmitted. The second is because all data is required by the GPP for calculating the checksum. When the checksum is offloaded no data transfer has to take place, increasing the performance.

5.3 Conclusion

Several protocols were implemented and tested on the PPU. While some tests were only done in simulation, most functionality was also tested on real hardware. From these tests the following general conclusion were made:

- The PPU can parse different protocols and process them simultaneously.
- The PPU requires a relatively large setup overhead, but when it is running the overhead is low and the performance is high.
- The PPU can work with data sizes from 1-32 bit without any problems.
- With a simple ICMP protocol the performance is 3.5 times higher than the performance of a modern CPU, measured in clock cycles per packet, assuming a hardware module for checksum offloading is available (which is implemented in many ethernet PHY devices already).

For a first version of a new concept the performance results are promising. The main performance bottleneck of the PPU is multiple branch calculations on select protocol fields. For example the ethertype field in an ethernet header can have many different values, each relevant for another protocol. Using multiple parallel compares would greatly increase the performance of the PPU.

The hardware size of the PPU is 1669 LUTs and 700 registers in an Xilinx Virtex-6 FPGA. This size is for a 2 core PPU design including a PWM, SPI, I^2C and a GMII (ethernet) module. Also a 2048x40 and a 1024x32 memory block is used for the PPU. Compared to a LEON3 processor (15,693 LUTs and 12814 registers) the PPU is a very small device.

The use of a grammar to describe a protocol was intuitive to use. With a protocol written in a grammar it is very easy to keep track in what state the protocol is, what has to be done next and to find errors. However, with only one person using this method it is hard to tell if other engineers would agree on this subject.

The biggest improvement that can be made for the current system is to design more hardware modules for various protocols. For each protocol a PHY module had to be created and most errors during implementation came from bugs in these hardware modules. For the PPU to be successful, many modules for many different interfaces should be available. If this would be possible the PPU can easily be used in new applications by just loading the correct modules into the library.

The compiler should also have a high priority for future work. The current version requires an engineer to program the PPU with byte-code. This takes a lot of time and it is error prone.

The last improvement for future work is better configuration and control of the PPU with the API. In the present state configuration registers can only be written and not read by the GPP. Some other functions for the GPP would also be welcome. For example it would be useful to allow the GPP to set the PPU

in a certain state and let it execute from the state. This is useful in for example a TCP/IP protocol where an "keep-alive" packet needs to be send every now and then, issued by the GPP. Also the lack of interrupts cause a load on the GPP for polling operations.

Chapter 6

Conclusion

The design of the PPU started by examining the current processing methods. Currently the 3 most used methods for parsing protocols are software parsing and hardware accelerated parsing, by means of an accelerator or with Content Addressable Memories. Software processing is the cheapest method with the fastest time to market, however, the performance of accelerators and CAMs is much better than software solutions (see Table 6.1). Software has another advantage and that is its flexibility. This together with the low cost make software solutions the most widely used option for protocol processing. Only when a large power or performance advantage can be achieved, accelerators or CAMs are used.

	Software	Accelerator	CAM
Hardware costs	++	-	- -
Development costs	-	+	+
Performance	- -	++	++
Flexibility	+	-	+
Time to market	+	-	+
Ease of integration	+	-	-

Table 6.1: Protocol processing solutions

The PPU is targeted between the accelerators and the software solutions. The goal is to achieve good performance and a very flexible processor based on parsing protocols like a grammar. This is the reference from which the design started.

