Proknet©: Not your average IP over ATM segmenter

Submitted in partial fulfillment of the requirements for 97.584:

VLSI Design

Professor Ralph Mason

Fall Term 2000

Wael Hermas†
Rami Abielmona†
Mohamed Abou-Gabal†

†(whermas,rabiel,mabou)@doe.carleton.ca

December 10, 2000

Contents

List of Figures

List of Tables

Part I

Preface

0.1 Abstract

This project presents Proknet©. It is a play on the following words: network and processor. It represents a module that is capable of receiving incoming variable-sized IP packets, and outputting fixed-sized ATM cells. The applications of this module are numerous, especially in the communications field, where this type of circuit is needed in order to implement the ATM forum specifications. The following report is presented in a book fashion, in order to clearly dissect all of the tasks that were delegated and accomplished by the designers. To be specific, any TLL- or SAR-related work is accredited to Wael Hermas, any trailer- or CRC- related work is accredited to Mohamed Abou-Gabal, and any micro-processor/memory- or ALU-related work is accredited to Rami Abielmona. Appendices are included in this report presenting the different modules and their corresponding source and output files (Verilog code, functional/timing simulation diagrams, and synthesis outputs). Finally, attached to this report is a CD (also see Appendix I), which contains all of module related work *and* a PowerPoint 2000 (Microsoft©) lasting approximately 5 minutes. This presentation is an overview of the overall system architecture and operation. Take a few minutes and view the presentation (as a slide show) in order to get familiar with the many topics that are discussed in this report.

0.2 List of Acronyms

AAL	ATM Adaptation Layer
ACC	Accumulator
ACU	Address Control Unit
ALU	Arithmetic/Logic Unit
AR	Address Register
AR	Flip-flop Asynchronous Reset
ARSE	Flip-flop Asynchronous Reset and Synchronous Enable
AS	Flip-flop Asynchronous Set
ASM	Algorithmic State Machine
ATM	Asynchronous Transfer Mode
CISC	Complex Instruction Set Computer
CPCS	Common Part Convergence Sublayer
CPE	Customer Premise Equipment
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CS	Chip Select
DPRAM	Dual-Port RAM
EOP	End of Packet
FAS	Full Adder-Subtractor
FF	Flip-flop
FIFO	First-in First-out
FSM	Finite State Machine
ICU	Instruction Control Unit
IP	Internet Protocol
IR	Instruction Register
MCU	Main Control Unit
OSI	Open System Interconnection
PC	Program Counter
PDU	Protocol Data Unit
PR	Pad Register
RAM	Random Access Memory
RAR	Read Address Register
ROM	Read-Only Memory
RISC	Reduced Instruction Set Computer
SAP	Service Access Point
SAR	Segmentation and Reassembly
SP	Stack Pointer
SPR	Special Purpose Register
SSCS	Service Specific Convergence Sublayer
TPRAM	Ternary-Port RAM
TLL	Total Length Logic
TLR	Total Length Register
VLIW	Very Long Instruction Word
WAR	Write Address Register

Part II

Introductions

Chapter 1

IP over ATM

The Internet has become one of the most diverse, information-filled library this century has ever seen, since the library of the U.S Congress. Its contents range from a variety of different topics, some of which include education, information, leisure and entertainment, as well as research and development. As the Internet becomes more reliable even on a daily basis, we are evolving to be more dependent on its various tools and services. Our dependency hence increases our awareness of the Internet, which in its turn leads to customers demanding more diverse and tailored services that they can map to their everyday lives. This demand drives he emergence of new technologies as well as the refinement of old ones in ways never thought possible before. Two of the leading technologies that have surfaced are the **Internet Protocol (IP)**, and **Asynchronous Transfer mode (ATM)**. There are many differences between the two communication protocols, but due to the scope of this document, only a few will be highlighted for clarity. The IP communication protocol utilizes connectionless-based datagrams that are routed using hop-by-hop routing algorithms to transfer information, while ATM utilizes connection-based cells that are guided using a pre-planned path between nodes. The main difference between the two is the processing power required by both as well as the reliability offered by both protocols. As shown in figure 1.1, the total length of the packet is variable ranging from 20 bytes to as big as 65,536 bytes. The datagram approach is best suited for this type of routing methodology because each packet is treated independently, with no reference to packets that are gone before it. Different packets may thus end up selecting different routes depending on the network's current status, and thus may end up arriving at the destination in different orders. Neither error control nor any reliability mechanisms are exercised in this protocol, which allows for the transmission of such variable sized packets, yet routing of every single packet is required which consumes greater processing power. Now referring to figure 1.2, the total length of an ATM cell is fixed at 53 bytes in total. The cell approach is alternatively best suited for this type of routing methodology because each packet is guided via a pre-planned route, where each cell traverses the dedicated route for the entire length of data transmission. With this scenario, error control and reliability mechanisms are provided on a connection basis rather than a cell basis, which reduces the processing power required by each node.

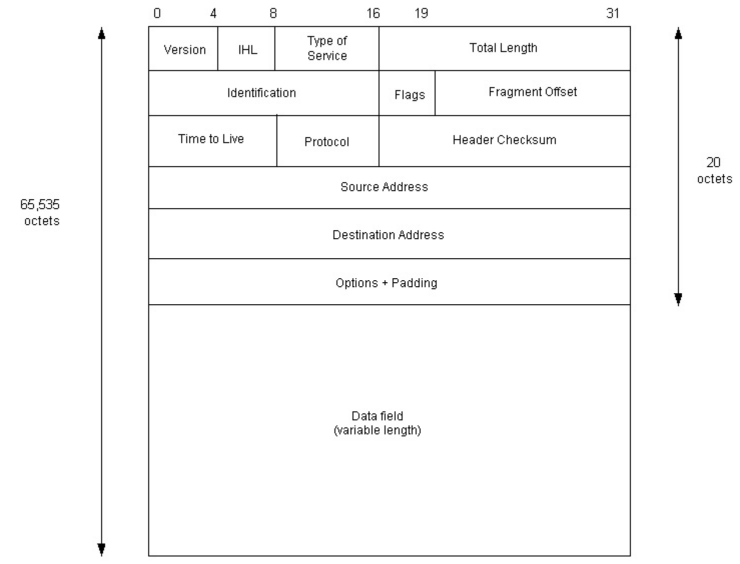


Figure 1.1: IP packet

Currently though, as a result of different amalgamated services, more and more packet-switching networks have been making use of the more reliable cell-switching networks. The benefit from both technologies hence improved the different service provider's delivered products. This allowed networks around the world to present a service that is as reliable and dedicated as ATM, while utilizing IP's greater transmission capability and reduced overhead in terms of route setups. Out of this concept transpired the migration of networks to offer **IP over ATM**. As described above, this protocol uses both technologies to deliver data, voice and video from one **Customer Premise Equipment (CPE)** to another **CPE** and ultimately to the user that desires this service. In order to be able to transmit variable-sized IP packets over a fixed-cell ATM network, we have to be able to segment the IP packets into 48-byte cells at the transmitter, and be also able to reassemble these same cells into the original packet at the receiver. Thus, this is going to be the focus of this project and the discussion of **Segmentation and Reassembly (SAR)**.

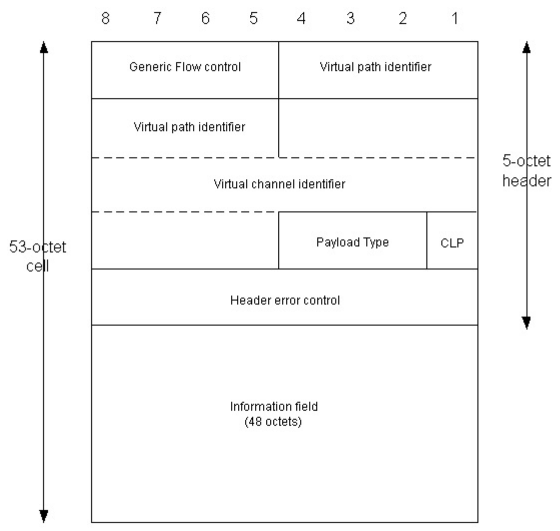


Figure 1.2: ATM cell

1.1 Segmentation Theory

Figure 1.3 refers to the **Open System Interconnection (OSI)** model developed by ISO as a model for computer communications architecture. Within the ATM reference model, the **ATM Adaptation Layer (AAL)** has the ability to provide a transitional layer between the ATM layer and the higher protocols. This allows for an enhancement of services provided by the ATM layer to all the higher level layers, which in our case is IP. The functions performed by the AAL depend on the higher layer requirements.

1.2 AAL Protocols

There are different types of protocols that AAL supports in order to meet the variety of needs of the higher layers, which are all defined under four classes of service that cover a broad range of requirements. The classification is based on whether the application requires timing between the source and

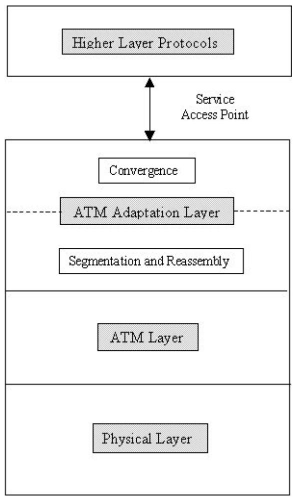


Figure 1.3: ATM protocol reference model

destination, whether it requires a constant bit rate for traffic, and finally whether it is a connection-oriented or connection-less transfer. Based on this criterion for classes of service, ITU-T defined four different AAL protocols that classify the operation the AAL based on each class of service [Hal92]. As mentioned in figure 1.3, the AAL is made up of two logical sub layers: the **Convergence Sub layer (CS)** and the **Segmentation And Reassembly sub layer**. The convergence sub layer is needed to provide functions for specific applications that are using the AAL. Higher protocols attach to the AAL by using a **Service Access Point (SAP)**, which is the address of the application. This makes the CS a service dependent layer.

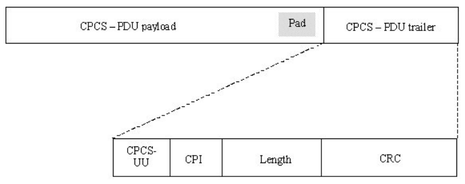
The SAR sub layer is then required to accept information from the CS while molding them into cells that are acceptable for transmission by the ATM layer, as well as doing the opposite upon reception of cells from the ATM layer at the other end. As mentioned above, the ATM layer can only accept a 48-byte payload. Thus the SAR has to group any SAR headers and/or trailers as well as the CS payload into 48-byte cells ready for processing by the ATM layer.

1.3 ATM Adaptation Layer 5

Thus, for our IP application, the characteristics are that timing is not required between the source and the destination, the bit rate is variable, and it is a connection-less transfer. This classifies IP as a class D application, which entails the utilization of the type 5 AAL protocol, which is a newly defined protocol. Like all other AALs, the higher layer protocols send a block of data to the CS, which encapsulates them into **Protocol Data Units (PDU)** [Tan96]. Actually, the CS is referred to as the **Common Part Convergence Sub layer (CPCS)**, making this layer in the CS the one that performs common functions for the higher protocols, while the **Service Specific Convergence Sub layer (SSCS)** performs the service specific operations depending on which AAL is being utilized at the time by the higher layer protocols based on their application.

1.3.1 The CPCS within AAL5

The function of the CPCS is very limited in for layer 5, in contrast with layer 3/4 which has a more involved role in the AAL. CPCS takes on most of the functions as the one used in layer 3/4, but with the difference of exclusively providing error protection without the involvement of the SAR, as well as not giving a buffer allocation size indication to the receiving peer entity. Mainly, the CPCS is responsible for the following functions as show in figure 1.4 [Pry95]:



Legend

CPCS – PDU payload	User Data	(0…65535 bytes)
Pad	Padding	(0…47 bytes)
CPCS – UU	User-User information	(1 byte)
CPI	Common Part Identifier	(1 byte)
Length	Length of the payload field	(2 bytes)
CRC-32	Cydic Redundancy Check	(4 bytes)

Figure 1.4: AAL type 5 CPCS PDU

- Padding (Pad) field This field allows the CPCS PDU total length to be a multiple of 48, aligning it with the cell-size of an ATM cell. Its size varies from 0 to 47 bytes.

- User to User field (UU) This field is used transparently between the AAL5 CPCS users, containing 1 byte of information

- Common Part Identifier (CPI) This field contains zeros at the moment, indicating that there are user data in the payload field.

- Length field This field indicates the length of the CPCS PDU payload stored in the CPCS PDU. This is required in order to determine the size of the pad field. The size length field varies from 0 to 2 bytes, which represents the size of the payload. As well, of note here is that the total length of an IP packet also varies from 16 bytes (header) to 65535 bytes.

- CRC-32 field This field contains the CRC-32 calculation, which is done over the entire CPCS PDU, including the payload, the pad, and the CPCS-PDU trailer in entire contents.

The polynomial used to generate CRC-32 is the following:

$$G(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1 \tag{1.1}$$

Chapter 2

System Architecture

The main goal of this project is to provide an IP over ATM segmentation entity, that will accept a variable-sized IP packet, and transform it into fixed-sized ATM cells. To perform this goal, it was necessary to select a base architecture, that will aid us in the design and implementation phases. This task was not taken lightly as it will influence many future design aspects and decisions. The system architecture has to present to the designer the main components, their interconnections and their operations. That said, the system architecture is not going to present the internal implementation details of any component, leaving it up to the imagination and innovation of the designer to design a functionally correct system. For this project, the system architecture went through quite a few design cycles before its eventual hardened declaration. The fact that the architecture is presented first in this document is due its vitality for a smooth module design transition. The drawbacks of "hard-coding" the circuit to match the underlying architecture are numerous, and expensive in terms of design cycles. Other pitfalls of a bad system architecture are work-arounds implemented in order to do exactly that: work around a system bug. On the other hand, a well thought-out architecture will provide a baseline for all designers to review and synchronize to.

The system architecture started out with a basic design involving all the functions needed for a SAR entity (segmentation only in this case). Please refer to figure 2.1 for the first design done. This design is very preliminary and did not include many of the details of the correct system operation. It does present, however, the goals of the project, the inputs and outputs of the system, and the high-level data flow. On the other hand, the design is missing quite a few important aspects, such as appending a trailer after each

IP packet. The design does not also take into effect which module should perform what operation. For that matter, the design does not even break down the system into different modules. As a first design, it is quite concise and presents the idea in an orderly matter.

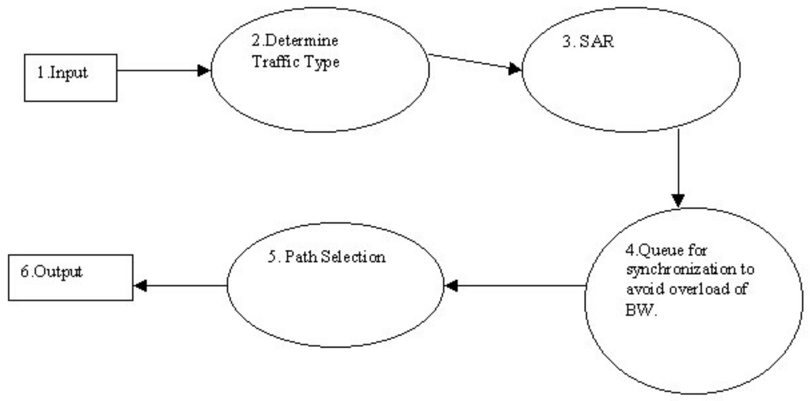


Figure 2.1: Preliminary system architecture

The next design is a CPU-intensive design, as seen in figure 2.2. This architecture plays very nicely into the re-programmability aspect of hardware design. The CPU sequentially performs its functions. The throughput of the design is considered to be low due to the intense CPU-memory interaction in sending completed packets to the SAR module. What this design presents is a breakdown of the system into components. The architecture also clearly shows the datapath as well as the system inputs and outputs. The architecture also defines some previously mysterious interactions between different logical components, as well as the inputs and outputs of each of the modules. As time went on, the design was modified in order to attempt to speed up the processing throughput.

An intermediate design that was briefly considered involved each of the modules shown in figure 2.2 interacting with the memory unit. This design allowed for multiple packets to be processed simultaneously, but required the use of a large multiport RAM and thus was instantly scrapped.

The final system architecture involved a pipelined design from start to end. There are two actual pipelines, an input and an output pipeline. Please refer to figure 2.3 for a clearer picture of the following discussion. The input pipeline contains the *TLL* module, the *CRC* module and the *trailer* module, while the output pipeline contains the *SAR* module. Sandwiched in

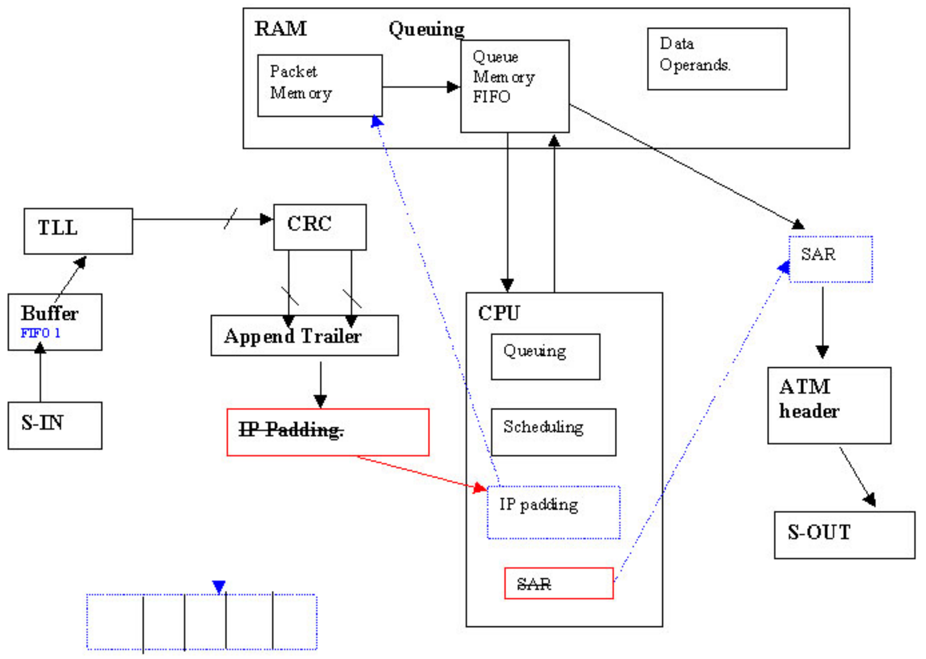


Figure 2.2: Intermediate system architecture

between both pipelines are the CPU and the memory modules, which link the output of the input pipeline to the input of the output pipeline. The pipelined design allows us to handle multiple packets simultaneously without the need for much external RAM. Each module is able to operate according to specification on the IP packet, as frames are transferred, byte-by-byte, through the pipeline (1 byte/1 clock cycle). Other advantages to this design is that each module views the whole packet one byte at a time, and hence packet processing time is minimized. The whole architecture seems to be operating in a parallel fashion, while latency and throughput are minimized and maximized, respectively. One final issue is that the CPU is left to handle non-data-intensive tasks, and thus memory interaction is also reduced.

The reasons for adding the **total length logic (TLL)** module on the pipeline are to find the total length of the IP packet, and setting the result into the **total length (TL)** register. The TLL also calculates the number of bytes (assuming they are zeros) needed to be used in padding. The latter is done by setting the **padding register (PR)**. The pipeline processes are

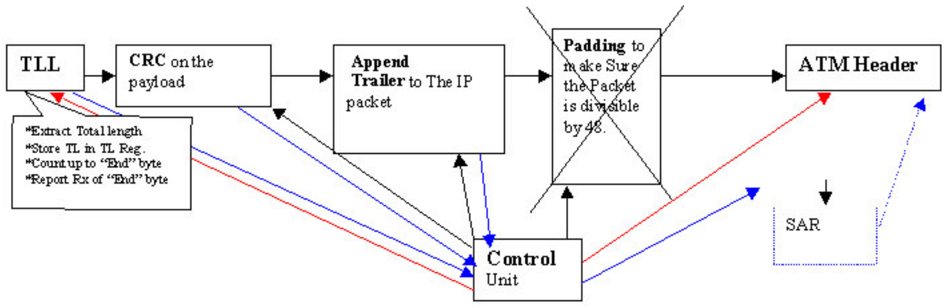


Figure 2.3: Pre-final system architecture

data intensive that do not present any delays. The modules on the pipeline are the ones which require control signals (from a control unit).

What follows are a few of the design reviews that were performed to the final system architecture, and are included here as is, in order to give a *reality* aspect to the project:

October 9th - Design Re-engineer(DR): The padding algorithm/module should be moved to a different position on the pipeline. The "Padding" should be implemented right after "TLL" and before "CRC". "Padding" was moved around like this because we did not take into consideration of why "padding" was placed right after "Append T Trailer". The only reason why is that we just switched "queuing" and "padding" from the CPU module. The other reason of why it makes sense to put it after TLL is because the way CRC is calculated. CRC needs to be done on all the bits in the frame including trailer and CRC (XOR + shift). Thus it is easier to pad after TLL because TLL has already supplied us w/ all the relevant info in order for padding to occur. Solution: The solution proposed was to move the "padding" module from where it was before to a position after "TLL" and before "CRC.

October 9th - DR: We have just realized that a bottleneck has occurred in our pipeline design. The bottleneck came up when we realized that what happens when you actually begin to pad. The "padding" module will begin to pad for the current packet, while it receives the byte of the next IP packet. This is not good because we have "racing" conditions creeping up in the design. The "padding" module will have to identify when the "END" byte was received before it can start the pad process. While this is happening, the Trailer module is already passing the next byte of the IP packet into the "padding" module. Extra logic would have to implemented like a mux or delay to allow the padding to do its job sequentially. Solution: The solution that was agreed upon is to move the "pad" module back into the CPU in order to relieve that delay or bottleneck introduced. This is because padding in the CPU is allowed in order to relieve the stress on the bottleneck and because CRC can be hard-coded for 1-byte of "zeros" in the padded area.

On October 21. We decided to insert Trailer Module before CRC because we have to perform CRC on the Trailer according to the spec and on the receiver end CRC will be performed on Trailer and The Message only. We assume that the receiver end will only do CRC on intelligent enough to recognize that where message starts and ends.

The final pipelined design is shown in figure 2.4. This design is discussed in more detail

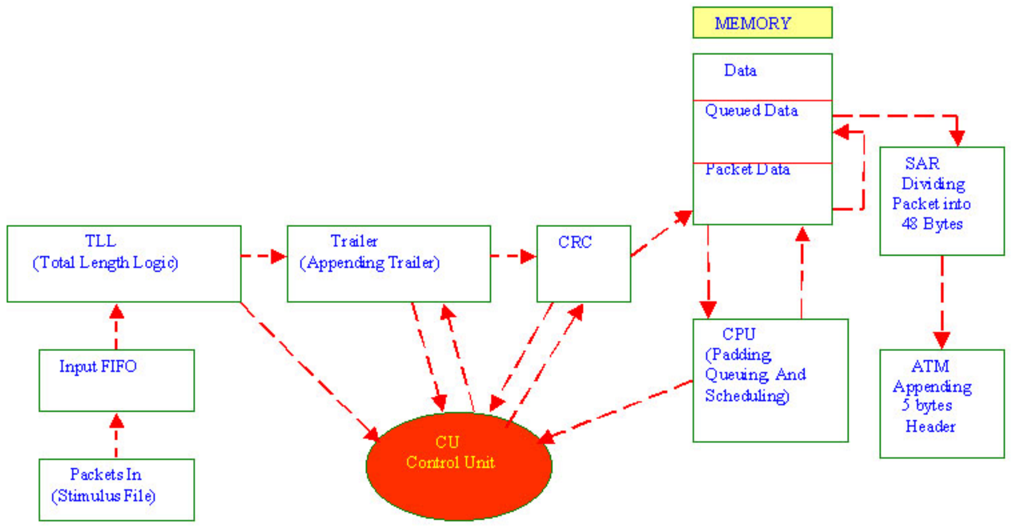

MEMORY
Data
Queued Data
Packet Data
SAR Dividing Packet into 48 Bytes
TLL (Total Length Logic)
Trailer (Appending Trailer)
CRC
Input FIFO
Packets In (Stimulus File)
CU Control Unit
CPU (Padding Queuing And Scheduling)
ATM Appending 5 bytes Header

Figure 2.4: Final system architecture

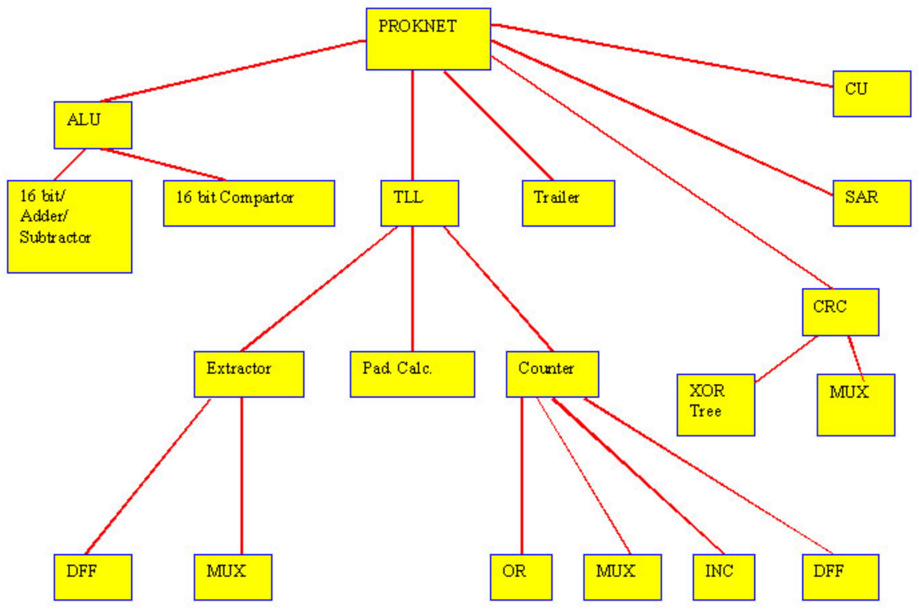


Figure 2.5: System breakdown

Part III

Module design

Chapter 3

TLL module design

3.1 Top Level

3.1.1 Theory

In order to be able to design a fully functional SAR according to our specifications, we will require the retrieval of certain information form the IP packet itself. This information is necessary in order for the SAR to proactively monitor the stream of packets entering its module, as well as perform preliminary setups for the downstream modules, such as padding and memory storage. As in figure 3.1, the most important information contained in the IP packet is the **Total Length (TL)** field, which is stored in the 3^{rd} and 4^{th} byte of the header, making it a 16 bit field, and hence setting the total length of the packet to be between 20 bytes (minimum length of the header) and 65,535 bytes (maximum allowable transferable length).

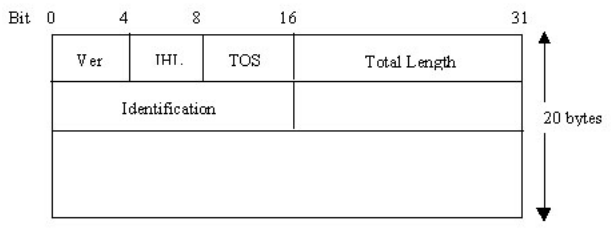


Figure 3.1: IP packet header

By foreshadowing the TL, we are able to determine several other necessities that are to be used by the pipeline at its various stages. Thus the need arises for a method that allows us to read the TL field for the incoming packet, storing it for further use as the packet transverses the pipeline, as well as determining the necessities early in the pipeline in order to ensure a heuristic transfer of the packet from module to module. Some of these necessities are calculating the number of padding bits required by the packet as well as the storage of that value in a manner that is easily accessible to the control unit and hence the modules along the pipeline The **Total Length Logic (TLL)** is the name given to the module that is to fulfill the latter requirements.

3.1.2 Design History

The original design of the TLL allowed for only the ability to read the TL of the IP packet and store that value in an entity labeled **Total Length Register (TLR)**. The TLR is to hold its value until the extractor reads the next packet's TL. Hence, due to this idea was born the *extractor logic* module within the TLL, whose sole purpose is to extract the TL of the IP packet from the 3^{rd} and 4^{th} byte of the header, and store that value in the TLR. The TLR is to be 16 bits long in order to hold the value of the TL, which is also 16 bits long. The extractor, along with the rest of the pipeline, is to operate assuming the follow conditions:

1. Externally fed clock oscillating at 25 MHz. Thus our processing power allows to output data at a rate of 200 Mbps. This is obtained by the following: 8 bits/cycle (width of bus) * 25 MHz (cycles/sec) = 200M bps

2. Our pipeline is to be fed one byte at a time clocking at a speed of 1/25 MHz = 40 ns, which is the period of one clock cycle.

Thus the extractor is designed to extract the 3^{rd} byte out of the IP header and store it in the LSByte of the TLR, followed by extracting the 4^{th} byte and storing it in the MSByte of the TLR. This is illustrated in figure 3.2.

Based on the above diagram, an algorithm was designed to implement the extractor and the TLR at the same time. Figure 3.3 outlines the algorithm.

We based the extraction mechanism and its transfer into TLR on two signals that indicate the processing of the third byte and the fourth byte. The availability of the third byte signals the data to be loaded into the least significant byte of TLR, while the availability of the fourth byte signals the

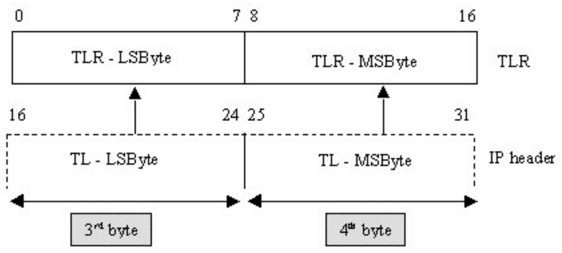


Figure 3.2: TLR storage

data to be loaded into the most significant byte of TLR. We thus obtained the following circuit, shown in figure 3.4 based on the defined algorithm. The algorithm will be discussed in detail in the functional specification section.

3.1.3 Functional and Hardware Specifications

The operational behavior of the TLL was defined to be a module that acts upon the total length field in the IP packet header. By clearly outlining the functions of the above modules, i.e. extractor, counter, and padder, we have reached the step where we would group all of them together under one module, and that is the TLL. Thus our definition of the TLL is nothing more than an instantiation of all three modules in order to operate correctly and output the relevant signals that re required by Proknet. Thus we will define the I/O signals of the TLL:

- Input: *clk, resetb, star_counting, pipeline_data_in*
- Output: *s_PR, s_Prout, s_TLR, pipeline_data_out, c_DoneC*

In short words as to not repeat the description of the above modules, the TLL receives packet an input buffer, which it continuously streams to the output module, in this case being the trailer. Along the passage of the bytes through the TLL, the three modules begin to operate according to their definition. The counter begins counting and signals the passage of several key bytes. The extractor is awaiting these signals in order to extract the TL from the IP header. Upon storing the TL, the padder begins its operations in order to calculate the pad bits. All these processes occur sequentially and in parallel of each other. The extractor module for example awaits the *s_Done3* signals from the counter in order to kick-start its task. Once it has begun, then it is not listening to the counter module anymore, up until it the arrival of a new packet to the TLL.

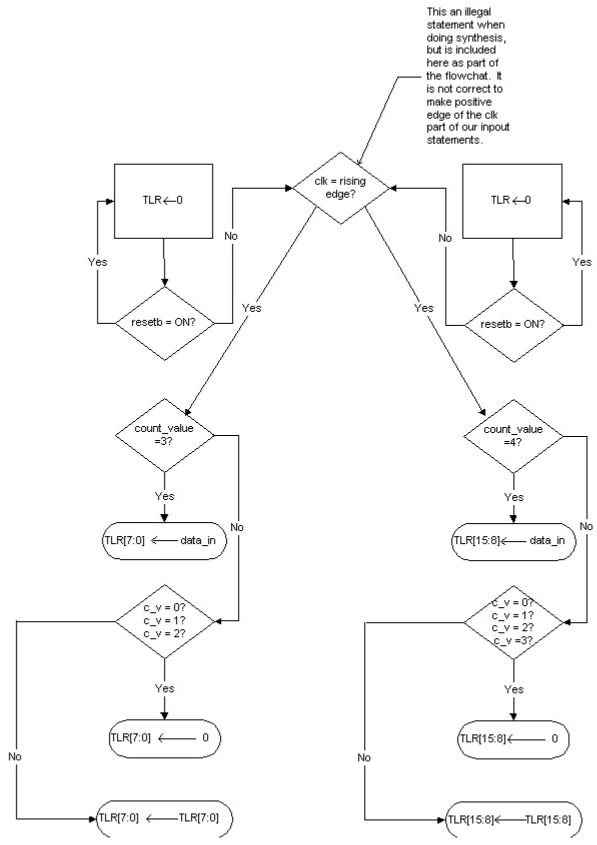
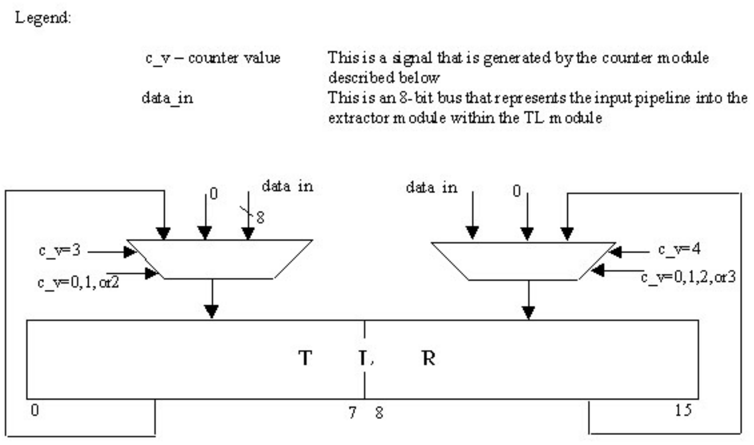

This an illegal statement when doing synthesis, but is included here as part of the flowchat. It is not correct to make positive edge of the clk part of our inpout statements.
clk = rising edge?
TLR ← 0
resetb = ON?
Yes
No
count_value =3?
count_value =4?
TLR[7:0] ← data_in
TLR[15:8] ← data_in
c_v = 0? c_v = 1? c_v = 2?
c_v = 0? c_v = 1? c_v = 2? c_v =3?
TLR[7:0] ← 0
TLR[15:8] ← 0
TLR[7:0] ← TLR[7:0]
TLR[15:8] ← TLR[15:8]

Figure 3.3: TLR/extractor algorithm

3.1.4 Functional Simulation

The TLL was defined in *TLL_top.v* and executed against the test bench labeled *TLL_top_tb.v.* the results of the functional simulation are outlined in Appendix A. Basically, the pipeline data streaming into the TLL contained only the TL, and in this case it was 15, i.e. the length of the IP packet was a fictional 15 bytes. As we can see, the PR and PRout registers both contain the number 25, which is the total number of padding bits required : # pad bits = 48 - (15 (TL of packet) + 8 (length of trailer)) = 25. as well, we can see that the *done* signal is asserted once the 15th byte is about to travel through the TLL, indicating that we have reached the end of our packet. Along the way to the end of the packet, the signals indicating the arrival of a certain packet were also asserted at their appropriate times during our cycle.


Legend:

c_v – counter value	This is a signal that is generated by the counter module described below
data_in	This is an 8-bit bus that represents the input pipeline into the extractor module within the TL module

Figure 3.4: TLR/extractor circuit

These are basically the functions that the TLL was defined to execute.

3.1.5 Synthesis

The synthesized TLL module is represented in Appendix A. We only show the top module synthesis, which contains COUNTER_TOP, TLL_EXTRACTOR, PADDER_TOP, ALU_TOP, and ACCUMULATOR. The accumulator was a simple module that was defined to serve the purpose of holding the result of an arithmetic operation in memory, for its use by other modules. The accumulator is defined in *accumulator.v*.

Thus, the rest of the modules are not expanded upon in this section as they will be discussed in their relevant sections.

3.1.6 Timing Simulation

Unfortunately, the gate level timing simulation for the TLL module did not execute as expected. As seen in the results, the number of padding bits did not correspond to the number obtained in the functional simulation. This is explained by the fact that upon the assertion of *s_LoadAB*, register A never gets loaded with the appropriate value, this is 15. on the next clock cycle, the result of the ALU addition in the ACC is loaded into register A. That addition is the value in 'A' (0) added with the value in 'B' (8), which get loaded accordingly. Thus, register A contains the number 8 as opposed to

23. It is easy to calculate how many padding bits are required for a packet of length 8 bytes, and that is 40 bytes, hence the contents of the PR and PRout registers. After several discussions, we have attributed this error to the fact that the mux responsible to correctly loading each register with its content was designed using behavior modeling, which might have caused some undesired results and configurations.

3.2 Counter

3.2.1 Theory

In order to implement the extractor/TLR, an up counter had to be designed for the sole purpose of tracking the number of bytes for a single packet that have transversed the TLL. As well, this is needed for the extractor/TLR module in order to signal the occurrence of processing the 3^{rd} and 4^{th} bytes in succession to allow the TLR/extractor to operate successfully. Thus, because of the latter mentioned reasons, a *counter logic* module is required within the TLL module. The counter design was originally thought out to be a just a simple register that is to be originally initialized with zeros and subsequently incremented on every successive clock cycle. This design method proved to be the wrong approach as it required the involvement of the ALU, which has a limited amount of clients that can access it at one time. This is a problem that we had to deal with frequently for several modules, and thus required multiple re-engineering efforts in order to satisfy our requirements. Thus, the most autonomous counter that we decided to implement is a 16-bit binary ripple up counter, which is easy and flexible to design as well as implement. Using **Algorithmic State Machine (ASM)**, a hardware design language that utilizes a concrete defined algorithmic methodology to define a design of any hardware circuit, the counter, along with its many control signals, was implemented The reader is welcomed to refer to [Lee00] for more details. The up counter will be discussed with more details in the following section.

3.2.2 Functional Specification

One of the most important and required modules that needed to be designed for this project is the counter module. As described above, the counter module has but two clearly defined purposes in the operation of the pipeline:

- When there are no packets or bytes of packets flowing through the TLL, the counter is to remain in the OFF state, while being preloaded

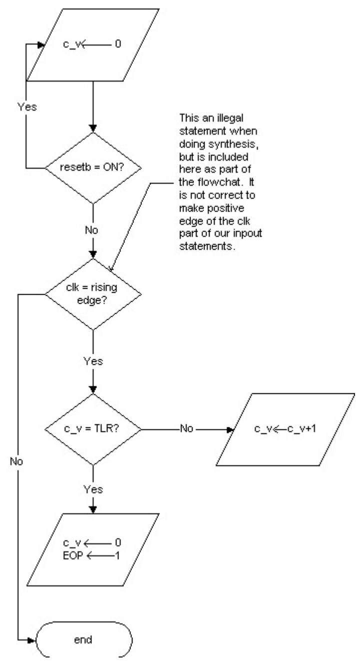

c_v ← 0
Yes
resetb = ON?
No
This an illegal statement when doing synthesis, but is included here as part of the flowchat. It is not correct to make positive edge of the clk part of our inpout statements.
clk = rising edge?
Yes
No
c_v = TLR?
No
c_v ← c_v+1
Yes
c_v ← 0
EOP ← 1
end

Figure 3.5: Initial counter algorithm

with "zeros" ready to begin counting at the moment a byte is presented to the TLL. One thing to mention here is that we have adapted a packet prediction mechanism to forecast to the counter the presence of a packet at the TLL input. This was necessary in order to kick start the counter to begin counting.

- When a packet is indeed at the input of the TLL, the counter is to begin counting, starting from zero and resetting when the TL value has been reached. The counter will have exposure to the TLR, loaded by the extractor module, and be able to use it as a comparator to determine when to stop counting and reset back to zero.
- As bytes traverse the TLL, the counter will generate several signals that are required by other modules. These signals indicate when the counter has reached a predetermined and specified value, i.e. the 5th byte, the 6th byte, etc. The following signals are generated by the counter module
  - ◇ *c_DoneC* : signal to indicate that the counter has done count-

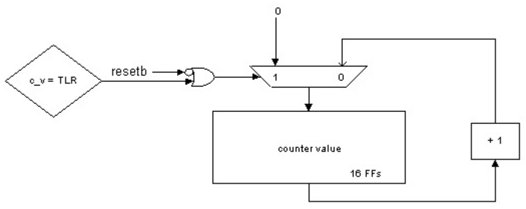


Figure 3.6: Initial counter implementation

ing; that is the counter has reached the TL value. Downstream modules refer to this as EOP.

- ◇ *s_Done3* : signal to indicate that the counter has reached the 3^{rd} byte.
- ◇ *s_Done4* : signal to indicate that the counter has reached the 4^{th} byte.
- ◇ *s_Done5* : signal to indicate that the counter has reached the 5^{th} byte.
- ◇ *s_Done6* : signal to indicate that the counter has reached the 6^{th} byte.
- ◇ *s_Done7* : signal to indicate that the counter has reached the 7^{th} byte.
- ◇ *s_Done8* : signal to indicate that the counter has reached the 8^{th} byte.

Besides these signals, the counter makes available the current counter value. This value is stored in the registers that make up the counter, and is broadcast to the rest of the modules if they need to use this value. The counter module is a simple module, and hence using ASM, we will examine its design. The ASM requires us to define pseudocode, generate an ASM chart, generate a *data path*, generate a detailed ASM chart, and generate a *control path*. Below is the design methodology of the up counter:

1. Pseudocode: S0. *s_counter_value* $\longleftarrow$ 0

   C0. If (*s_counter_value* = s_TLR), (goto s0) else (goto s1)

   S1. *s_counter_value*++, (goto c0)

2. ASM chart

3. Data Path

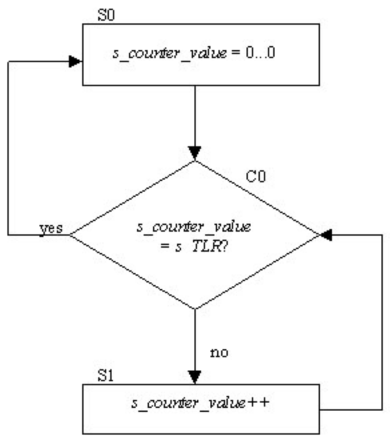


Figure 3.7: ASM chart

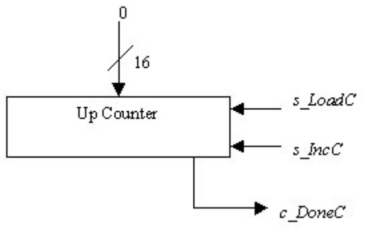


Figure 3.8: Data path

4. Detailed ASM chart
5. Control Path
6. Status Signals

The data path is responsible for the data part of the counter, i.e. incrementing the up counter by one at each rising edge of the clock. On the other hand, the control path is responsible for controlling the operation of the up counter, i.e. instructing the counter to preload the value zero into the up counter or instructing it to increment as long as it *c_DoneC* hasn't been asserted. *c_DoneC* and the rest of the status signals will be generated using a simple comparison method using primitive gates, described in the next section.

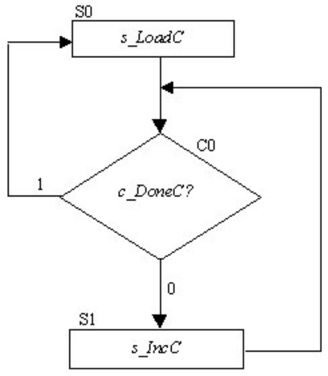


Figure 3.9: Detailed ASM chart

3.2.3 Hardware Specification

The counting sequence that is to be executed by the up counter will be of the following manner: 000, 001, 010, 011, 100, 101...but to be applicable over 16 bits. Examining the sequence with a closer view and with the help of truth tables and logic minimization, we were able to determine that the toggling capability of the T-flip-flops, which are nothing more that D flip-flops whose inputs are their inverted outputs, depending on the input makes them ideal for our up-counter. Combinational logic will have to be implemented in order to trigger the toggling of the flip-flops depending on what value needs to be changed at every clock cycle. For this reason, the flip-flops are to be made of the synchronous type, where the clock inputs of all the flip-flops are connected together and are triggered by the input pulses, making the flip-flops change states simultaneously. Figure 3.13 represents the interconnection of the flip-flops to implement the up counter.

In figure 3.13, the ”reset” and ”enable” signals to the flip-flops are left in their primitive state as they will be determined according to the ASM design. Both of these signals will be determined as we begin to clearly define our hardware circuitry for the counter module following the ASM methodology of above.

The counter module is subdivided into two main sub-modules: a data path module and a control path module. As mentioned earlier, the sole purpose of the data path module is the implementation of the up counter using the method of figure 3.13, as well as the generation of the various status signals. The up counter will utilize 16 FFs to count up to 65,535.

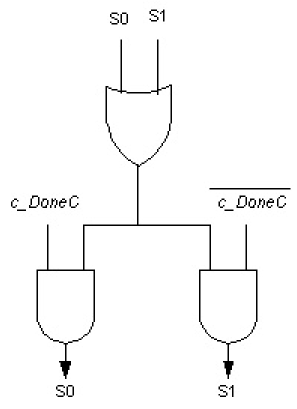


Figure 3.10: Control path

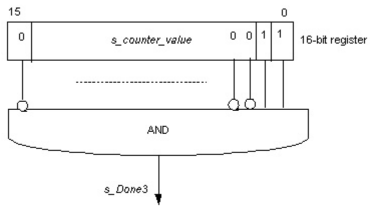


Figure 3.11: Status signal

Although we limited the maximum length of our packet to 100 bytes due to performance issues, we still implemented all 16 FFs. The status signals will be generated from within the XOR tree in the data path. *c_DoneC* will be a combinational output signal between *s_TLR* and *s_counter_value*, while the other status signals will each be a sequential output signal of the AND tree from within the XOR tree. Thus the inputs to the data path module are the global clock (*clk*), the global reset signal (*resetb*), *s_TLR*, *s_LoadC*, and *s_IncC* that are signals generated from the control path. The data path module also takes in a secondary input discussed earlier and is labeled as *start_counting*. This signal is used to forecast a packet that is about to transverse the TLL module. The *start_counting* signal also has for purpose to kick-start the up counter once every arrival of a new packet. Thus, it is connected with *c_DoneC* via a NOR gate to provide that functionality. *c_DoneC* is the only relevant signal because it is the one directly affecting the control path module. Thus, we have the following definition for the I/O signals of the data path module:

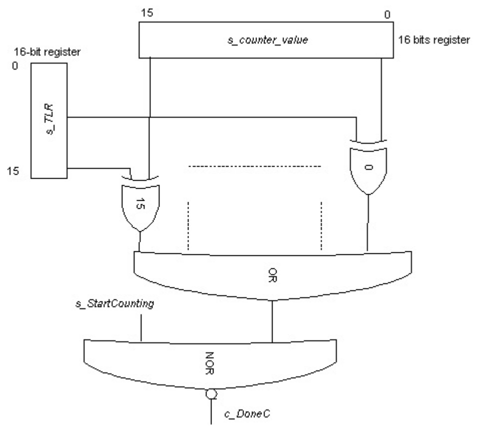


Figure 3.12: XOR tree

- Input: *clk, resetb, s_TLR, s_LoadC, s_IncC, start_counting*
- Output: *c_DoneC, s_Done3, s_Done4, s_Done5, s_Done6, s_Done7, s_Done8, s_counter_value.*

The control path module will control the operation of the up counter depending on the state that we are currently in. As per step 5 of the ASM method, the masked *c_DoneC* signal (containing *start_counting*) initiates the status change of the control path from loading with zeros to incrementing, which is going to initiate the up-counter's operation. Thus we can safely deduce from design (usage of "one-hot encoding" method) and from the above mentioned that whenever *s_IncC* is asserted, *s_LoadC* is de-asserted and vice-versa. Let us examine in more details those two signals as they prove to be the operating signals of the counter module. The control path module is further broken into two sub-modules as well: *STATE0_CL* module responsible for outputting *s_LoadC*, and *STATE1_CL* module responsible for outputting *s_IncC* . from a high-level perspective, we intended for the *s_LoadC* signal to preload zeros into the up counter while the *s_IncC* signal to increment the counter. We quickly realized that our choice of synchronous ripple counter using T flip-flops proved extremely valuable. The main characteristic of the counter's T FFs is that, provided all their control signals are present, i.e. Enable, clk, and reset, they will continue to operate at each clock cycle generating the correct counting sequence. Thus, in order to control their operation, we just have to make sure that the Enable signal is ON when there is a need to count, and OFF when there