Design considerations

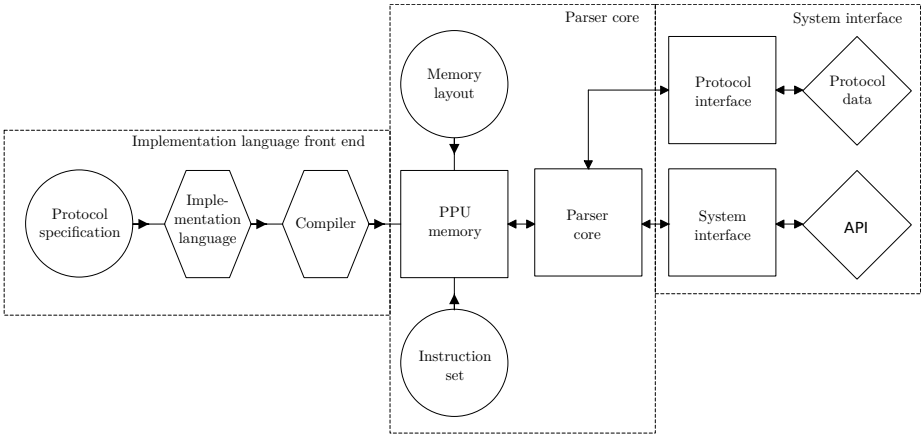


Figure 6.1: The PPU design

For the hardware design of the PPU a lot of different aspects were evaluated and decisions were made for the implementation. Figure 6.1 shows the global design used to specify the system. The implementation language is the bridge between the compiler and the protocol specification. ABNF is the language chosen for the implementation language. The language has the right properties for a grammar based parser, but it will require some extensions to support protocols. The focus for the PPU will be protocols that have a regular grammar.

The implementation language is converted into byte-code by a compiler. However, the compiler could not be finished and is left as a future project. During this thesis work the task of the compiler was done by hand.

The parser core is where the protocols are actually processed. The protocol grammar is loaded into the memory of the PPU. The memory architecture is a single issue memory (one instruction per clock cycle). The parser core loads the protocol specification from the memory and compares it with input from the protocol interfaces. For parsing, the core has the following instructions available:

- Single value match
- Ranged value match
- Alternative value match
- Branches (delayed)
- Defined repetition (delayed)
- Store
- Transmit

The IO of the PPU is handled by two modules, the protocol interface modules and the system interface module. The protocol IO is handled by separate IO

modules for each interface. The modules translate interface signal levels into signal levels for the PPU core. They also buffer incoming data if the PPU is busy. The system interface module is the connecting element between other processors (like a GPP) and the PPU. The system interface module will be designed such that it can be easily changed with another module. With this design the PPU can be made compatible with new systems with relative ease.

Hardware design

One of the core features of the design of the PPU is flexibility. The goal of the project was to create a hardware module that can process a wide variety of protocols like a grammar. The hardware design reflects this idea.

Protocol interface modules are required to connect physical interfaces with the PPU. The design and connection for these interface is made such that the number of connected interfaces can easily be scaled. To cope with a large number of interfaces multiple parser cores can be added to increase performance of the system. This is also designed to scale.

The main bottleneck for large systems will be the system interface unit. However the limitation of this interface is very dependent on the system it is integrated on. For this first design the PPU is connected to the AMBA 2 interface. But also here flexibility was important, the system interface unit is designed such that it can easily be replaced by another interface, to allow the PPU to be used in very different systems.

Results

Several protocols were implemented and tested on the PPU. While some tests were only done in simulation, most functionality was also tested on real hardware. From these tests the following general conclusion were made:

- The PPU can parse different protocols and process them simultaneously.
- The PPU requires a relatively large setup overhead, but when it is running the overhead is low and the performance is high.
- The PPU can work with data sizes from 1-32 bit without any problems.
- With a simple ICMP protocol the performance is 3.5 times higher than the performance of a modern CPU, measured in clock cycles per packet, assuming a hardware module for checksum offloading is available.

For a first version of a new concept the performance results are promising. The main performance bottleneck of the PPU is multiple branch calculations on select protocol fields. For example the ethertype field in an ethernet header can have many different values, each relevant for another protocol. Using multiple parallel compares would greatly increase the performance of the PPU.

The hardware size of the PPU is 1669 LUTs and 700 registers in an Xilinx Virtex-6 FPGA. This size is for a 2 core PPU design including a PWM, SPI, I^2C and a GMII (ethernet) module. Also a 2048x40 and a 1024x32 memory block is used for the PPU. Compared to a LEON3 processor (15,693 LUTs and 12814 registers) the PPU is a very small device.

The use of a grammar to describe a protocol was intuitive to use. With a protocol written in a grammar it is very easy to keep track in what state the protocol is, what has to be done next and to find errors. However with only one person using this method it is hard to tell if other engineers would agree on this subject.

The biggest improvement that can be made for the current system is to design more hardware modules for various protocols. For each protocol a PHY module had to be created and most errors during implementation came from bugs in these hardware modules. For the PPU to be successful, many modules for many different interfaces should be available. If this would be possible the PPU can easily be used in new applications by just loading the correct modules into the library.

The compiler should also have a high priority for future work. The current version requires an engineer to program the PPU with byte-code. This takes a lot of time and it is error prone.