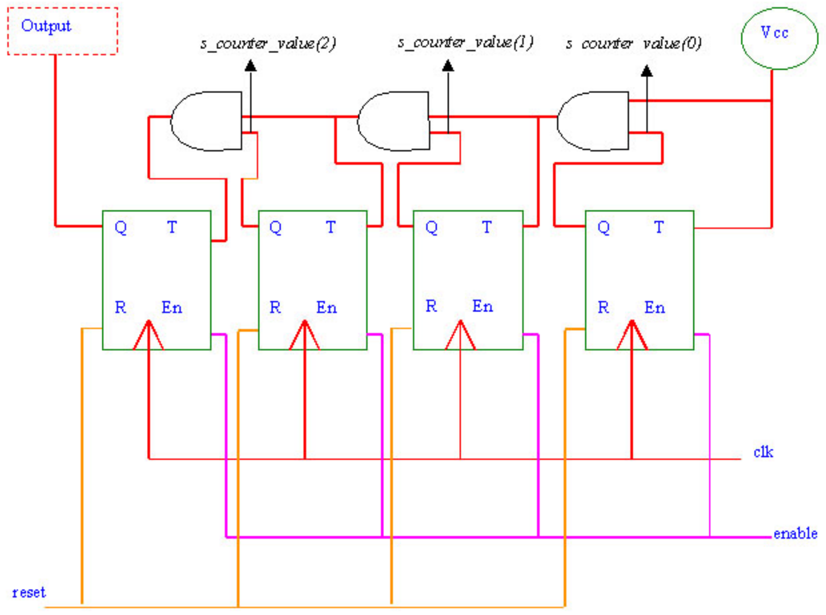


Figure 3.13: Ripple binary up counter

isn't. Hence, *s_IncC* as an output signal of STATE1_CL, and ultimately an output of the control path module, will be used as the Enable signal being inputted into the data path module. Similarly, in order to preload the counter with zeros, we need to preload or load all of the T FFs with zeros. This can be done by driving their reset signal whenever required. Thus, *s_LoadC* will be used as one reset signal that is to be combinationally incorporated with global reset signal in order to provide a reset circuitry for the FFs. Having defined the role of both output signals of the control path, a conflict aroused with regards to the operation of these signals. The "one-hot encoding" method allows us to only assert one signal at every clock cycle. As well, the T-FFs are require that when the Enable signal is asserted, the reset signal has to be de-asserted, whereas when the reset signal is asserted, it will take priority over any other signal. This will only cause a conflict when and only when the global reset signal is asserted, where both *s_IncC* and *s_LoadC* will be driven high. To combat this situation, we forced each signal to behave in an opposite manners by using an **Asynchronous Set (AS)** sequential logic circuitry for s_LoadC, while using an **Asynchronous Reset (AR)** sequential logic circuitry for *s_IncC*. The use of an AS flip-flop

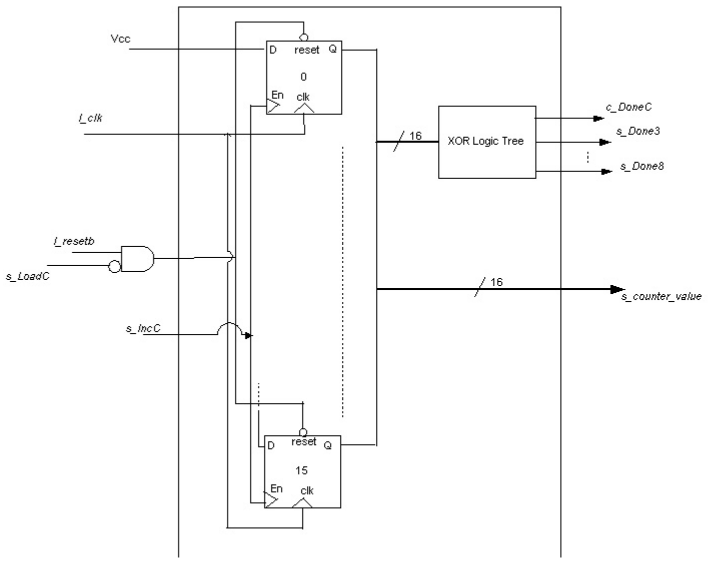


Figure 3.14: Data path of the counter module

for *s_LoadC* allows us to reset the T-FFs in the data path module every time the global reset line is asserted. This happens because the AS is being driven by the inverse of global reset, which we have chosen to be low. Thus, whenever the global reset is asserted, it will drive a logical high,'1', into the AS flip-flop, which will translate to a logical '1' as the reset line to the T-FFs, hence resetting all of them with a '0'. Thus, we can now define the inputs and outputs for the control path module ad being the following:

- Input: *clk, resetb, c_internal_reset (combinational circuitry for resetb and c_DoneC)*
- Output: *s_IncC, s_LoadC*

Figure 3.15 illustrates the control path module for the counter module. With both sub-modules defined for the counters, we can now examine the counter module's I/O signal port list as the following:

- Input: *clk, resetb, s_TLR, start_counting*
- Output: *c_DoneC, s_Done3, s_Done4, s_Done5, s_Done6, s_Done7, s_Done8, s_counter_value*

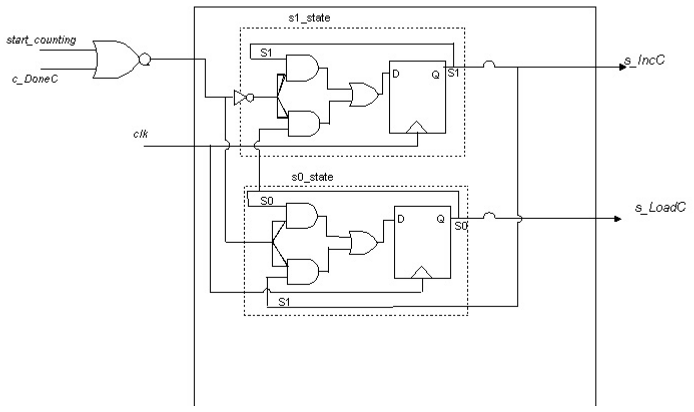


Figure 3.15: Control path of the counter module

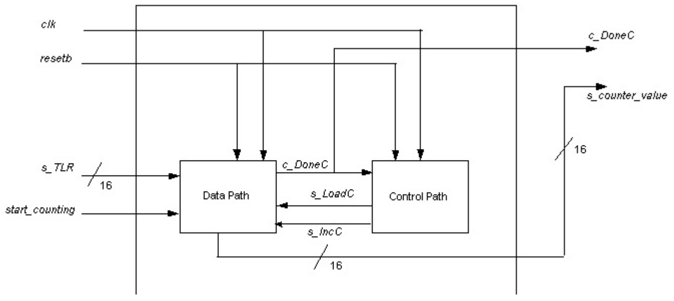


Figure 3.16: Counter module architecture

3.2.4 Functional Simulation

Functional simulation for the counter was executed using the top module test bench (*counter_top_tb.v*) definition along with the top counter module (*counter_top.v*). The functional simulation results are shown in the counter section of Appendix A. Functional simulation is needed to test the ability of the counter module to incrementally count beginning from zero and ending at a specified value. As well, we needed to test the ability of the module to produce all of the status signals required. The test bench targeted the behavior of the counter module as closely as possible to emulate its interaction with the rest of the modules. Thus, as can be seen from the simulation results, the test bench asserts the global rest line (active low) in the beginning of the simulation at 10 ns and de-asserts it at 30ns. This is required to load in the '0' values into the T-FFs. Next, the *start_counting* (referred to as *startC* in the counter module) is asserted at 60ns and de-asserted at

180ns. This has kick started the counter to begin the up-counting process. Of notice next is that TLR becomes valid after the 5^{th} byte transverses the TLL. This will represent the maximum value that the counter has to count to before asserting the *c_DoneC* signal. As we progress through the results with regards to time, we can see how the status signals are all being asserted as designed, i.e. *s_Done3* after the 3^{rd} byte, *s_Done4* after the 4^{th} byte, etc.. finally, after the 21^{st} byte has been received, the *c_DoneC* signal is asserted and the counter is reset back to '0' to start counting again. The results obtained from functional simulation proved to us that the counter behaved as it was designed to do, though one major flaw surfaced while closely examining the results. It was thought that upon the assertion the *startC* signal, the counter would immediately begin to count incrementally. Yet, this is not the case in our simulation results. The counter's actual behavior is attributed to the fact that writing the code in RTL to model its operation allowed us more flexibility and control over the behavior of the counter, but in the process created extra sequential logic that might have not existed if the code was written behaviorally. As noticed, the counter begins counting 2 full clock cycles after the assertion of *startC*. This is attributed to the fact that when *startC* enters the control path module, it correctly initiates *s_IncC* on the next clock cycle. *s_IncC* in its turn enables the T-FFs to begin incrementing on the next clock cycle as well, hence losing two clock cycles in the process. This was considered to be a flaw in the operation of the counter due to the result of the design.

3.2.5 Synthesis

Beginning with the top module representation, the I/Os identified above are the signals that constitute the COUNTER_TOP black box. Delving closer into the design of that black box, we discover that two components are instantiated: the CONTROL_PATH_CL and the DATA_PATH_CL along with their associated I/O signals. The data path component is further broken into two T_FF_REG_EIGHT parts, each representing an ARSE_REG_EIGHT component whose input is tied to the combination of its inverted output with the input to the T FF, and an XOR_TREE, representing the combinational logic needed to generate the various status signals. Yet, the ARSE_REG_EIGHT component is nothing but a definition of 8 D-FFs (ARSEs) that make up our up-counter along with their interconnections according to figure 3.13. Looking closer at the ARSE components, we notice that there are 8 D_FF_ARSE components connected to their respective input/output busses (*d[7:0]* - *q[7:0]* ; *d[15:8]* - *q[15:8]*). The XOR_TREE on the other hand has two components to it: the combinational logic for *c_DoneC* and the sequential logic for *s_Done3*, *s_Done4* and the rest of the status signals. This is the circuit that was designed in step 6 of the ASM. Comparable to the data path, the control path is also divided into two components, STATE0_CL and STATE1_CL. As illustrated in figure 3.15, STATE0_CL defines a D_FF_AS along with the combinational logic as its input, while STATE1_CL defines a D_FF_AR along with its combinational logic as its input. Thus, with the definition of both path modules, we can conclude that the counter module synthesized into the precise hardware circuit that we intended for it using RTL code generation.

3.2.6 Timing Simulation

After synthesizing our counter for the creation of the gate level modeling, we used the Cadence design tools to generate the net list that makes up the design of the complete counter. We then attempted a first trial simulation using the same test bench as in the functional simulation to generate our gate level timing simulation results. The results are outlined in Appendix A for the TLL. As can be seen, the gate level simulation results matched in an accurate manner the results obtained for functional simulation. The net list verilog code reflected the exact behavior of our counter, hence declaring success with regards to the design of this module.

3.3 Extractor

3.3.1 Functional Specification

As mentioned in the section covering the counter module, and as stated in the beginning of the TLL design history, the extractor has a function of extracting the total length value stored in the 3^{rd} and 4^{th} byte of the IP packet's header. It is then responsible to transfer this value into a special purpose register called TLR. The TLR contents will be made available for other modules to use depending on their requirement. The TLR is to hold its contents until the next packet is injected into the TLL and along the pipeline. We have slightly modified the operation of the extractor module after designing the counter module. It became obvious that the extractor as well as the TLR can be grouped into one entity instead of two entities as the original design slated them to be. The TLR will then act as an extractor based on the reception of certain control signals as and act as a temporary storage medium as well for the TL value based also on these same control signals.

3.3.2 Hardware Specification

The extractor module will only consist of the **Total Length Register**, which is to be constructed using 16 D-flip-flops to hold the TL value of the packet. The reasoning behind the number of FFs is the same as the one adapted for the counter's 16-bit FFs: Although we are limiting our IP packet's length to be 100 bytes, where only 7 bits are needed, we chose to implement the extractor to reflect as much as possible a real-life application for the product. Thus, we have redefined our design for the extractor/TLR than was previously discussed in the above section of the design history. Figure 3.4 outlines the implementation of the D flip-flops that constitute the extractor/TLR. As illustrated by figure 3.4, the D flip-flops utilize their ability to employ their enable signal to be able to latch and hold the value that is currently available at their input. Thus FF#0 through FF#8 will be latching the current data available at the bus upon the assertion of *s_Done3*, which is indeed their "enable" signal. This will represent the least significant byte of the TL, which will occupy bits 0 through 7 in the TLR. On the other hand, FF#8 through FF#15 will be latching the current data available on the pipeline bus upon the assertion of *s_Done4*, which is their "enable" signal. This will now represent the most significant byte of the TL, which will occupy bits 8 through 15 in the TLR. The type of flip-flops used for the realization of the TLR are **ARSE (Asynchronous Reset - Synchronous Enable)** D flip-flops, which allows for the *resetb* signal to take precedence over any other signal, therefore making it asynchronous, and for the *enable* signal to be detected on the next rising clock edge, making it a synchronous signal. After the TLR has latched the 4^{th} byte of the current packet, it will be able to hold this value until *s_Done3* and *s_Done4* get reasserted again, making the TLR available to the rest of the modules while Proknet is processing the packet. The only setback from this design is that the TLR never gets re-initialized to zero after the arrival of the first packet, which was point recognizable in the design, but most likely having no side effects.

3.3.3 Functional Simulation

The functional simulation that was run for this module delivered the desired results in terms of what we designed the module to execute and in terms of what the test bench tested and verified. The results of the functional simu-

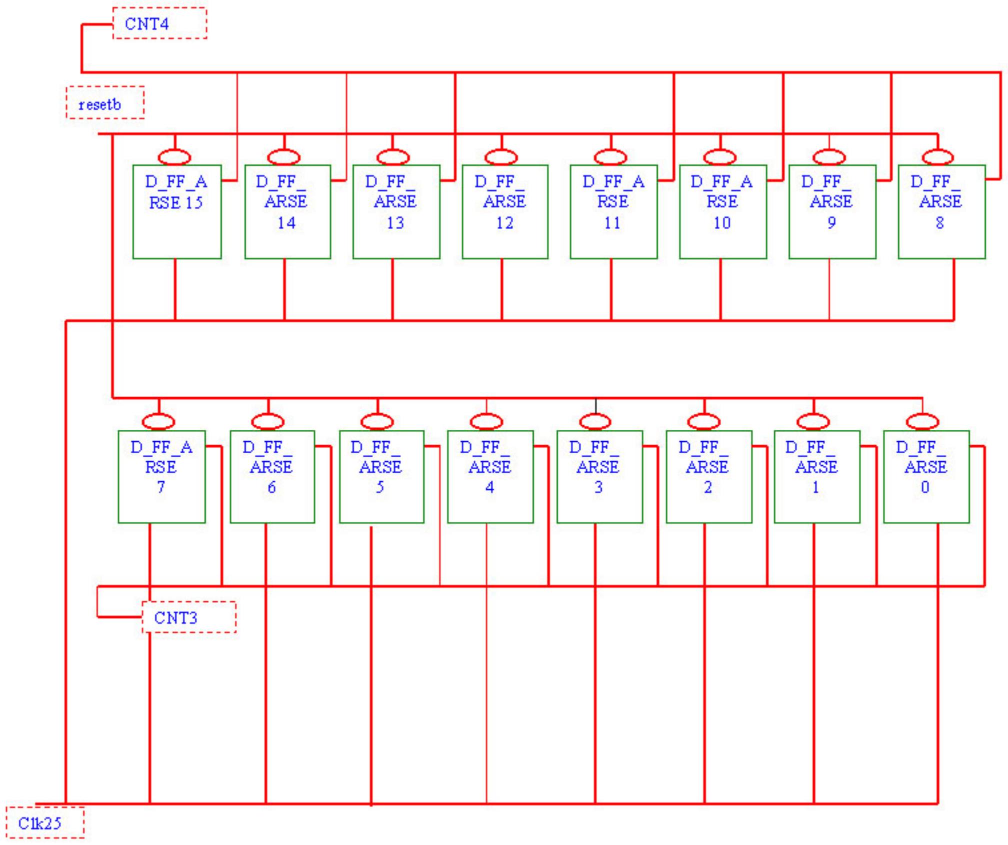


Figure 3.17: Extractor architecture

lation are presented in the extractor section of Appendix A. The extractor test bench (*extractor_tb.v*) used the top-level module instantiation of the extractor module (*extractor.v*), which is actually the only component that is defined in this module. Examining the results more carefully in parallel with the test bench, we can observe that the test bench asserts the *resetb* signal at 10 ns and then de-asserts it at 30ns. This is needed to initialize the FFs with a '0' value in order to begin the extraction. At the same time, *en1* and *en2*, which represent *s_Done3* and *s_Done4* respectively, are set to '0' at time 0ns, then asserted at different times in order to correspond to the arrival of the relevant count control signals, which are at 100ns for *en1* and 140ns for *en2*. As well, at 240ns *en1* is de-asserted and at 320ns *en2* is de-asserted, to allow for the FFs to latch the correct value from the pipeline. Now, as far as the pipeline data_in is concerned, the test bench sets a value at 140ns and another one at 320ns to correspond to TL's LSByte and MS-Byte respectively. As can be seen from the simulation, the TLR contains the correct values after *en2* is de-asserted, which is the total length of the current packet.

3.3.4 Synthesis

The synthesis process produced the hardware components that we defined in our RTL code. We haven't defined the I/O signals in the functional simulation section, so we will take the opportunity to do it now:

- Input: *clk, resetb, c_data_in (pipeline data in), s_Done3 (s_Enable1), s_Done4 (s_Enable2)*
- Output: *s_SPR (s_TLR)*

The synthesis results are located in Appendix A for the extractor section. If we examine these results, we can confirm the fact that we were able to build the components that we had envisioned for the extractor/TLR. Looking first at the top module, the I/O signals outlined above are the ones that make up the black box labeled TLL_EXTRACTOR. Exploring the black box more closely, we identify two parts that have been defined labeled as ARSE_REG_EIGHT_0 and ARSE_REG_EIGHT_1, each of which is an entity representing 8 D_FF_ARSEs along with their respective I/O signals, i.e. *s_Enable1* for ARSE_REG_EIGHT_0, *s_Enable2* for ARSE_REG_EIGHT_1, as well as the distribution of the input bus (*d[7:0] - d[15:8]*) and the output bus (*q[7:0] - q[15:8]*). And finally, we obtained the physical representation of the D FF as defined by Cadence design tools labeled DFCNS1Q, which is what was achieved via the newly generated verilog code containing the net list.

3.3.5 Timing Simulation

Once the synthesis was completed, we proceeded to generate the net list that composes the physical representation of the counter circuit in hardware. Again using the Cadence simulation tools, we re-ran the test bench that was formulated for the functional simulation against the newly generated net list verilog code. The results of the gate level timing simulation are placed in Appendix A in the extractor module section. As can be seen from the results, the final value stored in *SPR* register reflects the data_in values after de-assertion of *en1* and *en2* at specified points in time. Comparing these timing results against the ones obtained through functional simulation, we can clearly declare that both are the same, hence achieving our goal in the design for this particular module.

3.4 Padder

Along the same lines as the counter theory, we can take advantage of the TLR and extractor performing their operation at an early stage in the pipeline. By retrieving the TL of the IP packet, it is possible to calculate the amount of padding bits that are required in order to make the total packet length (CPCS PDU) a multiple of 48, which is required by the ATM layer and hence the SAR. Obtaining the number of padding bits early on in the pipeline became a necessity in order to correctly store the CPCS SDU in memory as this will be explained in the memory module description. Thus, we have defined the need for another module within the TLL module labeled *padder logic* module. This module is solely responsible for calculating the required number of padding bits and storing them in a register for retrieval by the control unit and/or any other module on the pipeline. The design of the padder logic evolved extensively as it was hard to determine the type of method and algorithm to be applied in this case. The basic idea is to be able to calculate the number of padding bits required for each packet. The trick lies in the ability to be able to correctly identify the TLR and use the ALU definition in order to obtain our result. Figure 3.18 illustrates our first iteration at designing the padder logic. The algorithm represented here basically states that the TLR is to be incremented by 8 bytes to account for the trailer, then for the result to be loaded into the accumulator. The accumulator is then checked for negativity; if true, then we have reached our desired number for padding bits, thus perform inverse two's complement on the current contents of the accumulator, and load it into the **Padding Register (PR)**. If the negativity check is false, then we have not yet reached our number, thus keep iterating by subtracting 48 bytes from the accumulator until we reach our desired number. The problem with this approach is that we need to utilize the ALU for operations such as addition and subtraction as well as use the accumulator for temporary storage. This eventually created a bottleneck in our pipeline design as we only have access to only one ALU, which is also being used by the microprocessor for instruction decoding and operand fetching. This will in time generate racing conditions as well as arbitration conflicts with regards to which module is allowed to use the ALU at any given clock cycle.

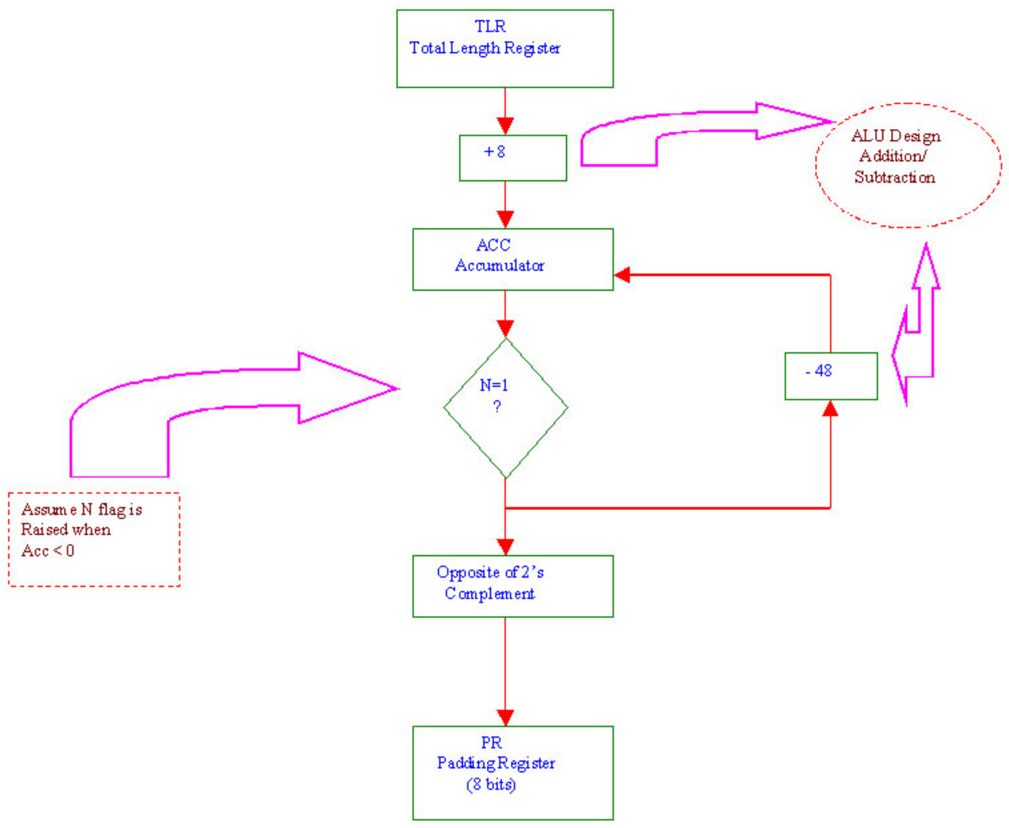


Figure 3.18: First approach to padder algorithm

A better design was hence implemented keeping in mind the above mistakes, and while it was not the final design, it did lead us in the right path. We have taken in consideration here that the ALU can only perform one operation at a time, and that the accumulator should only be used as a temporary register if and only if there is a valid number to be examined, i.e. the number of padding bits is valid. This allowed for the creation of an algorithm that utilization of the ALU by the padder logic module at specified and set periods of the clock cycle, where it is easily predictable what is to occur next in terms of the pipeline and the flow of bytes per packet. The padding algorithm was hence modified to operate in the same manner as the counter algorithm, in terms that the padding algorithm was to be executed only upon the availability of two signals: *s_Done5* and *s_Done6*. These two signals indicate the TLL's processing of byte 5 and byte 6 respectively, which in its turn is only an indication by the counter module that it has reached counting up to value 5 and 6 respectively as well. The algorithm is defined in figure 3.19.

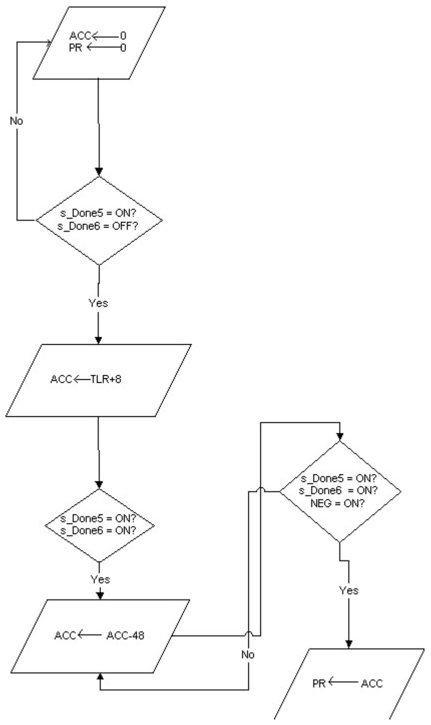


Figure 3.19: Next-generation approach to padder algorithm

As mentioned above, the algorithm is designed to operate around the availability of *s_Done5* and *s_Done6*. These two signals are going to be generated by the counter module and relayed directly to the padder module. *s_Done5* indicates to the padder module that the value in the TLR is valid and that it is available for use. Upon validation of the TLR, the padder module will perform an operational request to the ALU instructing it to add the value in TLR with the numerical value 8. This is necessary in order to account for the addition of the trailer to the CPCS PDU, since padding will have to be calculated with the trailer attached to the PDU. The next clock cycle should allow the processing of the 6th byte on the pipeline, hence the counter reaching a value of 6 and generating *s_Done6*. This signal is implemented in order to allow a grace period or a buffering mechanism for the ALU to perform its instructed operations. But as it turned out, the ALU was designed in combinational logic and not sequential, hence consuming zero clock cycles per operation. The next step in the algorithm is to begin subtracting the value 48 from the accumulator itself, which now contains the TLR. Once we reach a negative value in the accumulator, the padder module will receive the negative flag signal via the control unit. At this very instant, the value held in the accumulator contains the value of the padding bits that need to be inserted into the CPCS PDU to make it a CPCS SDU. The next step would be to perform inverse two's complement on that value, and instantly move it into PR, which is outlined as the last step in the algorithm. Figure 3.20 illustrates the padder logic operations.

Thus, we have clearly identified the three most crucial modules that are to be included in the TLL as separate entities operating within the scope of the "total length" logic. These modules, autonomous as they may be, all depend on each other in order to either begin or terminate operation or kick start another module's operation. Left to include in the TLL is the ALU module and the accumulator module. These two modules will not be designed in the TLL, but merely instantiated in the TLL based on their design that stems from the CPU design discussed later in this document. The padder module will only use the ALU and the accumulator in the TLL.

3.4.1 Functional Specification

As we have now outlined two of the three modules that compose the TLL, we will now examine the most important module needed for the correct operation of Proknet. In very few words, the Padder Logic is responsible for calculating the number of padding bits that are needed to make the IP packet's length a multiple of 48, otherwise referred to as "pad bits" in the introduction. The number of pad bits ranges from 0 to 47 bytes to compensate for the SAR's ability to trigger on 48 byte cells only. The implementation we followed here is the same one outlined in the theory section with minor details to correctly operate the padder logic. Figure 3.21 below is an illustration of the logic implemented to calculate the number of required pad bits.

From the operational behavior of the padder logic defined above, we were able to determine a corresponding algorithm that is to be implemented.

This algorithm was defined using the ASM methodology as was done for the counter module. We will omit the details of ASM process due to its length, but we will include the algorithm in figure 3.22.

The algorithm begins by initializing the various temporary registers to zero by loading them with the '0' value. This is necessary to occur in order to account for any previous values that where not resetted or cleared by the previous iteration of the padder module. The padder remains in the idle state up until the counter has reached the 5^{th} byte, when *s_Done5* becomes asserted. As this state, the padder module will load the value '8' in register 'A' while loading the *s_TLR* value in register 'B'. This is done in order to account for the extra bytes that are added by the trailer in our calculation of the pad bits. Register 'A' and register 'B' are propagated to the ALU module for the addition operation. To note here is that the ALU is designed so that it always perform the addition operation by default until it is instructed to perform a subtraction. For this reason, we don't need to instruct the ALU to add at this state because it is already ready to perform the addition as soon as it receives its inputs. The ALU does not consume any clock cycle while executing its operation, thus we can load the value stored in the accumulator module into the accumulator register defined in the TLL module. At this point we are waiting for *s_Done7* to be asserted in order to move to the next operation. Actually, it was later discovered that this is an obsolete conditional state as our pipeline architecture allowed us to naturally increment from one byte to another, and hence from one count value to another. Yet, we chose to implement it the way we designed it. Hence, once *s_Done7* is asserted, register 'A' is loaded with the accumulator register's value, while register 'B' is loaded with the number '48', where both registers are propagated to the ALU along with the subtraction signal. At this point, we keep checking for the *c_Neg* signal to be generated by the ALU for the subtraction. As long as the *c_Neg* is low, we keep loading register 'A' with the accumulator value, register 'B' with the value '48' and we keep subtracting. Once c_Neg is high at a certain clock cycle, this signals the padder module to move the current value of the accumulator from the accumulator module into register 'temp' in the padder module, which in the previous clock cycle contained the '0' value. The 'temp' register, which now contains a negative number, feeds into the inverse 2's complement module that is to calculate, as the name suggests, the inverse two's complement on the number that is stored in register 'temp' to output the number of padding bits. The inverse two's complement module is designed using combinational logic, thus it will output its result in the same clock cycle to register 'PR', which previously held the '0' value. At this point , register 'PR' will hold its value until *c_DoneC* is asserted, which indicates that we are processing a new packet. This way, the rest of the modules, such as the memory module and the SAR module, will have the ability to access this register.

Following this algorithm, we can define a data path sub-module and a control path sub-module within the padder logic module, so that it is clearly defined what is involved in the data and control paths. We will now examine the implementation of the padder module in hardware.

3.4.2 Hardware Specification

From the description of the functional simulation, we can determine the hardware that is to be implemented for our padder module. Since this is one of many complicated designs in this project, we had to rely on a myriad of components in order to determine what is to be included in each module (data path and control path).

The data path module is responsible for correctly configuring the different data items in order to generate the pad bits. Hence, it is responsible for the implementation of all the relevant registers ('A,' 'B', 'temp', 'PR'), all the multiplexors that are needed to properly switch the data to the adequate register, and the implementation of inverse 2's complement logic. Beginning with the registers, they were created in the same fashion that the counter FFs were created, by using ARSE FFs each controlled by its own control signal. Register 'A' and 'B' utilized 16 FFs created by molding 8 ARSE FFs at one time, assigning them to represent the LSByte and the MSByte of the registers. Register 'A' and 'B' are both controlled by *c_LoadAB*, which becomes the enable signal for their respective FFs. The input to these two registers (i.e., FFs) are the multiplexors that correctly switch the data depending at which state the data path module is at. The input to register 'A' will be the output of the multiplexor toggling between the *s_TLR* value and the accumulator value generated by the accumulator module. Although this register is located in a different module, we had to instantiate it, as is the same case with the ALU, within the TLL module in order to gain access to their functionalities. Thus, the accumulator register is also controlled by a signal labeled *c_LoadAcc*. On the other hand, the input of register 'B' will be the output of a multiplexor that toggles between the value'8' and the value '48'. Both multiplexors are controlled by *s_SelMuxInput* generated again from the control unit depending at the current state. The other two registers are 'temp' and 'PR'. Register 'temp' is controlled (enabled) by *s_LoadTemp*, which gets loaded with the accumulator contents when *c_Neg* is asserted. The output of register 'temp' is fed into the combinational circuitry that generates inverse 2's complement. On the subsequent clock cycle, the output of the combinational circuitry is fed into register 'PR', which is controlled by *s_LoadPR*. The output of register PR is *s_PR*, representing the number of pad bits required for this packet. All of the above mentioned registers are reset globally and loaded with the '0' value internally by the combinational circuitry of *resetb* and *s_ClearRegs*. Let us recap the different I/O signal for the data path module:

- Input: *clk, resetb, s_ClearRegs, c_LoadAb, s_LoadTemp, s_LoadPR, s_LoadPRout, s_SelMuxInput, s_TLR, s_ACC (content of accumulator register)*
- Output: *s_PR, s_PRout. s_inputA (content of register 'A'), s_input_B (content of register 'B')*

The control path contains less variety of components, but nevertheless contains several instances of these same components. Essentially, the control path module is employed to provide all the control signals that are required to operate the data path components. Based on the ASM modeling of the control path illustrated in Figure 3.23, the control signals were implemented using the "one-hot

encoding" technique, which maps each control signal with a flip-flop. If we examine figure **??** closely, we can see that the process is initiated with the assertion of *c_DoneC*, which indicates that we are now processing a new packet. This would cause all of the registers to be cleared by asserting *s_ClearRegs*. This is done by using an AS flip-flop for the same reason explained in the counter module, and that is the fact that the 'reset' flip-flops would need to be differentiated from the others and this is by making them generate a different output whenever they are to be resetted. As stated in the counter module, the AS FF will receive an active-high reset signal that will be mapped as a 'set' signal that sets off a logical high ('1') output when asserted. The other control signals are implemented using AR flip-flops which receive an active-low reset signal that is mapped as a 'reset' signal that sets off a logical low ('0') when asserted. Thus, except for 2 control signals (*s_LoadAB* and *s_LoadAcc*), all of the others do not require any combinational logic to be implemented for their generation. Thus, each state will be using an AR FF to generate their respective control signals, which will ultimately drive the 'enable' signal to the relevant registers and their flip-flops. For example, state 6 labeled *s6* will generate *s_LoadPR* which will be the 'enable' signal driven into registers PR's flip-flops. In similar manner, the other states will generate their corresponding signals.

on the other hand, *s_LoadAB* and *s_LoadAcc* will be generated as a result of combinational OR gates. This is the case because register 'A', 'B', and 'PR' are used by several different stats, hence require to be mapped correctly. Figure 3.24 illustrates the above-discussed circuits.

Let us now recap the different I/O signals for the control path:

- Input: *clk, resetb, s_Done5, s_Done6, s_Done7, s_Done8, c_Neg, c_DoneC*
- Output: *s_ClearRegs, s_LoadAB, s_LoadAcc, s_LoadTemp, s_LoadPR, s_LoadPRout, s_ALUSel, s_SelMuxInput*

If you noticed that there is an extra control signal (*s_LoadPRout*) on the port list that was not discussed earlier, that is quite normal. We decided to add an extra register, labeled PRout, in the padder logic module when we completed the system integration. It was revealed during that run that register PR was not holding its value long enough to be retrieved by the memory module in order for it to operate correctly (the memory module will be discussed in the following section). We noticed that the *c_DoneC* signal was causing the registers to reset quicker than expected. To combat that, we decided to introduce a non-intrusive register whose input is register PR's output, and whose output is the same as register PR's output, i.e. the number of pad bits. The trick with this register is that it is not tied to the reset circuitry that the others are tied to, which means that this register will never be reset. It just follows register PR's output one clock cycle later. This worked as an effective delay for the rest of the modules to be able to latch the contents of PR because upon the assertion of *c_DoneC*. With both sub-modules defined, we can examine the complete I/O signals of the padder module:

- Input: *clk, resetb, c_DoneC. S_Done5, s_Done6, s_Done7, s_Done8, c_Neg, s_ACC, s_TLR*
- Output: *s_ALUSel, s_LoadAcc, s_ClearRegs, s_PR, s_PRout, s_InputA, s_InputB*

3.4.3 Functional Simulation

The functional simulation of the padder module was not executed due to the unmanageable number of I/Os that are used to operate the module. We decided that the integration of the padder module into the TLL module would give us enough coverage that we would be able to debug any problems that would arise from the padder module.

3.4.4 Synthesis

Comparable to the counter module, the padder module achieved the representation of the many components that were born out the hardware specification. As outlined in Appendix A for the padder module, the op module is divided in two components, the data path and the control path. The data path unit itself had several components that it encircled as part of it makeup. Of notice are the 10 ARSE_REG_EIGHT modules that constitute the 5 defined registers, each of which gets broken down to 8 D_FF_ARSE flip flop definition. As well of notice are the 16 muxes that are defined in the data path module. These multiplexors take as input a single bit from the two elements that have to be selected, and output a single bit as a result. Our definition of these multiplexors was written in behavior mode, hence losing all control of the components and thus obtaining the observed results. Squeezed in there after register 'Temp' is TWO_COMPL_INVERSE, which is nothing more than combinational logic circuitry. The control path module is broken up into 7 different state representations, each of which corresponding to a specific control signal. The input to these various state modules is nothing more than the combinational circuitry outlined in figure **??**. it is left up to the reader to verify the synthesis results for the control path.

3.5 ALU

3.5.1 Theory

In this section, we introduce the design of the **arithmetic/logic unit**, or **ALU** for short. This is usually found in the main processing unit, and performs all additions, subtractions, multiplications, divisions and logical operations upon integers. The processing unit controls the ALU by providing it with signals in order to inform it which operations to execute. The ALU's output is usually held in a special purpose register, called the **accumulator**. The width, in bits, of the words (conventionally, two inputs are utilized) that the ALU handles is usually the same as the one used by the external busses.

The ALU is a purely combinational circuit which represents the workhorse of the main processor, and thus must be carefully designed, in order to ensure that it does not pose any bottleneck issues [Lee00]. Since the ALU aids in the instruction execution, it resides on the datapath of the main processor, and feeds, as mentioned, a register (the accumulator) in order to hold the value that it just calculated. The ALU also aids us in minimizing logic gates and routing paths, since it can also be used to handle memory access instructions, which need calculations to be performed in order to be executed. Different algorithms can be implemented within the ALU to speed up the operations, while for even faster execution, multiple ALUs can be instantiated within the circuit to distribute the processing load. The typical ALU consists of an adder that can, with external control, perform other functions. That said, it is now assumed that the adder is the nucleus of the ALU, and the addition operation is the most basic and time-consuming operation. The adder implementation can be one of many: ripple carry-adder, fast adders, carry-skip adders and carry-select adders.

3.5.2 Design History

The ALU design phase started by defining the operations needed for Proknet to perform its duties. What came out of this first discussion, is that a 16-bit full adder, and a 16-bit comparator were the only two circuitries that are needed. The full adder would be used in conjunction with a circuit that implemented a two's complement inversion in order to perform 16-bit subtraction. The fact is that subtraction is just addition of a negative number, which is the reason for the two's complement inversion circuit. The comparator would be a simple one, whose outputs consist of feedback that the first input is either greater than, less than or equal to the second input. This was envisioned to be used in the padding algorithm, where we need to compare to a particular number (48) in order to segment the IP packet into fixed-size payloads.

The 16-bit comparator is presented first. Table 3.1 shows the truth table of a 1-bit comparator. There are four inputs, mainly Ai and Bi, the respective input bits, and PGT (Previous Greater Than) and PLT (Previous Less Than), the memory elements of the circuit. The outputs at each stage are two in number: GT (Greater Than) and LT (Less Than), which indicate that, using previously found decisions, we can currently choose which of the inputs is greater than the other (if at all). The idea is that if the output of each bit decision is made and propagate through to the next bit, at the end of the most significant bit position decision, the GT and LT outputs will define which input is greater than the other (see figure 3.25 for a diagrammatical view of the schematic). If both GT and LT are de-asserted, then the numbers must be equal, and hence external logic has to be built to create an Equal line. From the truth table, we can derive the following equations

Ai	Bi	PGT	PLT	GT	LT
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	0	0

Table 3.1: Comparator circuit truth table

$$GT = \overline{Ai}\overline{Bi}PGT\overline{PLT}+\overline{Ai}BiPGT\overline{PLT}+Ai\overline{Bi}\overline{PGT}\overline{PLT}+Ai\overline{Bi}PGT\overline{PLT}+AiBiPGT\overline{PLT} \tag{3.1}$$

$$LT = \overline{Ai}\overline{Bi}\overline{PGT}PLT+\overline{Ai}Bi\overline{PGT}\overline{PLT}+\overline{Ai}Bi\overline{PGT}PLT+Ai\overline{Bi}\overline{PGT}PLT+AiBi\overline{PGT}PLT \tag{3.2}$$

$$EQ = \overline{GT + LT} \tag{3.3}$$

where it is assumed that LT = 1, when Ai <Bi; and GT = 1, when Ai > Bi.

The 16-bit full adder is also implemented using 1-bit full adder blocks. The truth table of a 1-bit full adder is shown in table **??**. There are two inputs, mainly the previously calculated carry, and the current input. The outputs are the calculated carry and the current sum. Just like in the comparator design, each 1-bit adder is cascaded by another, and thus sixteen instantiations are needed in order to correctly and efficiently operate on 16-bit inputs. The carry out of each block is fed as the carry in of the next block, and thus at the most significant bit position, the carry out is the actual final carry status for the whole 16-bit operation (see figure 3.26 for a

A	Cin	Co	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 3.2: Full Adder circuit truth table

diagrammatical view of the schematic). From the truth table, we can derive the following equations:

$$S = \overline{A}Cin + A\overline{Cin} \tag{3.4}$$

$$Co = ACin \tag{3.5}$$

As a last note for this section, it is worth mentioning that numerous a design cycle were spent in order to attempt the definition of the control signals for the ALU. A myriad of designs were presented and discussed, but their inclusion is only restricted to Appendix H (pgs 20-24). The actual solution to this issue is presented in section 3.4, as the ASM methodology was used to resolve the problem.

3.5.3 Functional Specification

The implemented version of the ALU was majorly modified, as design holes were found in the previous implementations. First of all, the comparator was scrapped in favor of a better padding algorithm (see section 3.4). Second, the full adder was transformed into a **full adder-subtractor (FAS)** through the addition of XOR gates at the inputs of the same 1-bit full adder blocks designed previously. This change allowed the user to add or subtract, through the simple change of a *Select* signal. If the latter is a logical '0', then an addition occurs, because the first 1-bit full adder receives a grounded carry in signal. On the other hand, if the *Select* signal is a logical '1', then a subtraction occurs, because the first 1-bit full adder will receive a carry in of 1, while the B bit input is XOR'ed with that same *Select* signal (logic '1'). The end result, is the two's complementation of the B input, since the XOR acts as a standard inverted, and the initial carry in of 1 forces an addition of 1, as seen in figure 3.27.

Through intuitive analysis, it was shown that if the *Select* signal was a logical '1', and thus the ALU was performing subtraction: if the final carry is itself a logical '1', then the result of the subtraction is positive, otherwise, it is negative. This aids us in the padding algorithm, as we need to determine when the output of the ALU subtraction operations reaches a negative value, for us to make a decision.

The result of the design process is an ALU capable of adding and subtracting two 16-bit numbers, and signaling whether the result is negative or positive. It is controlled by signals discussed in section **??**. The two's complement inversion logic is still needed at the output of the ALU, in order to recover the absolute value of the ALU output.

3.5.4 Hardware Specification

The ALU module was implemented using instantiations of the 1-bit full adder design. The latter implemented a simple full adder circuitry, as seen in figure 1-bit Full Adder. A FASFOUR module instantiated four of these simple 1-bit full adders (FASONEs). The FASFOUR modules were instantiated in a group of four in the main module, effectively instantiating 16 1-bit full adders. The concept was to connect the full adder inputs and outputs as seen in figure 3.27. The top-level ALUFAS module was instantiated and properly connected in the ALU module. The latter included code to generate circuitry for three ALU related flags: the zero flag, the carry flag and the negative flag. The zero flag is easily generated by OR'ing all the outputs of the ALU, thus indicating if we received a zero or not. The carry flag, as previously mentioned, is connected to the final carry out of the full adder-subtractor. Finally, the negative flag is generated as discussed above.

3.5.5 Functional Simulation

Functional simulation was run on the ALU top level module, and the corresponding results are shown in Appendix A. Functional simulation of this module included the instantiation of the ALU module. The top level module of the ALU is termed ALU_TOP, while the test bench module is termed ALU_TOP_TB. The former is instantiated in the latter, and subsequently tested by the variation of the inputs to the module. As we can see from the functional simulation, the ALU receives two 16-bit zero inputs and adds them to get a 16-bit zero output. Then, the inputs are varied to 0x0003 and 0x0005, whence we get a 0x0008 output. Finally, the *Sel* line is changed to a logical '1' which instructs the ALU to subtract (a-b), whence we get a 0xFFFE output, which is the correct result. Taking the two's complete inverse of 0xFFFE, we get 0x0002, thus proving that the ALU functions correctly, since 3 - 5 = -2.

3.5.6 Synthesis

The ALU module was synthesized successfully (for diagrams see Appendix A). The output includes all the schematics for the aforementioned modules. As was expected, all the modules are easily recognizable because of the RTL style of coding for most of the internal architecture.

3.5.7 Timing Simulation

The ALU module was passed through gate level timing simulation, but it failed to produce valid outputs. The final netlist was generated, but the libraries that were called did not match the technology that was installed (and utilized before). This issue was not resolved, and we refer you to section 10.2 for a detailed discussion.

3.6 Accumulator

3.6.1 Theory

The accumulator is, as mentioned in section 3.5.1, a register that is used to store the output of the ALU module. This special purpose register acts as a temporary storage facility that could be queried in order to find out the status of certain actions. The accumulator's value raises several flags, including the negative and the zero flags. The functionality of the accumulator is discussed in section 3.4, and thus will be saved for that section.

3.6.2 Functional Specification

The accumulator is merely a 16-bit register, that is loaded with its value according to its enable signal. If the enable is asserted, then the input is latched on to the output of the register, otherwise the register keeps its old value. Upon global reset, the accumulator is driven all zeroes.