The last improvement for future work is better configuration and control of the PPU with the API. In the present state configuration registers can only be written and not read by the GPP. Some other functions for the GPP would also be welcome. For example it would be useful to allow the GPP to set the PPU in a certain state and let it execute from the state. This is useful in for example a TCP/IP protocol where an "keep-alive" packet needs to be send every now and then, issued by the GPP. Also the lack of interrupts cause a load on the GPP for polling operations.

Review

Now the questions from the start of the thesis can be answered:

- Can a hardware module be designed that processes different communication protocols like a grammar?
  - Yes, it is possible.
- How does the performance of the hardware module compare to a software implementation?
  - The initialization poses a higher load on the GPP with a PPU, compare to an initialization without a PPU. During runtime the performance of the PPU is higher (up to 3.5 X for ICMP) and the overhead is lower.
- What is the cost of such a hardware module?
  - 1669 LUTs and 700 registers and a block of memory. The cost of this hardware is very low for modern IC processes.
- What are the limitations of such a module?
  - Main limitation is the lack of an arithmetic unit for advanced protocol fields. Another limitation is the requirement of hardware modules for PHY interfaces. This can be a problem for programmers with no experience with HDLs.

To review the performance of the PPU compared to the current solutions, table 1.1 is expanded with the a column for the PPU.

	Software	Accelerator	CAM	PPU
Hardware costs	++	-	- -	-
Development costs	-	+	+	+
Performance	- -	++	++	+
Flexibility	+	-	+	++
Time to market	+	-	+	+
Ease of integration	+	-	-	+/-

Table 6.2: Protocol processing solutions

The PPU fits nicely between the current solutions as was intended. The main disadvantage compared to software is that extra hardware is required. However the hardware cost is low and development costs are lower because it is easy to program the device. The performance will likely never reach the level of an accelerator, but it can come close to it. The biggest advantage is it flexibility. The hardware is able to deal with a large variety of protocols and data sizes. This allows for fast development reducing the time-to-market. The ease of integration is something that is subject to opinion. One has to select or develop the required PHY modules and then load the correct software. Even in the current state where there is no supporting software available, this process is not difficult and easy to do.

6.1 Future work

In this thesis work not everything that is possible with the PPU could be done, due to lack of time. This section is devoted to all things which are not finished or started, but could improve the PPU.

The compiler is one of the most important things that has to be completed. With a compiler, designing new protocols for the PPU will become a lot easier and faster. This is a key component for the PPU and it is vital to have if the PPU will be launched to the market.

Beside the compiler the main focus of improvement is with the hardware design. One of the first additions that could be made to the PPU and be of use are interrupts. The APB which is connected to the hardware has interrupt support, integrating this would take some time, but should not be a very big problem.

Another improvement is the addition of multiple comparators per PPU core. In the current design only one comparator is used for parsing incoming data. When branches with multiple entries are encountered the PPU has to parse all of them sequentially. With multiple comparators, multiple entries (or at least some of them) can be parsed in parallel. This does require an adjustment of the memory interface and the PPU core input, but the hardware performance could greatly benefit from this improvement.

The PPU could also be augmented with an accelerator. However a fixed function accelerator will not be useful in the flexible design of the PPU. Therefor improving the PPU with a Molen [17] accelerator would be much smarter. In the

current design, the PPU is unable to cope with an arithmetic operation if it is required by the protocol (for example checksum or offset calculations). Because each protocol requires other types of calculations it is not efficient to implement them all in the PPU core. The Molen architecture allows the PPU to have a very wide range of accelerator support, with only a few added instructions. Also because a protocol implementation is pretty much fixed during runtime, the Molen reconfigurable core only needs to be programmed once. With this added accelerator the performance gain of the PPU can be increased significantly.

With reconfigurable logic implemented with the PPU not only the back-end (the Molen concept), but also the front-end can be improved. By adding a reconfigurable logic section with the IO modules, support for various protocols can be implemented during programming. With this method a system equipped with a PPU can be adjusted to almost any protocol processing module. Another advantage is that dynamic pin assignment becomes trivial. With the reconfigurable logic blocks at the front- and back-end the PPU design would become like fig 6.2.