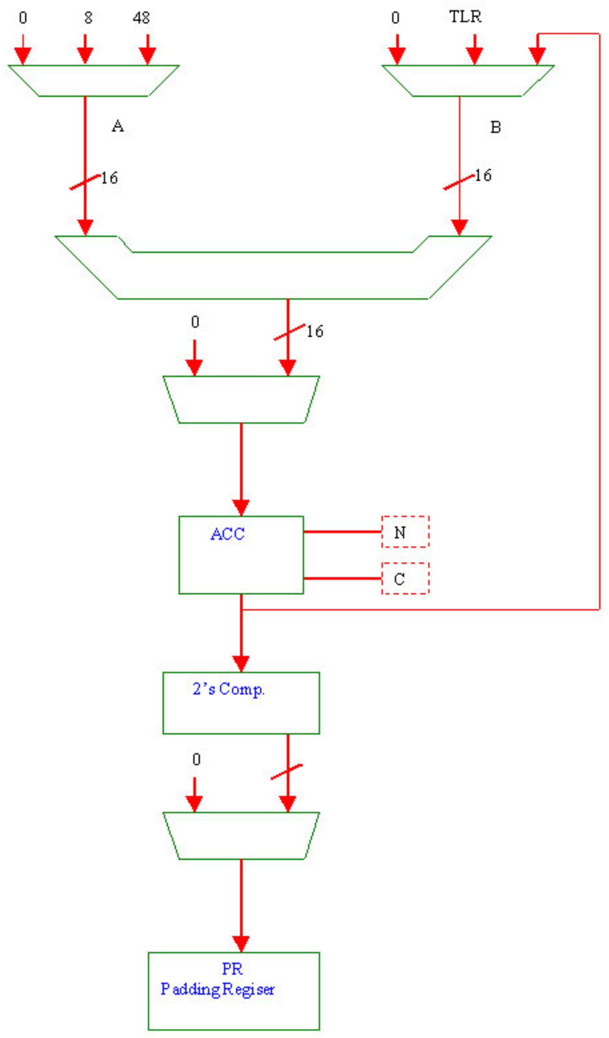


Figure 3.20: Padder logic operations

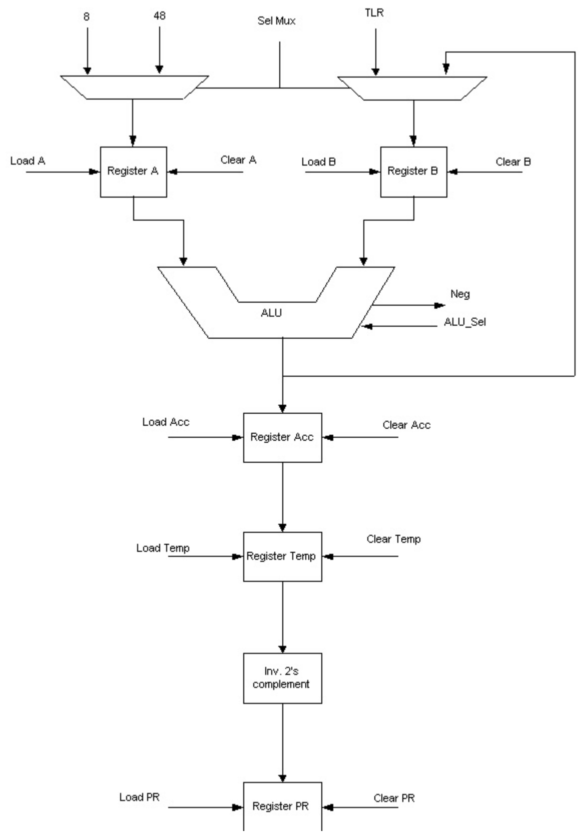


Figure 3.21: Padder operations using the register method

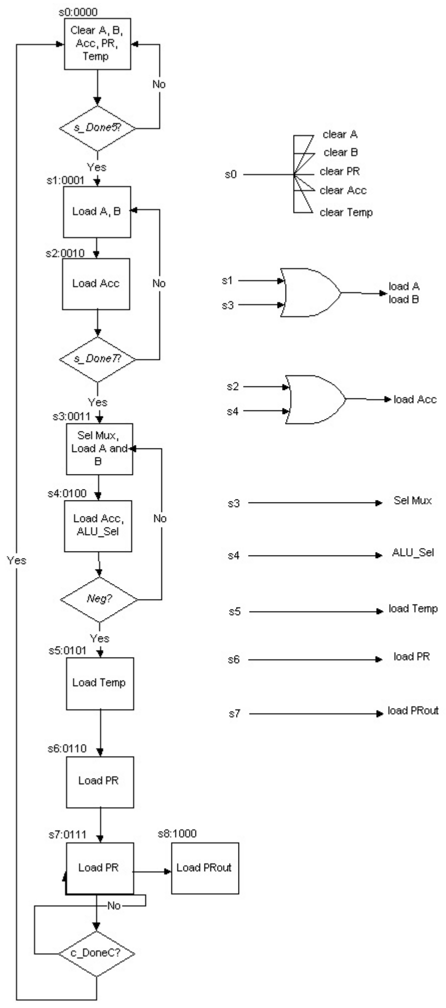


Figure 3.22: Padder algorithm

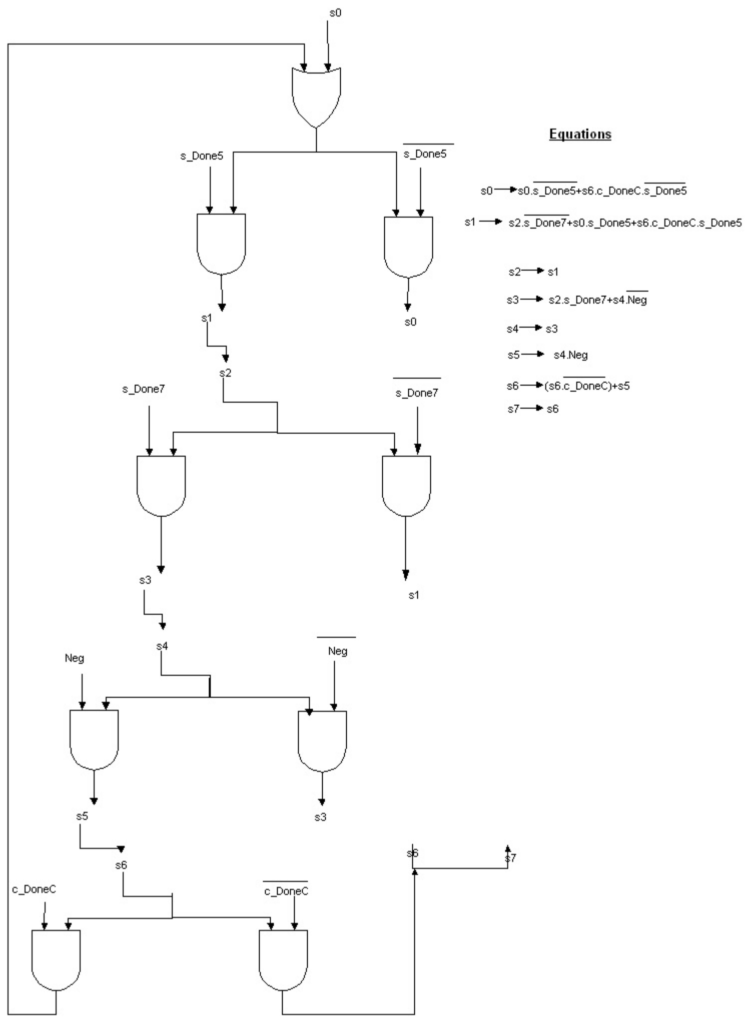

s0
Equations
s_Done5
$\overline{\text{s\_Done5}}$
s0 → s0.$\overline{\text{s\_Done5}}$+s6.c_DoneC.$\overline{\text{s\_Done5}}$
s1 → s2.$\overline{\text{s\_Done7}}$+s0.s_Done5+s6.c_DoneC.s_Done5
s2 → s1
s3 → s2.s_Done7+s4.$\overline{\text{Neg}}$
s4 → s3
s5 → s4.Neg
s6 → (s6.$\overline{\text{c\_DoneC}}$)+s5
s7 → s6
s1
s0
s2
s_Done7
$\overline{\text{s\_Done7}}$
s3
s1
s4
Neg
$\overline{\text{Neg}}$
s5
s3
s6
s6
s7
c_DoneC
$\overline{\text{c\_DoneC}}$

Figure 3.23: Control path using gates

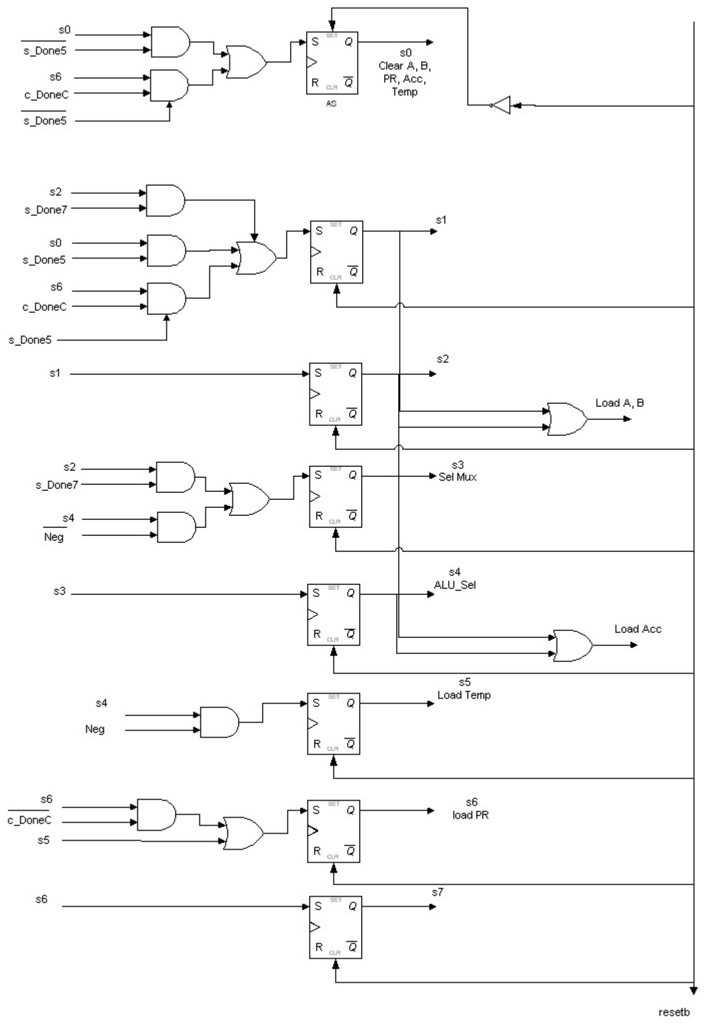


Figure 3.24: Flip-flops of the 7 states of the control path

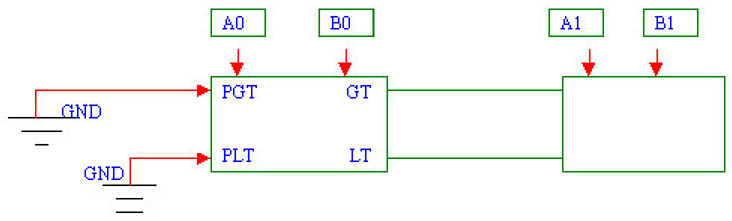


Figure 3.25: Comparator circuitry

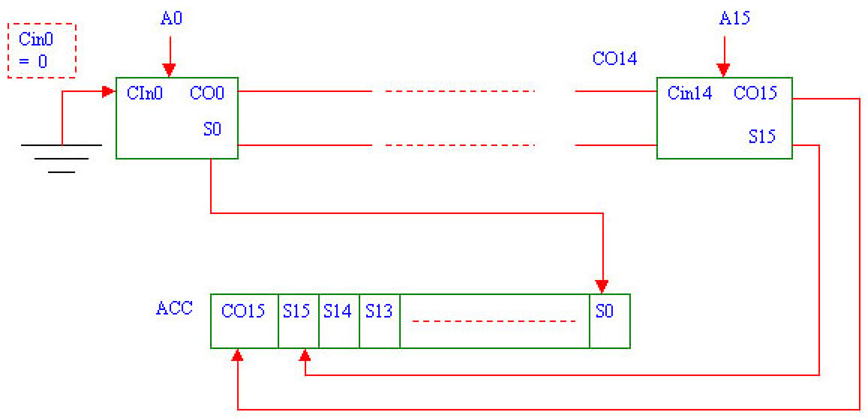


Figure 3.26: Full adder circuitry

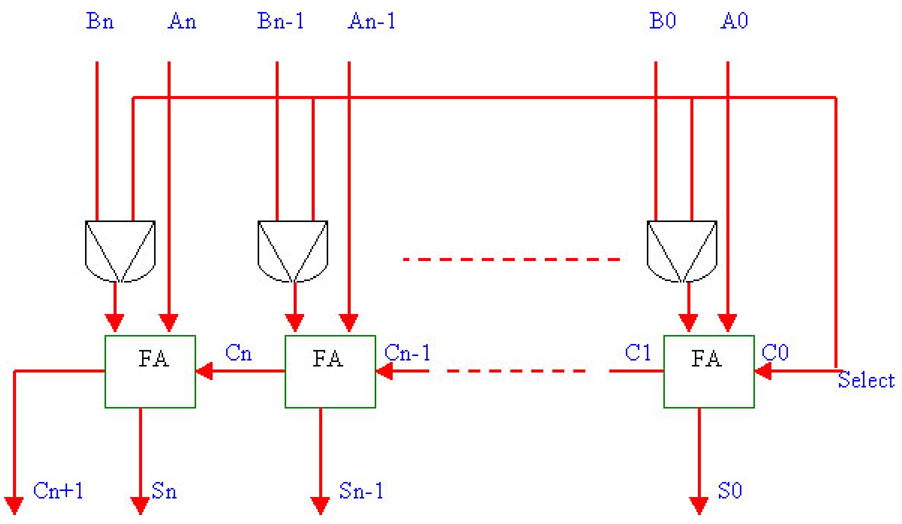


Figure 3.27: Full adder-subtractor circuitry

Chapter 4

FIFO module design

4.1 Theory

The acronym, FIFO, stands for **First-In, First-Out**. In communication systems, a transmitter may send data in bursts faster than the receiver can handle, therefore a FIFO buffer is needed, because it can accept short bursts of high-speed data and then allow data to be read out as needed; the bytes in the data stream remains in order so that the first byte entered into the buffer is the first byte read out from the buffer. Now let's go into more details of the FIFO design.

4.2 Functional Specification

Having introduced the FIFO module, the control unit must control it by the *Trailer_Rd* and *Trailer_Wr* signals. The FIFO module will be responding to the main control unit with the status signals *Empty* and *Full*, see figure 4.1.

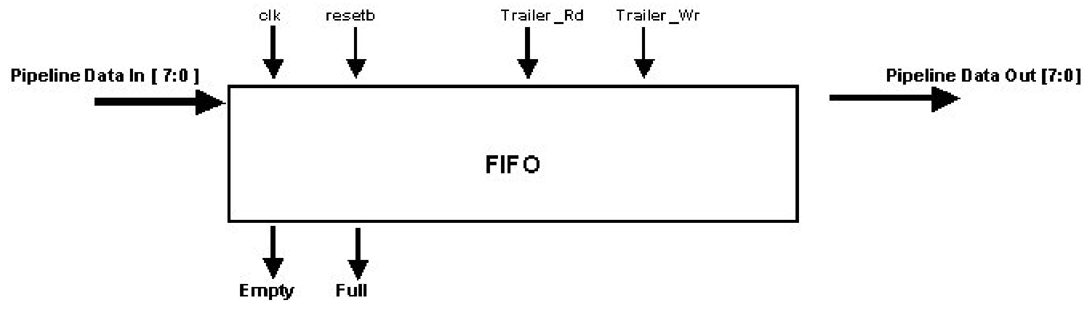


Figure 4.1: I/O of the FIFO module

Within the FIFO buffer itself, we have the following modules (see figure 4.2):

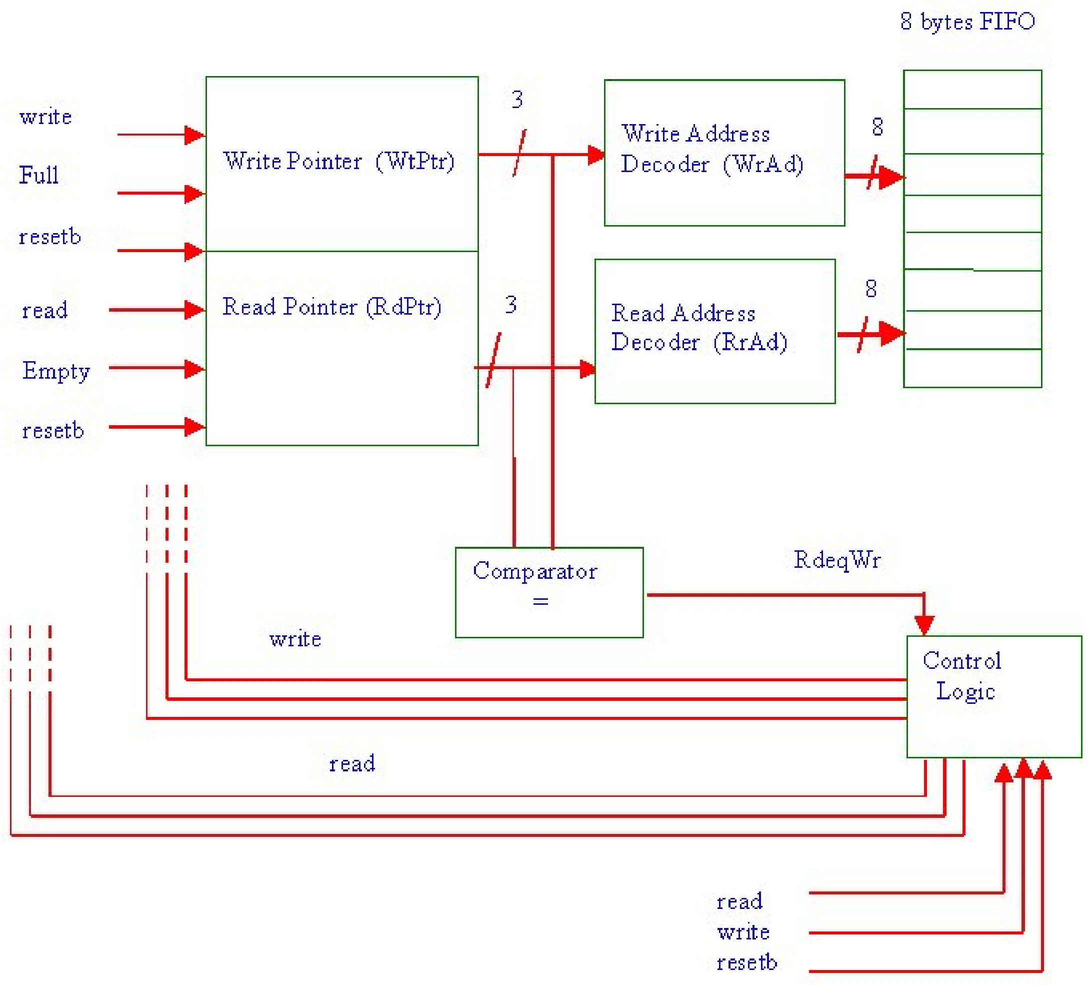


Figure 4.2: FIFO module architecture

Control Logic (CL) needed to control the incrementing of both and read and write pointers depending upon whether FIFO is Empty or Full.

Write Pointer (Wrptr) basically a counter that increments in the wrap-around sequence 0,1,..7, 0,1..,7,0,..(and so on), so that all 8 memories locations can be addressed. A write operation increments the pointer, but not if the FIFO is Full.

Read Pointer (Rdptr) also a counter that increments in the wrap-around sequence 1..,7. A read operation causes the pointer to be incremented, but not if the FIFO is empty.

Comparator compares read and write pointer. If they are equal then this means FIFO could be either Full or Empty.

Write Address decoder (WrAd) needed to decode write pointer and enable the right memory location pointed to by the Write pointer

Read Address decoder (RdAd) is needed to decode read pointer and enable the right memory location pointed to by the Read pointer

FIFO Memory static RAM; it's needed to store the data.

4.3 Hardware Specification

FIFO HW Components

- ⋆ Control logic;
- ⋆ Read pointer(Rdptr , 3 bits ($2^3 = 8$));
- ⋆ Write pointer(Wrptr , 3 bits ($2^3 = 8$));
- ⋆ Comparator;
- ⋆ Read Address decoder (RdAd, 8 bits);
- ⋆ Write Address decoder (WrAd, 8 bits);
- ⋆ FIFO Memory (8x8 bits).

Again some source code was written in behavior, while another part was in gate level.

The control logic (see Appendix B, FIFO_Control_Logic.v) was designed as shown in figure 4.3.

Since the read and write pointers are just pointers triggered by Rd and Wr respectively, they were designed in the same fashion as the 3-bit counter (see Appendix B, FIFO_PTR.v).

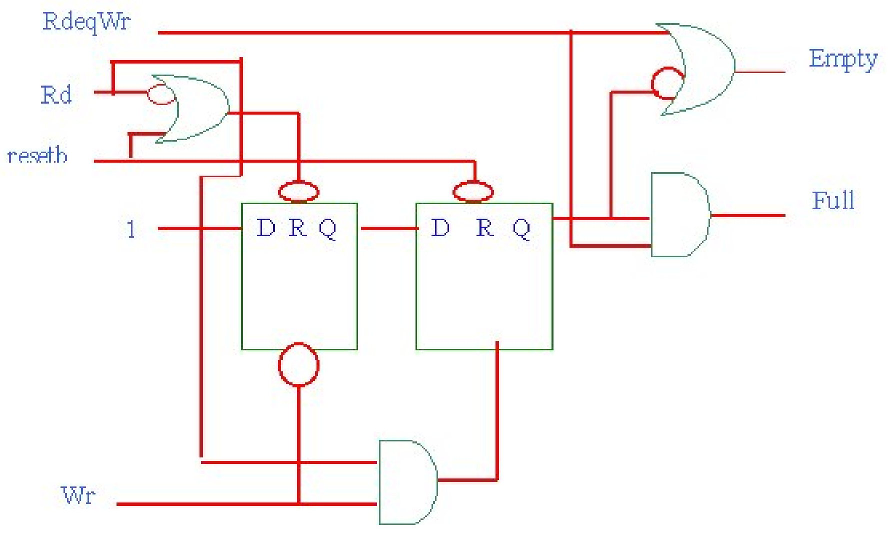


Figure 4.3: FIFO control logic design

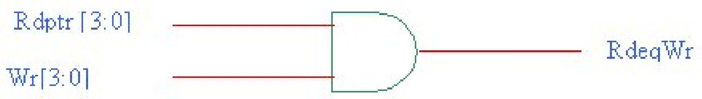


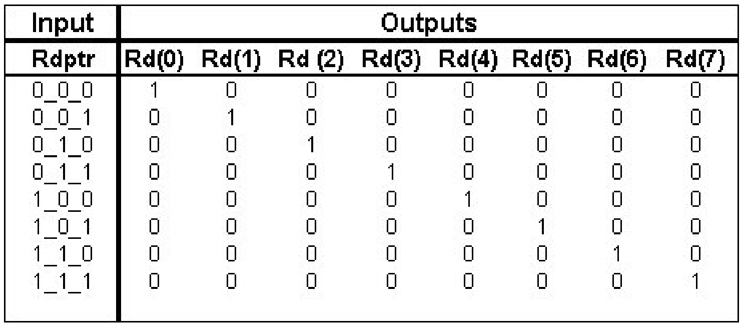
Figure 4.4: Comparator design

A comparator is just a simple AND gate that takes in Rdptr and Wrptr and outputs RdeqWr signal to the control logic (see Appendix B, FIFO_Comparator.v). See figure 4.4.

An address decoder for Read and Write was simply computed through a truth table

Rd(0) corresponds to the first memory location in the FIFO. So if Rd(0) is high, then we are reading form the first location in the memory. The Write address decoder functions in exactly the same way (see Appendix B, FIFO_Address_Decoder.v).

Finally the FIFO memory (the RAM); this one is very simple to design, since it is simply a number of flip-flops, with a tristate buffer on the output of each flip-flop (see figure 4.5. It is dual port memory that allows the write and the read signals to be executed at the same time. (see Appendix B, FIFO_Memory.v)



Input	Outputs							
Rdptr	Rd(0)	Rd(1)	Rd (2)	Rd(3)	Rd(4)	Rd(5)	Rd(6)	Rd(7)
0_0_0	1	0	0	0	0	0	0	0
0_0_1	0	1	0	0	0	0	0	0
0_1_0	0	0	1	0	0	0	0	0
0_1_1	0	0	0	1	0	0	0	0
1_0_0	0	0	0	0	1	0	0	0
1_0_1	0	0	0	0	0	1	0	0
1_1_0	0	0	0	0	0	0	1	0
1_1_1	0	0	0	0	0	0	0	1

Table 4.1: Truth table for address decoder

4.4 Functional Simulation

A test bench was written to test the FIFO module functionality (see Appendix B, TB_FIFO.v and functional simulation TRAILER.FIFO). This test bench checks if the FIFO is able to handle the write and read signals from the control unit. Looking at the timing diagram. First, we input a byte and on the next clock cycle it is read with no problem, on the third clock cycle, a second byte is on pipeline, but the read pointer is pointing to another location, therefore this byte is not read. So by looking at the functional simulation we have proved that the FIFO is functionally correct.

4.5 Synthesis

The FIFO module was synthesized successfully (for diagrams see Appendix B). The output includes all the schematics for the aforementioned modules. As was expected, all the modules are easily recognizable because of the RTL style of coding for most of the internal architecture.

4.6 Timing Simulation

The FIFO module was passed through gate level timing simulation, but it failed to produce valid outputs. The final netlist was generated, but the libraries that were called did not match the technology that was installed (and utilized before). This issue was not resolved, and we refer you to section 10.2 for a detailed discussion.