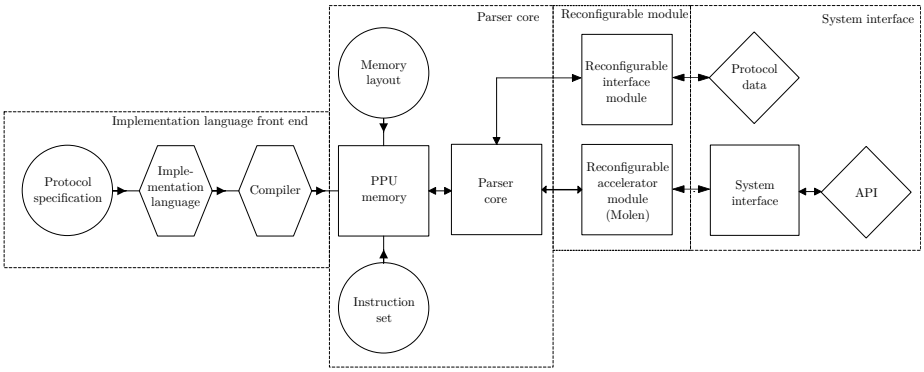


Figure 6.2: PPU overview with reconfigurable logic

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2 edition, 2002.

[2] ARM. Amba specification. Technical report, ARM, 1999.

[3] G. Berry. From regular expressions to deterministic automata. *Elsevier*, 48:117126, January 1986.

[4] R.H.J. Bloks. A grammar based approach towards the automatic implementation of data communication protocols in hardware. Technical report, TUe, 1993.

[5] David Chisnall. Understanding arm architectures, 2010. [Online; accessed 12-January-2015].

[6] Noam Chomsky. On certain formal properties of grammars. *Information and control*, (2):137–167, 1959.

[7] Tony Mason Doug Brown, John Levine. *lex & yacc, 2nd Edition*. O'Reilly Media, 2 edition, 1992.

[8] Annie P. Foong and Thomas R. Huff. Tcp performance re-visited. March 2003.

[9] Internet Engineering Task Force. Augmented bnf for syntax specifications: Abnf, 2008. [Online; accessed 6-January-2015].

[10] Aeroflex Gaisler. Leon3 processor. [Online; accessed 14-January-2015].

[11] Sanjeev Kumar. Ping attack, how bad is it? *Elsevier*, 2005.

[12] Nick McKeown. Packet switch architectures, 2006. [Online; accessed 13-January-2015].

[13] Microchip. Section 16. output compare, 2008. [Online; accessed 10-March-2015].

[14] P. Naur. Revised report on the algorithmic language algol 60. Technical report, May 1960.

[15] NXP. I2c-bus specification and user manual, 2014. [Online; accessed 10-March-2015].

[16] PassMark. Cpu benchmarks, 2015. [Online; accessed 10-March-2015].

[17] G. Gaydadjiev K. Bertels G. Kuzmanov E. Moscu Panainte S. Vassiliadis, S. Wong. The molen polymorphic processor. Technical report, Delft University of Technology, 2005.

[18] A.S. Tanenbaum. *Computer Networks, 4th Edition*. Prentice Hall, 4 edition, 2007.

[19] Guido Wachsmuth. Compiler construction, 2014.

[20] Wikipedia. Bit banging — wikipedia, the free encyclopedia, 2014. [Online; accessed 9-March-2015].

[21] Wikipedia. Content-addressable memory — wikipedia, the free encyclopedia, 2014. [Online; accessed 5-January-2015].

[22] Wikipedia. Decision problem — wikipedia, the free encyclopedia, 2014. [Online; accessed 3-February-2015].

[23] Wikipedia. Pulse width modulation — wikipedia, the free encyclopedia, 2014. [Online; accessed 10-March-2015].

[24] Wikipedia. Serial peripheral interface bus — wikipedia, the free encyclopedia, 2014. [Online; accessed 10-March-2015].

Chapter 7

Appendix

7.1 Instruction-set

10xx yyyz	Compare	32 bit compare value
11xx yyyz	Compare from register	16 bit register address
01xx xyyy	Multiple compares	-
0010 0xxx	Transmit	32 bit transmit value
0011 0xxx	Transmit from register	16 bit register address
0000 0000	Nop	-
0000 0001	Immediate jump	16 bit memory address
0000 0010	Delayed jump	16 bit memory address
0000 0011	Execute jump	-
0000 0100	Store in register	16 bit memory address
0000 0101	Store in memory	-
0000 0110	Repeat from register	16 bit register address
0000 0111	Repeat	16 bit repeat value
0000 1000	Repeat stop	-

Table 7.1: Instruction set