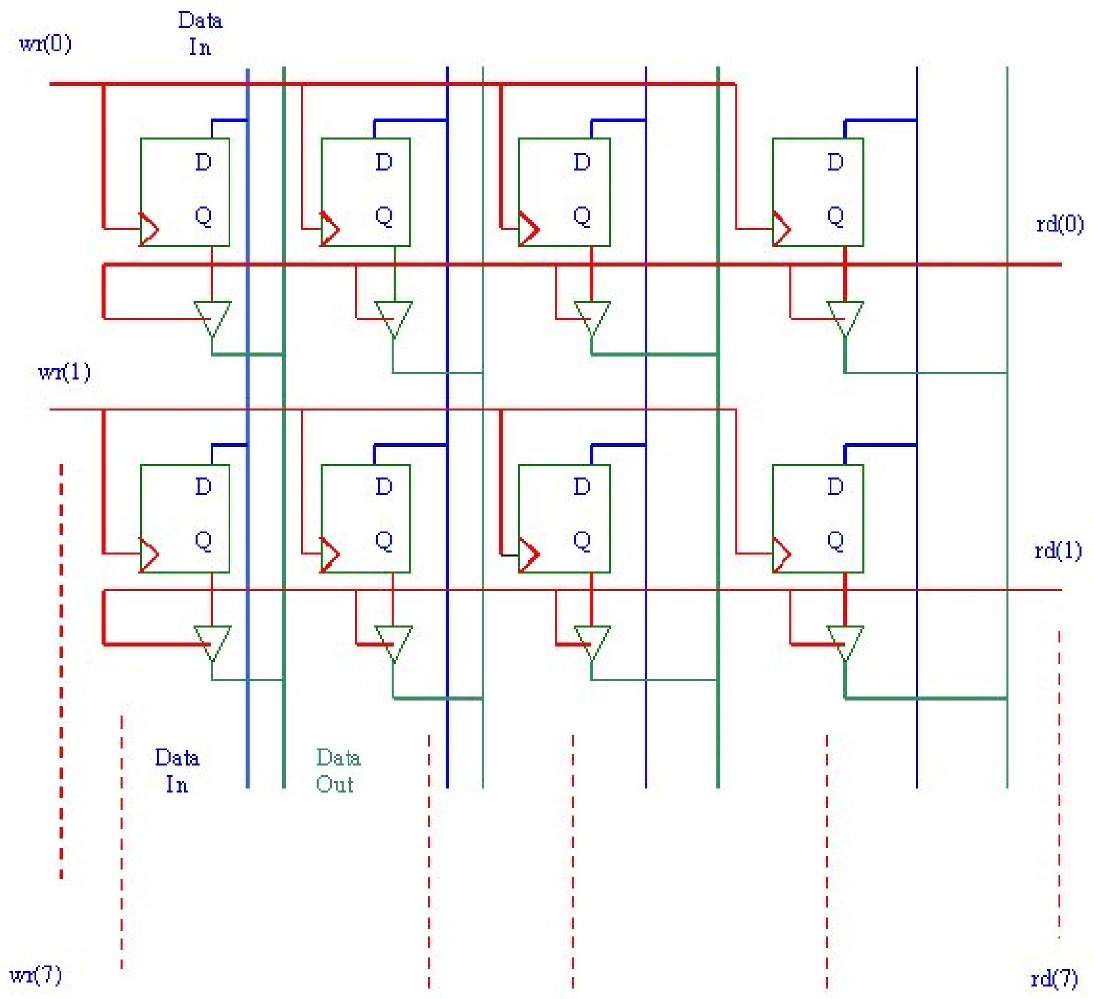


Figure 4.5: FIFO memory design

Chapter 5

Trailer module design

5.1 Theory

The trailer is an 8-byte field that will get appended to the end of the packet. The trailer contains 4 fields, as seen in figure 5.1. The fields are described below:

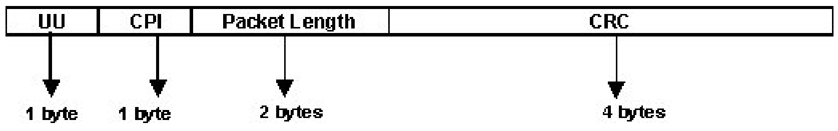


Figure 5.1: Trailer overview

UU (User to User) The field contains one octet of information, which will be transferred transparently between users of the AAL5 CPCS users

CPI (Common Part Indicator) The field is used to interpret subsequent fields for the CPCS functions in the CPCS header and trailer

Packet Length This field contains the size of the incoming packet.

CRC-32 This field is filled with the value of a CRC calculation which is performed over the entire content of the packet. It is filled with zeros before going to the CRC circuitry. Once passed through the CRC-32 circuitry, it will be filled with the correct values and then transmitted (see section 6.1.1 for more details).

The main concerns in this project are the packet length field and the CRC field. Packet length is used to calculate how many bytes the packet will be padded with, in order for it to be divisible by 48 bytes to achieve our goal (IP over ATM segmentation). Therefore after our packet has been passed through the Trailer and CRC modules, it will be stored in the Packet Memory and it will have the format shown in figure 5.2. PAD is the field used to align the Packet on a 48 octet boundary. Its size varies from 0 to 47 unused octets, but it does not convey any information.

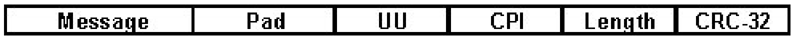


Figure 5.2: Packet in memory

5.2 Functional Specification

The trailer module appends 8 bytes at the end of the incoming packet and feeds the new packet into the CRC module, the latter located next on the pipeline. The trailer module allows the packet to pass through and waits for the last byte of packet to be received, when it appends the trailer. In this project the trailer fields are initialized according to figure 5.3.

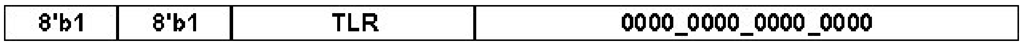


Figure 5.3: Trailer field initializations

The **TLR (Total length Register)** that was extracted from the packet by the TLL module, is stored in the third and fourth byte positions. The CRC field is initialized to zeros because of the CRC algorithm requirements.

5.2.1 Trailer I/O

The trailer module takes in a variety of inputs from the control unit and the pipeline (see figure 5.4)

Examining the figure, the horizontal lines are inputs and outputs of the pipeline. The vertical lines represent inputs and outputs of the main control unit. In order to achieve the implementation of the trailer, we need the following modules:

FIFO To store the next packet while the trailer module is appending the trailer on the previous packet, and since we have 8 bytes of trailer, we will need a 8x8 byte FIFO memory

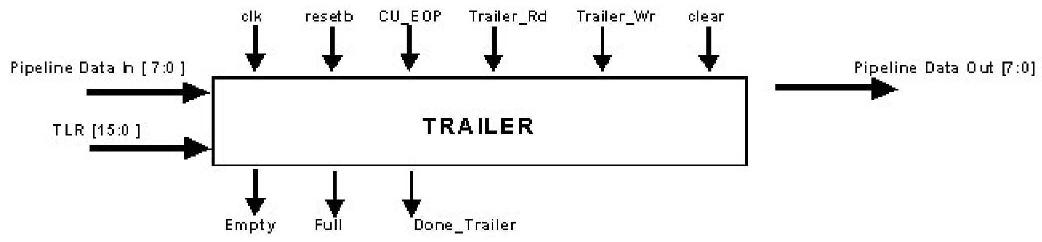


Figure 5.4: I/O of the trailer module

8-1 input mux To allow the selection of the trailer fields one at time

3-bit counter To control the select signals of the 8-1 input mux, i.e. it is the select bus.

2-1 input mux As you can see (figure 5.1 we already have two routes in the design and only one of them can be sending bytes out to the pipeline, thus, the introduction of this multiplexer will help resolve the issue.

5.2.2 Trailer Operation

Please refer to figure 5.5 for a diagram of the following explanation:

1. When a byte of a packet is received from the TLL module, the control unit writes into the 8-byte FIFO

2. On the next clock cycle, the byte that was written in step 1, gets read and at the same time another byte is written to the FIFO

3. Step 1 and 2 are repeated until the *CU_EOP* is set to high by the control unit. When *CU_EOP* is set high, two things happens. First, the 2-1 input mux switches up and secondly, the 3-bit counter starts selecting the select signals for the 8-1 input mux

4. The counter keeps counting until all trailer fields are mux-ed out and have been put on the pipeline, then the counter sends a *Done_Trailer* signal to the control unit, informing the latter that the trailer has been appended to the packet.

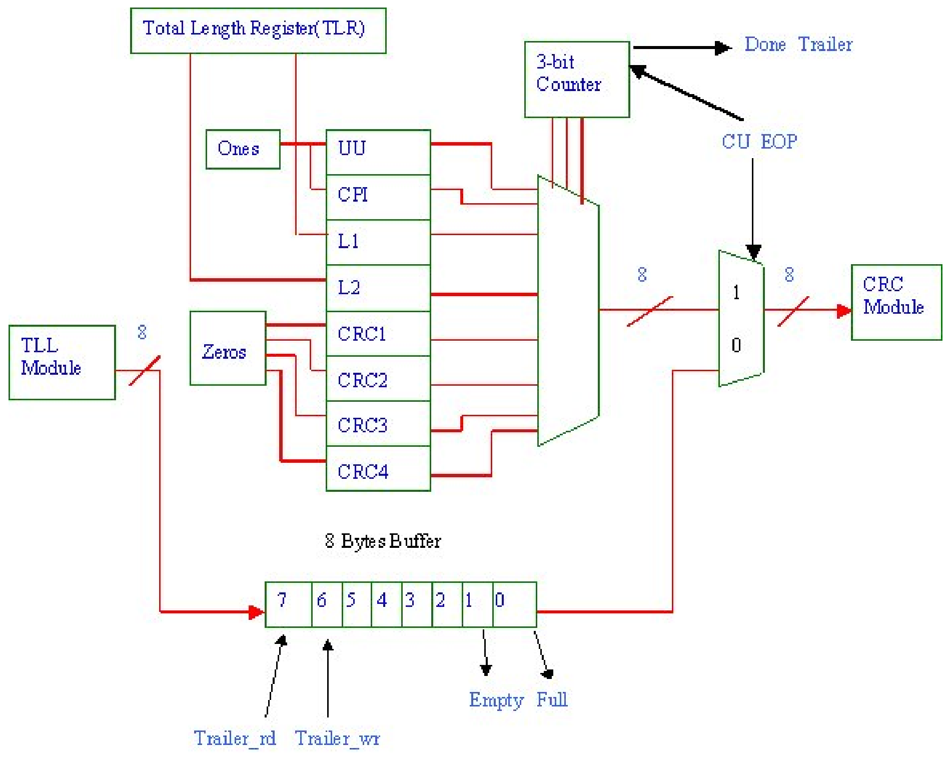


Figure 5.5: Trailer module architecture

5.3 Hardware Specification

Trailer HW Components

- ⋆ 8 to 1 input multiplexer (8 bits input);
- ⋆ 2 to 1 input multiplexer (8 bits input);
- ⋆ 8-byte FIFO (8x8 bits);
- ⋆ 3-bit counter.

In this project some of the code was written in *register-transfer logic (RTL)*, behavioural coding and even in gate level.

Here's an example of how the multiplexers were derived (see figure 5.6 :

Now the above mux is just for a 1 bit input (see Appendix B, Mux2_1.v), in order to have 8 bits inputs. This mux was instantiated 8 times (please refer to Appendix B mux21_7_0.v for a 2 input mux).

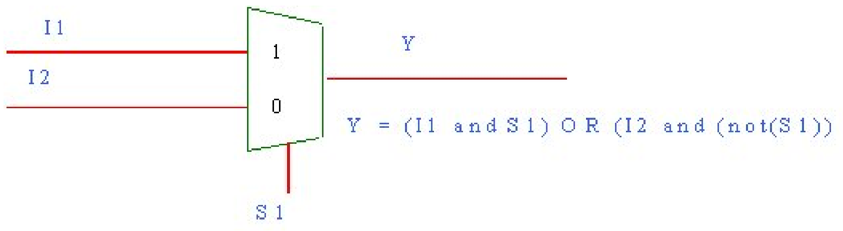


Figure 5.6: 2 to 1 input multiplexer

For the 8 input mux, the design was approached exactly the same way as the 2 input mux (see Appendix B, Mux8_1.v and Mux81_7_0.v).

For details on the FIFO design, please refer to section 4.1. In this project, the 3-bit counter was designed behaviorally (please see Appendix B, trailer_counter.v). The pointer is incremented when it receives *CU_EOP* and it keeps incrementing until *CU_EOP* goes low.

Having understood the design and operation of the trailer, the trailer module was implemented in Verilog please refer to Appendix B, TRAILER.v. The code is a simple instantiation of all the modules that were described above.

5.4 Functional Simulation

After the trailer was implemented, a test bench was written in order to test the trailer module's functionality. Please refer to Appendix B for the test bench and for the corresponding functional timing diagram. In the test bench we attempt to set *CU_EOP* high first (which indicates that it's time to mux the the trailer on the pipeline), then we see the all 8 fields of trailer are on the Pipeline_Data_Out. Once *trailing* is done, the *Done_Trailer* signal is set high (indicating that all trailer fields are out on the pipeline). In order to test that the FIFO is properly working, the test bench writes two bytes into the FIFO and after two clock cycles, the read signal is set high, therefore the two bytes are on the pipeline (PP_Data_Out). The timing diagram simulation proves that everything was well implemented and successfully working.

5.5 Synthesis

The trailer module was synthesized successfully (for diagrams see Appendix B). The output includes all the schematics for the aforementioned modules. As was expected, all the modules are easily recognizable because of the RTL style of coding for most of the internal architecture.

5.6 Timing Simulation

The trailer module was passed through gate level timing simulation, but it failed to produce valid outputs. The final netlist was generated, but the libraries that were called did not match the technology that was installed (and utilized before). This issue was not resolved, and we refer you to section 10.2 for a detailed discussion.

Chapter 6

CRC module design

6.1 Theory

The **cyclic redundancy check (CRC)** is a number derived from, and stored or transmitted with, a block of data in order to detect corruption. By recalculating the CRC and comparing it to the value originally transmitted, the receiver can detect some types of transmission errors. A CRC is more complicated than a checksum. It is calculated using division either using shifts and exclusive ORs or table lookup (modulo 256 or 65536). The CRC is "redundant" in that it adds no information. A single corrupted bit in the data will result in a one-bit change in the calculated CRC but multiple corrupted bits may cancel each other out. Most CRC implementations seem to operate 8 bits at a time by building a table of 256 entries, representing all 256 possible 8-bit byte combinations, and determining the effect that each byte will have. CRCs are then computed using an input byte to select a 16- or 32-bit value from the table. This value is then used to update the CRC.

CRC is often used because it is easy to implement and it detects a large class of errors. For any given message, CRC can detect the following:

- All one or two bit errors
- All odd numbers of bit errors
- All burst errors less than or equal to the degree of the polynomial used
- Most burst errors greater than the degree of the polynomial used

In a system employing CRC, the message being transmitted is considered to be a binary polynomial M(X) (which is the message or payload). It is

CRC 16	X^16 + X^15 + X^2 + 1
SDLC (IBM, CCITT)	X^16 + X^12 + X^5 + 1
CRC 12	X^12 + X^11 + X^3 + X^2 + X + 1
CRC 16 Reverse	X^16 + X^14 + X + 1
LRCC 16	X^16 + 1
LRCC 8	X^8 + 1
CRC 32	X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10 X^8+X^7+X^5+X^4+X^2+X+1

Table 6.1: CRC algorithm

first multiplied by X^k and then divided by an arbitrary generator polynomial G(X) (also known as the predetermined divisor) of degree k (k is the number of bits in G(X)), which results in a quotient Q(X) and a reminder R(X)/G(X). All arithmetic is done in modulo-2. This process is shown in the following equation, in which $\oplus$ is the sign for addition in modulo-2 arithmetic:

$$\frac{X^k M(X)}{G(X)} = Q(X) \oplus \frac{R(X)}{G(X)} \tag{6.1}$$

In modulo-2 arithmetic, the results of subtraction are equivalent to the results of addition. By applying this property and some simple algebra to the equation, we get

$$X^k M(X) \oplus R(X) = Q(X)G(X) \tag{6.2}$$

R(X) will always be of degree k or less.

The CRC algorithm calculates R(X) and appends it to the message being sent. Since X^{k}M(X) $\oplus$ R(X) equals Q(X)G(X), the original message with the CRC appended will be evenly divisible by G(X), if and only if no bits are changed. At the receiving end, the received message (original message plus R(X)) is divided by the generator polynomial G(X). If the remainder is zero, it is assumed that no errors have occurred. If the remainder is zero, it is assumed that no errors have occurred or that an error has occurred but has gone undetected by the algorithm. A list of commonly used generator polynomials is shown in the following table:

The CRC-16 polynomial is a common standard used around the world (it is the polynomial used in the Bisync protocol). SDLC synchronous data link control is used by IBM and is the standard in Europe. The CRC-12 polynomial is used with six bit bytes. The CRC-32 is used in the field of

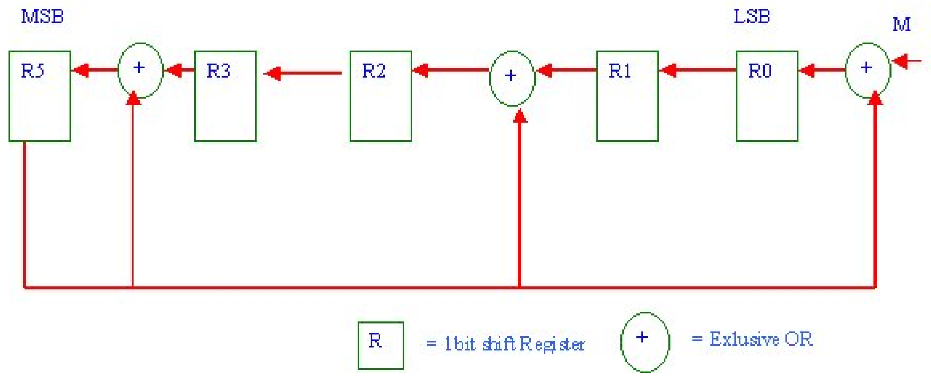


Figure 6.1: Logic circuitry

telecommunication. It is mostly used in checking the errors in the **AAL5 (ATM Adaptation Layer 5)** messages. However ATM CRC-32 is difficult to use because it is based on a polynomial of degree 32 that has many more terms (15) than any other CRC polynomial in common use. CRC checking and generation are generally carried out on a per-byte basis, in an attempt to cope with the dramatic increase of the data throughput of higher-speed lines.

Since CRC arithmetic is done in modulo-2, it can be easily implemented in hardware with shift registers and exclusive OR gates. Each flip-flop contains one bit of the CRC register. Most software routines emulate the hardware method, thus operating on one bit at a time. Since most processors are not bit-oriented, the bit-wise software approach requires lengthy periods of CPU time. Given that many microprocessors are byte-oriented, an algorithm to calculate CRC on a byte-by-byte basis would be of a great benefit.

Let's take an example to make sure that the logic CRC implementation is well understood. Assume we have the following :

- M(X) = 1010001101 (10 bits) (This is going to be the input to our circuit)
- G(X) = 110101 (6 bits) = $X^5 + X^4 + X^2 + 1$ (which will represent the logic circuitry)
- R(X) to be calculated (5 bits)

As you can see the the Remainder R(X) = 1110 which are the content of (R4, R3,R2,R1,R0) . So the transmit message to the receiver is

	R4	R3	R2	R1	R0	R4 + R3	R4 + R1	R4 + M	M	
Initial	0	0	0	0	0	0	0	1	1	
Step 1	0	0	0	0	1	0	0	0	0	
Step 2	0	0	0	1	0	0	1	1	1	
Step 3	0	0	1	0	1	0	0	0	0	
Step 4	0	1	0	1	0	1	1	0	0	Message to be sent
Step 5	1	0	1	0	0	1	1	1	0	
Step 6	1	1	1	0	1	0	1	0	1	
Step 7	0	1	1	1	0	1	1	1	1	
Step 8	1	1	1	0	1	0	1	1	0	
Step 9	0	1	1	1	1	1	1	1	1	
Step 10	1	1	1	1	1	0	0	1	0	
Step 11	0	1	0	1	1	1	1	0	0	
Step 12	1	0	1	1	0	1	0	1	0	Five Zeros added
Step 13	1	1	0	0	1	0	1	1	0	
Step 14	0	0	1	1	1	0	1	0	0	
Step 15	0	1	1	1	0	1	1	0		

Table 6.2: CRC example

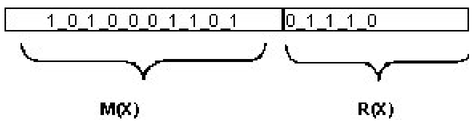


Figure 6.2: Transmitted message

So to finish up with this example, the receiver has exactly the same hardware circuitry. So it will (M(x)R(X)) / G(X) , and if the remainder is zero therefore no errors have been found.

6.1.1 Byte-wise CRC

Since we want to calculate the CRC eight bits at a time, we need an algorithm that will produce the same CRC value as would occur after eight shifts of a bit-wise CRC calculation. There is already a well-established algorithm developed to do the byte-wise CRC. Simply the contents of the CRC register after eight shifts are a function (exclusive-OR) of various combinations of the input data byte and the previous contents of the CRC register. Following the byte-wise CRC algorithm [Ibe97], we managed to develop the following table.

The above table shows the CRC register for each of the eight shifts. The notation used is as follows:

* The ”SH” column is the shift number

Shift	In	CRC registers															
		R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17	R16
8	M8	M5	M7	M7	M6	M7	M6	M3	M7	M6	M0	M5	M4	M7	M7	M6	M5
		C23	M4	M6	M5	M5	M4	M2	M2	M1	C14	C13	C12	M3	M6	M5	M4
		C29	C22	M3	M2	M4	M3	C17	M1	M0	C24	C29	C28	C11	M2	M1	M0
			C28	C21	C20	M1	M0	C26	C16	C15				C27	C10	C9	C8
			C31	C27	C26	C19	C18	C27	C25	C24				C31	C26	C25	C24
				C30	C29	C25	C24		C26	C25					C30	C29	C28
				C31	C30	C28	C27		C30	C30					C31	C30	C29
						C30	C28										
						C31	C30										

Table 6.3: CRC equation calculations (Most significant 16 bits)

Shift	In	CRC registers															
8		R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
		M7	M7	M7	M6	M4	M5	M5	M4	M7	M7	M7	M6	M7	M7	M7	M6
		M5	M6	M6	M5	M3	M4	M3	M5	M6	M6	M6	M4	M3	M6	M6	M0
		M4	M4	M5	M4	M1	M2	M2	M1	M3	M5	M5	M3	M2	M2	M1	C24
		M3	M3	M3	M2	M0	M0	M1	M0	M2	M4	M4	M2	M1	M1	M0	C30
		C7	M2	M2	M1	C3	C2	C1	C0	M0	M2	M3	M0	C25	M0	C24	
		C27	C6	M1	M0	C24	C24	C25	C24	C24	M1	M1	C24	C26	C24	C25	
		C28	C26	C5	C4	C25	C26	C26	C25	C26	C25	M0	C26	C27	C25	C30	
		C30	C27	C25	C24	C27	C27	C28	C27	C27	C26	C24	C27	C31	C26	C31	
		C31	C28	C28	C25	C28	C29	C29	C28	C29	C28	C25	C28		C30		
			C30	C30	C26					C30	C29	C27	C30		C30		
			C31	C31	C28						C30	C28			C31		
					C29						C31	C29					
					C30							C30					
												C31					

Table 6.4: CRC equation calculations (Least significant 16 bits)

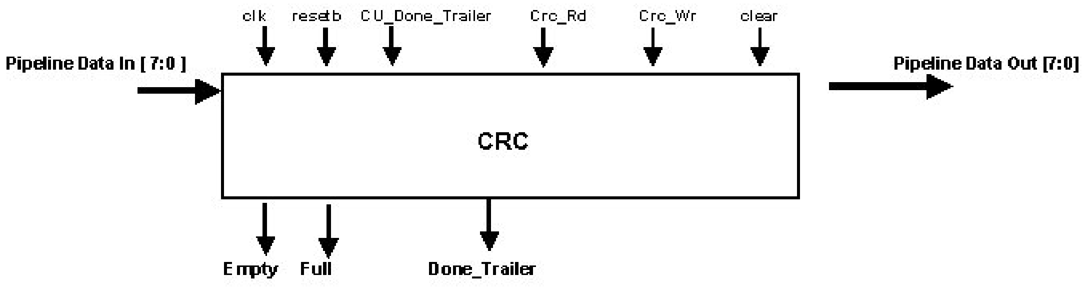


Figure 6.3: I/O of CRC module

* The "IN" column is the data in, with M_i being the ith bit of the current byte of message M(X)

* R_i is the ith bit of the CRC register

* C_i is the ith bit of the initial CRC register, just before any shifts due to the current input byte

* Vertical entries in the R_i columns denote that the entries are to be exclusive-Ored to form the contents of each R_i

6.2 Functional Specification

In order to implement the CRC-32 into our pipeline, the CRC module will have the following inputs and outputs (see figure 6.3):

The horizontal lines represents inputs and outputs of the pipeline, while the vertical lines represent input and output signals of the control unit.

A FIFO buffer will be used in the CRC for two reasons. First, the trailer might be transmitting faster than the CRC can handle (therefore congestion must be avoided); second, the packet must pass through and the last 4 bytes of trailer (which are zeros) must be held in the FIFO, while the 4 *good* bytes of CRC module (which is R(X)) are put on the pipeline. To make our CRC module more effective, the following components were introduced (see figure 6.4):

- 2 to 1 input mux
- 4 to 1 input mux

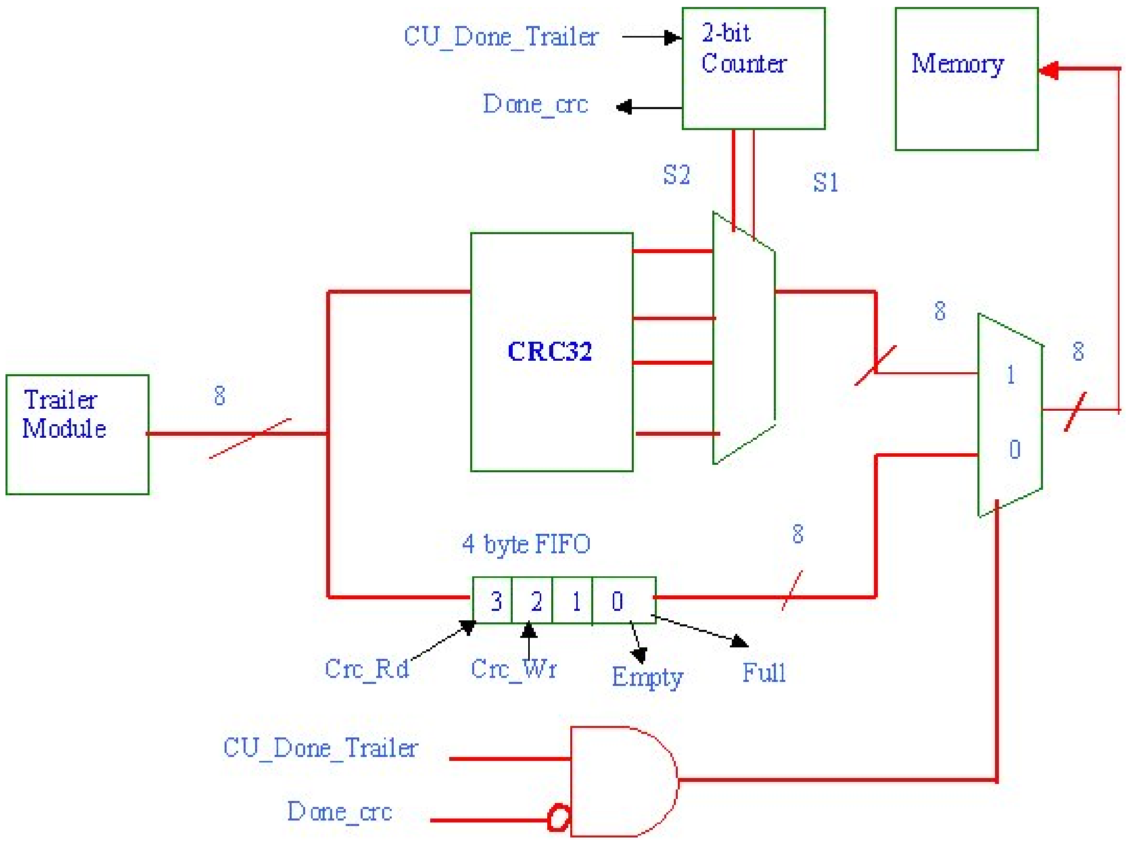


Figure 6.4: CRC module architecture

- 4 byte FIFO
- CRC-32 (It is an implementation of tables that were introduced in section 6.1.1)

6.2.1 CRC Components and Operation

1. Now say that the trailer module sends the first byte of packet, thus the control unit writes it to the 4 byte FIFO and this is done by asserting *Crc_write*
2. On the next clock cycle, a second packet comes in, the control unit will write it in the FIFO, and at the same time it will read the first byte. These actions are done by asserting *Crc_write* and *Crc_Read* at the same time
3. Steps 1 and 2 are repeated until *CU_Done_Trailer* is asserted, and, when this happens, two things are triggered: first, the 2 to 1 input multiplexer and the 2-bit counter are enabled, which basically means it is time to output the 4 bytes of CRC (known as predetermined divisor). Therefore, *CU_Done_Trailer* will stay high for 4 clock cycles to allow the 4 bytes to be placed on the pipeline.
4. Once all 4 bytes are put on the pipeline, the 2-bit counter sets *Done_crc* signal high (meaning that the CRC module has finished its job).

6.3 Hardware Specification

Trailer HW Components

- ⋆ 4 to 1 input mux (8 bits input);
- ⋆ 2 to 1 input mux (8bits input);
- ⋆ 4 byte FIFO (8 bits x 4);
- ⋆ CRC32 module.

Some of the design in Verilog was done behaviourally , RTL and in gate level.

The 4 input mux was designed as shown in figure 6.5.

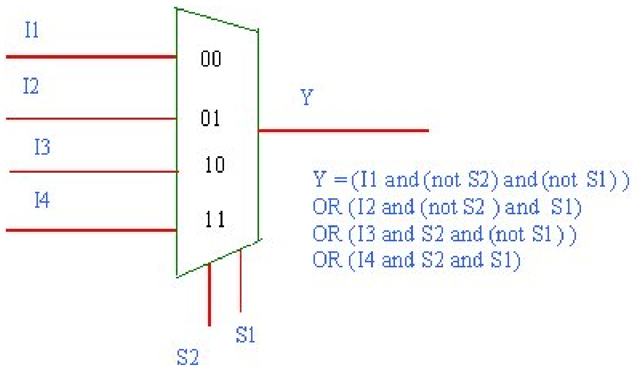

I1
I2
I3
I4
00
01
10
11
Y
Y =(I1 and (not S2) and (not S1))
OR (I2 and (not S2) and S1)
OR (I3 and S2 and (not S1))
OR (I4 and S2 and S1)
S2
S1

Figure 6.5: 4 to 1 multiplexer

The design shown (see Appendix C, mux4_1.v) is only 1 bit input. In order to have 8 bits input, this mux was instantiated 8 times (see Appendix C mux41_7_0.v). The same concept was used to design the 2 to 1 input mux.

The 4 byte FIFO was designed exactly the same as the one designed in the trailer module (see section 4.1), except this fifo is 4 bytes wide. Therefore, the only changes that were made are: FIFO HW Components

- ⋆ Control logic (no changes) (see Appendix C, fifo_control_logic.v);
- ⋆ Read pointer (Rdptr , 2 bits (2 2 = 8)) (see Appendix C, fifo_ptr.v);
- ⋆ Write pointer (Wrptr, 2 bits);
- ⋆ Comparator (no changes) (see Appendix C, fifo_comaprator.v);
- ⋆ Read Address decoder (RdAd, 4 bits) (see Appendix C, fifo_address_decoder);
- ⋆ Write Address decoder (WrAd, 4 bits);
- ⋆ FIFO Memory (4*8 bits) (see Appendix C, fifo_memory.v).

The CRC32 module was implemented using the tables that were shown in theory section (see Appendix C, CRC32.v). A good view of the implementation is shown in figure 6.6.

Once all these components were implemented, they were all merged to create the final_crc.v (see Appendix C).

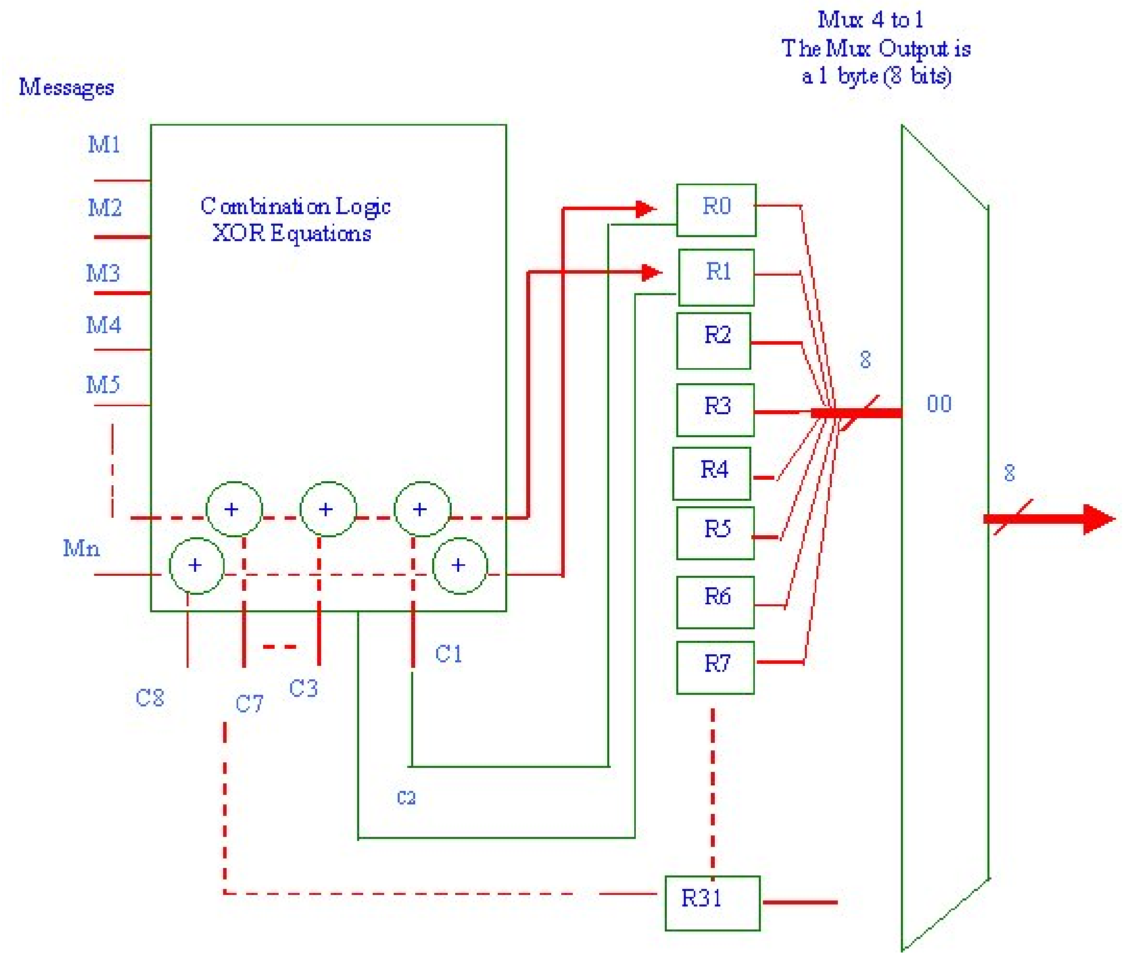

Mux 4 to 1
The Mux Output is a 1 byte (8 bits)
Messages
M1
M2
M3
M4
M5
Mn
Combination Logic XOR Equations
R0
R1
R2
R3
R4
R5
R6
R7
R31
8
00
8
C8
C7
C3
C1
C2

Figure 6.6: CRC-32 overall design

6.4 Functional Simulation

A test bench was written to model the control unit actions and to verify the functionality of the CRC module. (see Appendix C,tb_final_crc.v and CRC functional diagram). At the beginning of the test, we test the flow of the packet, two bytes are written and then read with no problems, this shows that the 4 byte FIFO is functionally working. As said earlier, when *Done_Trailer* is set high for 4 clock cycles by the main control unit (which is our test bench for now), it causes the 2 to 1 input mux to switch up and the 2-bit counter to start counting, this allows the 4 bytes CRC (predetermined divisor) to be placed on the pipeline as you can see on the timing diagram. Once all 4 CRC bytes are out, *Done_crc* goes high. Therefore, the timing diagram has shown that the CRC is functionally correct.

6.5 Synthesis

The CRC module was synthesized successfully (see Appendix C). The output includes all the schematics for the aforementioned modules. As was expected, all the modules are easily recognizable because of the RTL style of coding for most of the internal architecture.

6.6 Timing Simulation

The CRC module was passed through gate level timing simulation, but it failed to produce valid outputs. The final netlist was generated, but the libraries that were called did not match the technology that was installed (and utilized before). This issue was not resolved, and we refer you to section 10.2 for a detailed discussion.

Chapter 7

SAR module design

7.1 Functional Specification

After the design and implementation of the preliminary modules, we finally reach the core module of our project labeled the SAR module. Although we intended to implement a segmenter as well as an assembler, the immense time consumption required for this project limited us to the design and implementation of only a segmenter, yet we will continue to refer to this module as the SAR module.

The main purpose of the SAR module is to partition incoming packets into cells. Referring back to figure 1.4 in the introduction, the SAR, residing in the AAL, receives CPCS PDUs that are tagged with a trailer. The SAR then, upon reception of the PDU, is to segment the variable-length PDU into fixed-length cells, each representing now a 48-byte cell, and send them to the ATM layer to be tagged with the appropriate header and transmitted to the physical interface. This is the high-level description of the SAR functionality, yet we have adopted a new method of performing this operation. We have chosen not to transmit the completed packet (CPCS PDU + trailer) to the SAR module at once and allow the SAR to section off 48-byte cells from the PDU for transmission to the ATM layer, until it has exhausted the entire PDU. This method presented two disadvantages from Proknet's point of view. First, this would impose a strain on the pipeline design that we have chosen for this project. The pipeline is not to be halted at any time for the processing of any type of operation. The above-proposed procedure of the SAR would force the pipeline to delay its operations in the SAR module, hence creating an unnecessary bottleneck for the processing of a PDU of constantly changing length. Second, we have chosen to adopt an 8-bit bus that clocks 8 bits at any one time on the rising edge of our clock. Thus, for example, in order to transfer a 100-byte packet, we require either a faster clock, or a wider data bus able to accommodate a large packet, neither of which is available to us. Thus, we slightly modified the operation of the SAR in order to fulfill our design requirements and conditions. The SAR will receive packets from the packet memory only when there is an entire completed packet residing in memory. When this occurs, the packet will not be transferred to the SAR module in its entirety. Instead, it will be transferred from memory to the SAR one byte at a time, where no processing or delaying of that byte is to occur. The byte will simply pass through the SAR module seamlessly, and onto the ATM layer. All the meanwhile, as soon as the first byte of the packet is passed through the SAR, a counter is initiated to count up to '48', signal the ATM layer that we have just reached 48 bytes, re-initialize to '0' and begin counting again. Once the last cell of the packet has been received, a notification is sent, via a control signal, to the ATM layer informing it of that fact, thus allowing the ATM layer to write a bit in its header indicating that this is the last cell pertaining to the current packet. This is indeed required because at the receiving end, the reassembler needs to know when it has processed the last cell of packet in order to complete the concatenation of the PDU. At this time, the counter will remain at '0' and the packet will not be outputting anything toward the lower layer. The last cell will be detected by performing manual calculations throughout the transfer of the packet through the SAR. When a packet is first received by the TLL, it is allocated a beginning address in packet memory. The SAR module also latches this address. Once the CRC has been completed on this packet, a signal is sent to the control unit, which relays it to the SAR module indicating that a completed packet is now residing in memory. This signal serves two purposes. First, it allows the SAR to begin streaming bytes into its module as well as initiate the counter. Second, it allows the SAR to latch the end address of the packet. By knowing the start address and the end address, the SAR is able to calculate the size of the completed packet. This will aid the SAR in determining when it has received the last cell of the packet in order to inform the ATM layer. We have also designed the SAR module to have a memory capability for the purpose of remembering how many completed packets reside in memory. In the case where there is a completed packet residing in memory while the SAR module is still operating on a current packet, then the SAR will not pause when it comes to service the next packet residing in memory. On the contrary, after it has informed the ATM layer that it has just transmitted the last cell of the packet, the memory capability of the SAR will allow it to continuously stream the bytes of the next packet without any interruption. This was designed for the reason of being able to track the number of completed packets that are residing in memory at any one time.

7.2 Hardware Specification

The implementation of the SAR module components closely resembled the implementation of the counter module components. Mostly, the counter was the main driver in this module to enable the ability to do the up counting. The data and control paths defined for the counter module were replicated within this module, except that our counter consisted of only 8 bits, as opposed to 16 bits for the counter module. Thus, we will not re-examine the definition of these sub-modules in this section. There is one subtle difference between the SAR module and the counter module and that is the memory capability of the SAR, which we will discuss next. As outlined in the functional specification, the memory component was added to the SAR module in order to keep track of how many outstanding completed packets are residing in packet memory. This is extremely important in order for the SAR to continuously stream the bytes of each packet without any interruptions, to keep in-line with the pipeline design as well as repetitively keep track of the start and end of each packet. This will flag the ATM layer about the completed transmission of one complete packet. As mentioned above, once the CRC is done on the current packet, *crc_done* will be generated from the CRC module for the control unit to process. This signal will be transformed into *s_BeginCounting* for the SAR module, which indicates that a completed packet is now residing in packet memory. This signal will used as the clock of our memory flip-flop, which will be driven high all the time and only output its value when its clock, *s_BeginCounting*, which is only a pulse of one clock cycle duration, is asserted. We attach our memory FF to a secondary FF that is clocked to our main clock, in order to make the memory signal a clocked signal. This procedure will allow us to buffer any *s_BeginCounting* signals that are not serviced instantly and be able to keep track of them via a state machine.

The state machine is a Mealy finite state machine that is responsible for generating *s_PacketDone* signal, which, as the name implies, indicates that we have completed the segmentation of a completed packet. The FSM is also responsible of keeping a running tab of how many *s_BeginCounting* signals were received but not serviced. When we designed this function, we intended it to be as close as possible to an interrupt system, complete with

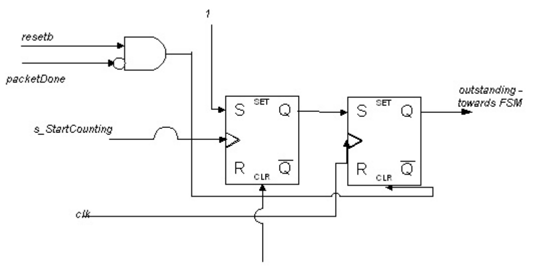


Figure 7.1: Start counting and corresponding flip-flop

its signals and service routines. In short words, the FSM is started when a global reset is asserted. This is when we kick into state0 of the FSM. We will remain in state0 until we receive an *s_BeginCounting* signal. The FSM will change states to state1 while incrementing an internal counter by one, which keeps track of how many of *s_BeginCouting* signals we have received. State1 is only an intermediate state, thus the FSM will naturally move to state2, in the process asserting a signal labeled enable. This is the signal that will act as the 'enable' signal for the entire counter FFs. The FSM will remain in state2 until it either encounters one of two cases. If the FSM receives another *s_BeginCounting* signal while there is one outstanding, then the FSM will switch back to state1, while incrementing the internal counter and keeping *Enable* asserted. If the FSM receives an *s_PacketDone* signal (along with *resetb*, these signals are combinationally combined to provide the reset circuitry for the memory FFs), it will switch to state3 while decrementing the internal counter. In this state, the FSM will check if the internal counter is zero or not. If it is, then the *Enable* signal will be de-asserted and the FSM will return to state0. If the internal counter is not zero, then the FSM will return to state2, while keeping *Enable* asserted. The FSM was designed using behavior modeling, thus we will not delve into its hardware configuration.

7.3 Functional Simulation

Functional simulation for the SAR was executed using the top module test bench (*SAR_top_tb.v*) definition along with the top counter module (*SAR_top.v*). The functional simulation results are shown in the counter section of Appendix F. Functional simulation for the SAR module needed to test the ability of the SAR to output the bytes that it receives from the packet memory seamlessly. This is illustrated by watching as the *OUT_PM*

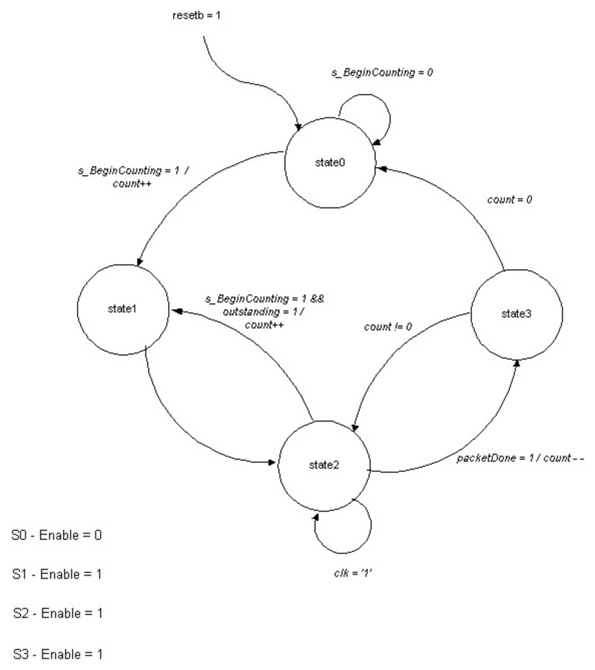


Figure 7.2: SAR FSM

signal follows the *IN_PM* signal exactly. The second test that we needed to prove was the ability to generate the atmdone signal, which indicates that we have counted up to 48 bytes, while handling nested s_StartCounting. The *startC* signal is asserted at 40ns and de-asserted at 80ns. At 40ns as well, an *endAddr* is also latched for calculating the packet's length. Thus, at 40ns the counter begins counting from 0 up to 48, where *atmdone* is asserted. Indeed, as we see at around 2000ns, *atmdone* is asserted and de-asserted at the next clock cycle. Now, the *donePacket* signal is asserted at around 4000ns, which is the amount of time it took to count till 94, constituting 2 complete 48-byte segmented cells (there are some discrepancies in this case due to the way the packet memory was designed). And lastly, we tested the ability of the SAR to handle nested *s_StartCounting* signals. As we can see, at 240ns, *startC* signal is asserted while we still are servicing the previous *startC* signal. As we have designed it, our counter didn't stop from counting when *donePacket* was asserted for the first packet. On the contrary, the length of the second packet was stored in a temporary register, while operating on the first packet. Once it came time to service the second packet at 4,100ns, the counter remained in its incrementing state, of course after it reset back to zero. Once it reached 47 bytes, *atmdone* signal was asserted, shortly followed by *donePacket* signal. These results were enough to convince us that the SAR operated as designed.

7.4 Synthesis

The SAR module culminated in a having a variety of synthesized components as part of its complete structure. The SAR_top module's I/O signal were defined as being the following:

- Input: *clk resetb, in_packet_memory* (bytes from the packet memory towards the SAR), *s_BeginCounting, s_EndAddr, s_StartAddr*

- Output: *c_CellDone, out_packet_memory* (bytes from the SAR module towards the ATM layer), *s_Packet Done, s_atmcounter*

Exploring the top module in a closer manner reveals two sub-modules, the data path and the control path module. The data path module can be seen as being composed of the 'atm counter', which is labeled as T_FF_REG_EIGHT. Since this is only an 8-bit counter, we only require one of the FF modules. As well, we notice the FSM_SAR_ENABLE component, which constitutes the finite state machine for the 'enable' signal. Feeding the FSM are the two memory FFs labeled as D_FF_AR_2 and D_FF_AR_1. The last part of the puzzle is the XOR_Tree_SAR, which is setup in the same manner as the counter module, except that is it triggered upon the detection of the number '48'.

The control path in this module is the same as the one implemented in the counter module. Thus, we can note the two states each using a different type of FF. The reader is invited to revisit the counter module section for more details. The results of the synthesis are located in Appendix F.

7.5 Timing Simulation

As usual, the synthesized files generated the netlist for the verilog code. The newly formulated verilog code was given a secondary test run against the original test bench, and the results were recorded in Appendix F. The gate level simulation closely resembled the functional simulation, except for one glaring difference. If we examine the results, we realize that the same functionality was tested in this case, regarding the generation of *atmdone* and *donePacket*. As well, we attempted at testing the buffering capability of our SAR regarding the *s_BeginCouting* signal. *atmdone* was generated correctly according to the results, as well as being generated correctly when the buffering mechanism was involved. The only signal that did not work is the *donePacket*. We were not successful in generating this signal because of a major error in our implementation code. The code was written in a manner that it passed functional simulation, but we weren't able to generate the net list without modifying the code. This modification caused the *donePacket* signal not be generated

Chapter 8

Microprocessor Module Design

8.1 Theory

In this section, we introduce the final piece to the puzzle. Proknet has, up to now, been defined and outlined in terms of its functional blocks. All these modules can operate autonomously, provided valid inputs, and can execute their intended functions, by supplying temporal correct outputs. To make Proknet a viable IP over ATM segmentation entity, it is required to automate the whole process. By automation, we mean *allowing the machine to execute its purposed functionalities from power up, without the need for user intervention or guidance*. Enters the illustrious "microprocessor". By definition, a microprocessor is "an integrated circuit semiconductor chip that performs the bulk of the processing and controls parts of a system" [1.697]. The bulk of the processing and the system control can be assembled into a common term: the **Central Processing Unit (CPU)**. The CPU's main tasks are the interpretation and execution of instructions. It is traditionally composed of a register unit, an **arithmetic/logic unit (ALU)** and a control unit. Other less-renowned functions of the CPU include address decoding, address and data bus buffer control and memory management. As we can see, the CPU is the central nucleus which manages the information flow within the system. When included on a chip, the CPU is normally labeled as the microprocessor. For this reason, from here on, we will use the terms "microprocessor" and "CPU" interchangeably to represent the same entity [Cle92][Tre87].

There is a myriad of characteristics that a microprocessor can be assessed by, the vital ones being: the widths of its address bus and data bus, its clock rate and its instruction set. A microprocessor can also be classified as being a **reduced instruction set computer (RISC)** or a **complex instruction set computer (CISC)**. Recently, other classes have emerged, such as **very long instruction word (VLIW)**, but do not apply in this scenario. These features are used to describe the microprocessor, and can encapsulate a large amount of information about the operation and utilization of that particular one. Different combinations of these features produce microprocessors tailored for different applications. It is very important to carefully study the design at hand, and incrementally plan the architecture of the microprocessor, knowing that major design changes to this module are very expensive in terms of *man-hours* and *man-neurons*. The former implies that the microprocessor sub-system always resides on the critical path of a project chart, when discussing any processor-oriented or driven implementation, while the latter implies that the microprocessor sub-system exhausts the system designer(s) thinking because of its inherent intertwined complexities and its module inter-dependencies. The design will also involve an open portal of communication between all the other module designers, in order to facilitate the automation process, and not have to spend unnecessary design cycles, attempting to correct a miscommunication-caused erroneous functionality (for example, a module is expecting a signal to be asserted for the duration of an operation, while the microprocessor control unit provides a pulse to that module).

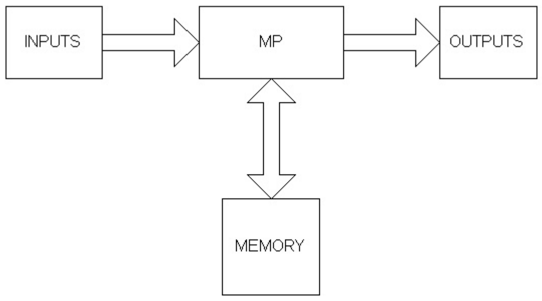


Figure 8.1: Microprocessor general architecture

A typical microprocessor is shown in figure 8.1. It indicates that the CPU interacts with memory and input/output devices to perform a system-wide task. The memory module will be discussed in more detail in section Memory module, but can be introduced as the **random access memory (RAM)** accessible from the system buses. The memory itself in the memory module can be of two major types: RAM and **read-only memory (ROM)**. The former allows the microprocessor to perform read and write operations, while the latter only provisions for *read* operations. The ROM is thus used for holding data that rarely changes, such as the operating systems, interpreters for languages or bootstrap programs. The latter is the first program that runs, after power up, and helps in the initiating sequence of the system. RAM, on the other hand, is used to store the uncompressed operating system, variables used during the execution of a program or in this case, packets ready for segmentation. The input and output modules represent the external peripherals of the microprocessor. These modules are required to give significance to the execution order and output of the microprocessor, as the only way to alter the operation of the microprocessor is through the input module, and subsequently, the only way for the microprocessor to alter another system's operation is through the output module. These will, however, not be delved into more than their considerations as input and output buses. A traditional environment would have had the microprocessor regard these modules as a set of memory cells (an I/O port), that could be written to or read from by simply addressing them, but in our case, an attempt was not done to emulate this tradition as it was not seen to be integral to the correct functionality of the system. Hence, the modules are regarded as buses, that are externally controlled by the corresponding input or output modules.

As described above, one of the main tasks of a classical microprocessor is the decoding and execution of instructions. The instructions are usually stored in memory, for the CPU to fetch, decode and execute. For this matter, registers are used to direct program flow. A register is a high-speed location used to store important information during the execution of CPU operations. One of these registers, the **program counter (PC)** keeps a record of the address of the next instruction to fetch, while the **instruction register (IR)** holds the recently fetched instruction. Figure Execution state machine shows the state machine involved in the execution of program instructions [dB98]. As we can see, the following process is iterated:

Fetch

- The instruction is addressed by the PC;
- The memory is read and its output is placed on a data bus, thus transferring the value to the IR;
- The PC is incremented to point to the next instruction.

Decode

- The instruction is decoded by the control unit.

Execute

- The control signals are generated according to the decoded instruction;
- Go back to the Fetch state.

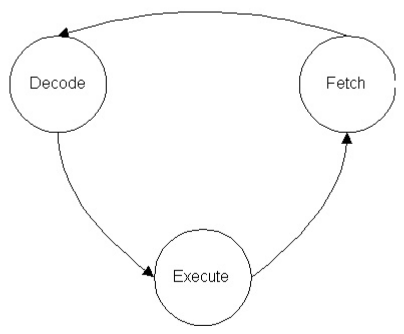


Figure 8.2: Execution state machine

The next section will discuss the design process that was undertaken in order to achieve a working microprocessor module, that performs the bulk of the processing and controls the execution of the IP over ATM segmenter.

8.2 Design history

Initially, the microprocessor module was thought to be one of the few first modules to be designed. Time proved otherwise, as we realized that our microprocessor is a tailored one, in that it provides typical microprocessor functionalities, but also extends to supply network processor functionalities. The initial design started with a look-back towards our pipeline design. The latter included modules, that were later scrapped, such as Queuing and Scheduling and its associated Queue Memory, the Input Buffer Control Unit and the pipeline buffer. Refer to figure 8.2 for a reminder. These modules were taken into consideration when the initial design was laid out. The following brainstorm of instructions was also undertaken:

A - Input Buffer Load from RAM to Input Buffer, and check Empty/Full status of buffer.

B - TLL Start counting and calculate padding; wait for ”End of Packet” from TLL, and interrupt UCPU.

C - Trailer Accept user information for UU & CPI fields (store them in SPRs); start trailing, and send MUX switch signal.

D - CRC Interrupt on CRC_DONE.

E - SAR Wait for data in memory, and segment packet.

F - QAS Move packets from PM to QM, while padding.

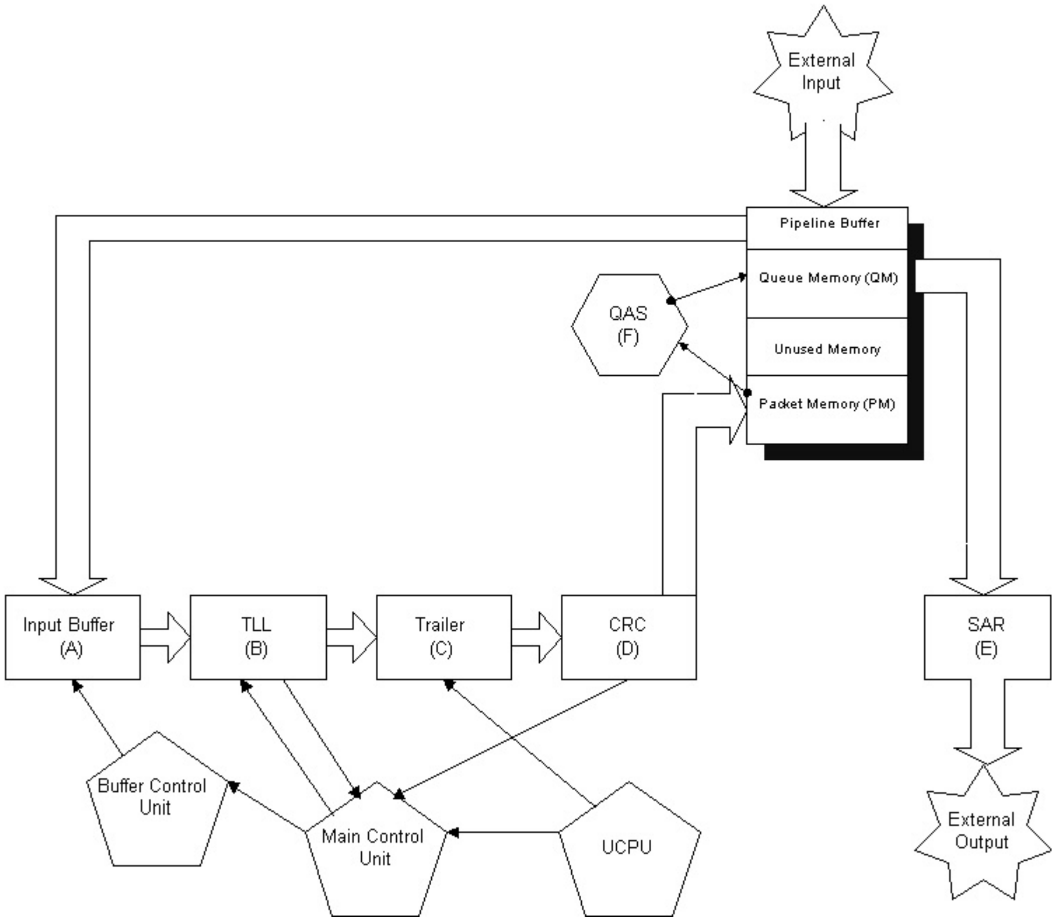


Figure 8.3: Initial pipeline design

The various acronyms will be explained and described later on in this section. The pipeline revisited, the next step is to selectively choose which instructions go into the instruction set. Many instructions were investigated such as: load, store, add, clear carry flag, set carry flag , jump on carry, jump on zero, jump, jump to subroutine, return from subroutine, return from interrupt, subtract, multiply and test. Initially, the frontrunners were load/store (because of QAS operations), add/subtract (because of padding operations), jump/jz (for program control), and jsr/ret/reti (for subroutine calls and interrupt handling). Other instructions were later appended to the list such as ”load input buffer”, ”start pipeline” and ”start counter”. The preliminary bootstrap program resembled the following:

```
0                    clear_flags
1                    load_ibuff
2                    start_pipeline
3                    start_counter
4   pad              jsr pad_count
5   testing          test pack_done
6                    jz testing
7                    jump pad
8
9
10
11  pad_count       save_context
12                   add TLR,8
13                   sub 48
14                   ret
15
16
17  ISR_EOP         load_UU #10000101
18                   load_CPI #00010010
19                   mux_switch
20                   reti
21
22
23  ISR_IBUFF       test ibuff_empty
24                   jnz empty
25                   test ibuff_full
26                   reset write
27  empty            reset read
```

```
                 reti


ISR_CRC         move
                 set pack_done
                 reti
```

The previous program was written with the pipeline design already in mind. Thus, we start by clearing the ALU flags (such as the carry, zero, negative and pack_done flags). Pack_done will be used to indicate the completion of the processing of a packet, and its subsequent placement into memory. In short, the packet is ready for segmentation. The "load_ibuff" instruction was quickly annuled and joined with the "start_pipeline" instruction. The latter commands the input buffer to open its gate and start feeding the pipeline with packet bytes, one a clock cycle. The "start_counter" instruction sends a control signal to the TLL to initiate the counting process. A jump to a subroutine follows, whereas the padding algorithm is implemented. First, the TLR (which contains the total length field, extracted from the IP packet) is added to the eight bytes that make up the trailer, and the latter value is decremented by 48 (the number of bytes to segment each cell to). Once the value reaches 0 or becomes negative, then we know that we have finished the padding algorithm, and the absolute value of that number is the actual number of padding bytes. Please refer to the TLL section for a more in-depth discussion of the padding algorithm. The program above does not completely implement the algorithm, because it was decided that the process should be implemented on the pipeline, and not be controlled through machine instructions. On the return of the subroutine call, we proceed to test for the Pack_done signal. A unity value would indicate that the "ISR_CRC" had executed and set the Pack_done signal, otherwise, the "CRC_DONE" signal has yet to be asserted. The other two interrupts occur when the TLL is done counting the packet, and when the input buffer is either empty or full. In either case, control signals have to be asserted, in order to control the bytes through the pipeline. An interesting feature in the ISR_EOP (the TLL initiated ISR), is the fact that we can control what the user would like to see in the UU and CPI fields of their IP packet. The vision was that these values would be read in from memory and stored into special purpose registers within the microprocessor, and upon signaling, would be transferred over to the pipeline data bus, in order to complete the trailer of the packet. This feature was scrapped due to the replacement of the interrupt mechanism by the main control unit.

The initial microprocessor memory interface is shown in figure Initial memory interface. The design involves the PC and the IR, but introduces two new special purpose registers, mainly the **stack pointer (SP)** and the **address register (AR)**. The normal operation of the circuit involves the PC addressing a value in memory which is brought into the IR, and executed. The PC, meanwhile, gets incremented in order for it to point to the next valid address of the program. The IR can provide the PC with its value, instead of the incremented PC value, and that is because of the jump instruction, which causes the PC to be loaded with the jump address with the latter stored within the jump instruction in the IR. There are a couple of variances to this scenario, and that is the additions of the interrupt mechanism and the pipeline design. The interrupt mechanism requires a register to point to an address in memory where the important system information can be stored in the case of an interrupt occurring. This address is usually stored in the SP. The pipeline design, also introduces a new register. The reason for this is that the CRC module is required to write out a byte of pipeline data every clock cycle. Hence, a mechanism must be arranged in order to keep track of the address that the CRC writes to next. This address is stored in the AR. The external memory has an 8-bit data bus, and a 12-bit address bus, thus, the total available memory is 2^{12} x 8 bytes, which comes out to 4,096 bytes or 4 kB. The memory itself is byte-addressable, in that each address points to a byte of data.

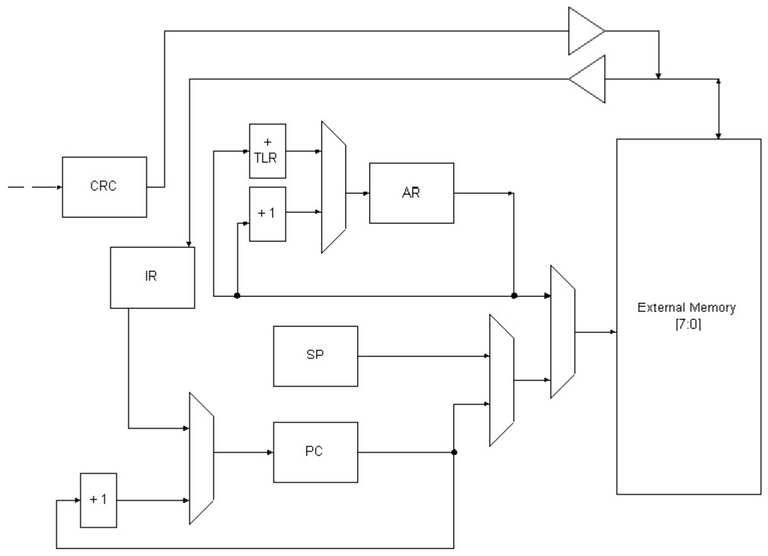


Figure 8.4: Initial memory interface

The initial memory map is shown in figure 8.5. It shows two major decisions that were made. First, the largest IP packet that will be processed in the pipeline will be 100 bytes long, and the input buffer, in the worst case, can only hold 10 maximum-sized IP packets. The memory is further subdivided into a QM and a PM, both of which are 10 kbytes long, and can hold 100 maximum-sized IP packets at any one time. The major difference is that the PM holds them successively, in the order they appear, while the QM holds them in ordered queues (ordered according to priority). The last segment of the memory map is the program memory, which was chosen to be 128 bytes long. The reasoning behind this is that all of the instructions are 8 bits long, and thus in a jump instruction, the largest memory that we can jump to is $2^7 = 128$ bytes away, assuming that the jump instruction itself steals the most significant bit of the instruction, and leaves us with 7 bits to use as an address to jump to (e.g. 10000111 decodes to a jump to memory location 3). The memory map was introduced in this section because of its relevancy to the microprocessor module. A more detailed description of the memory module can be found in the succeeding section.

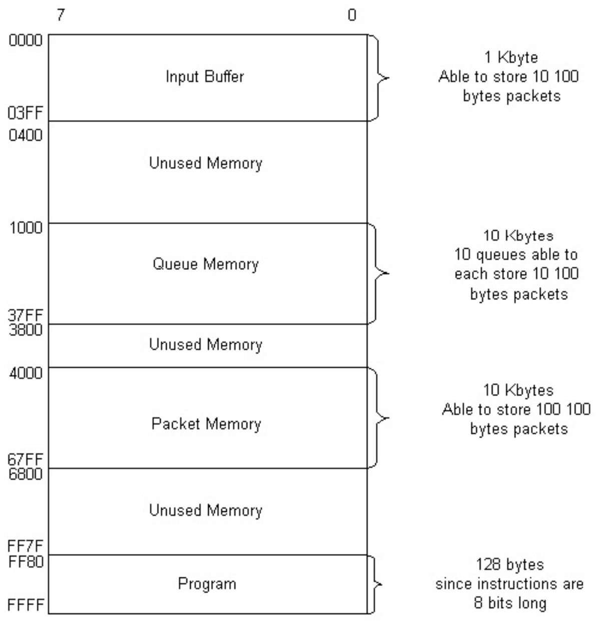


Figure 8.5: Initial memory map

The instruction set was initially encoded according to table 8.1. This table shows that all instructions are 8 bits long, and that the largest memory we can jump was changed to 32, because of the encoding of the jump instruction to to "111a4a3a2a1a0". Other significants are the operations in the execute state of every instruction. Some instructions are typical, thus have common execution steps, but others, such as "start_counter" and "start_pipeline" drive network functionality-specific control signals. One last point to mention about the instruction set is that during the fetch state of all instructions, a byte of memory will be written out to memory, thus in register-transfer notation, the following would hold true 'RAM[AR] $\longleftarrow$ CRC output'.

8.3 Functional Specification

The microprocessor chosen for implementation in this project, is an 8-bit microprocessor. All instruction opcodes are limited to 8 bits. The memory is addressed with a 12-bit address bus, while the data path (according to our pipeline structure) is 8 bits wide. The clock period was originally defined to be 40 ns, thus the clock frequency is 25 Mhz. This microprocessor falls under the RISC family of processors, as its instruction is quite reduced. The final instruction set will be presented in this section. Also hardened will be the memory interface, the bootstrap program, and the control units (yet to be mentioned).

8.3.1 Instruction Set

The final instruction set was reduced to the following instructions: *clear_flags, start_pipeline, start_counter* and *jump*. Their subsequent encoding is shown in table 8.2. Instead of explaining why all the other instructions were taken out, it is more valuable to mention why these particular instructions were kept in. The *clear_flags* instruction allows us to, upon global reset of the system, to clear all the flags that might depict actions taken during the previous operation. The *start_pipeline* instruction allows us to signal the input buffer to start streaming the packet bytes. The *start_counter* instruction allows us to inform the TLL (the first module on the pipeline) to commence its starting sequence, in order to signal the "End of Packet" state. The final instruction, *jump*, is kept because it allows us to loop constantly upon pipeline initialization. Taken out of the design were the subroutine calls, the interrupt mechanism, the addition/subtraction control mechanisms, the mux switch instruction and many more. The reasons vary from the use of a piplined-design, to inherent operations executed within the pipeline (see

Mnemonic	no. bytes	Encoding	Operations
clear_flags	1	00000000	$C \longleftarrow 0; N \longleftarrow 0; p\_d \longleftarrow 0$
start_pipeline	1	00000010	$TM\_rd \longleftarrow 1$
start_counter	1	00000011	$TM\_start\_counting \longleftarrow 1;$
test pack_done	1	00000100	$if p\_d == 1 \longrightarrow Z = 1;$ $else Z = 0$
test empty	1	00000101	$if empty == 1 \longrightarrow Z = 1;$ $else Z = 0$
test full	1	00000110	$if full == 1 \longrightarrow Z = 1;$ $else Z = 0$
add #h	1	0010h3h2h1h0	$ACC \longleftarrow TLR + h;$ $C \longleftarrow carry\_out$
sub #l	1	01l5l4l3l2l1l0	$ACC \longleftarrow ACC - l;$ $C \longleftarrow carry\_out$
jz #a	1	101a4a3a2a1a0	$if z == 1 \longrightarrow PC \longleftarrow a;$ $else PC \longleftarrow PC + 1$
jsr #a	1	110a4a3a2a1a0	$STACK[SP] \longleftarrow PC + 1;$ $SP \longleftarrow SP + 1;$ $PC \longleftarrow a$
jump #a	1	111a4a3a2a1a0	$PC \longleftarrow a$
ret	1	11111110	$SP \longleftarrow SP - 1;$ $PC \longleftarrow STACK[SP]$
reti	1	11111111	$SP \longleftarrow SP - 1;$ $ACC \longleftarrow STACK[SP];$ $SP \longleftarrow SP - 1;$ $PC \longleftarrow STACK[SP]$

Table 8.1: Instruction set encoding

Mnemonic	no. bytes	Encoding	Operations
clear_flags	1	00000000	$C \longleftarrow 0; N \longleftarrow 0; p\_d \longleftarrow 0$
start_pipeline	1	00000010	$TLL\_CU\_start \longleftarrow 1$
start_counter	1	00000011	$tll\_startC \longleftarrow 1;$
jump #a	1	111a4a3a2a1a0	$PC \longleftarrow a$

Table 8.2: Final instruction set

padding), to the desirable simplicity characteristic. The final bootstrap program follows:

```
0                   clear_flags
1                   start_pipeline
2                   start_counter
3   loop            jump loop
```

The program is not very long, but does keep its major functionality: starting the pipeline. One of the many advantages of using a pipeline in our design is shown here, as the bootstrap program (stored in ROM) does not need to constantly control the pipeline. Some inherent control is done by the pipeline itself, along with the aid of a number of control units. The progam also leaves room for future development by keeping the *jump* instruction. Not only does the instruction allow for branching to different parts of the 32 bytes program memory, but it also performs a continuous loop in this case, and hence allows for easy interruption of the program, should the interrupt mechanism deem useful at some time in the future.

8.3.2 Memory Interface

The final memory interface included a dissection of the previously designed interface. As was shown in the final instruction set, the subroutine calls and the interrupt mechanism are no longer needed, hence eliminating with them the need for a stack pointer. One aspect that was modified from previous designs is the break up of the address registers into two: the **write address register (WAR)** and the **read address register (RAR)**. The memory itself will be discussed in the memory module design, but it is suffice to say that the memory is a **ternary-port memory**. Traditionally, when one needs to perform two operations upon the same memory, in one cycle, *dual-port memory* is utilized. In this project, we need to perform *three* distinct operations upon the memory, all in one cycle. The operations are the fetching of an instruction to be executed, the write out to memory a byte of packet information from the CRC module, and in some cases, the read out to the SAR module a byte of packet information from the memory. A solution to attempt to reduce the amount of ports on the memory (which are a disadvantage, as each will require a data bus), is to multiplex at least two of the addressing registers together, in order to alleviate the number of data buses to be interfaced to external memory, but this particular solution was not adopted because of two major reasons: first, the operations are independent and do not have any relationships other than the RAR and WAR addressing the packet memory (queue memory was taken out because of the removal of the QAS module); the second reason for not adopting multiplexed addressing schemes is that a decision was made to design the memory internally, and thus avoid the external interface mechanism. The data buses are thus found within the chip, and do not cost the designers any more than their routing specifications. The overall final memory interface is shown in figure 8.6. The PC is associated with the dual-port address bus, while the IR is associated with the dual-port data bus. The main address bus is connected to the WAR, while the value to write at the WAR's designated address, is received from the CRC module. Finally, the RAR is associated with the ternary-port address bus, while the SAR module is associated with the ternary-port data bus. All data buses are 8 bits wide, while the address buses are 12 bits wide. That said, not all 12 bits are used in address decoding, as will be seen later on.

At this point, we know who addresses what, but we don't know how each does it. There are three main addressing entities: the PC, the WAR and the RAR. Each is vital to the operation of the whole pipeline, with the WAR and RAR playing more eminent roles. The PC is designed according to tradition [dB98], but, as we can see, is not very important in the final design, as it does not address a very large program memory (32 bytes). The PC design is shown in figure 8.7. The operation of this register does not change from the design history section, except that it is worth noting the bus size limit of 5 bits, as the PC only addresses (2^5) 32 bytes of program memory. The rest of the address lines (DPA5-DPA11) are connected to the power supply to cause the program memory to be addressed to the end part of the memory (see figure **??**). The WAR and RAR operate according to figures 8.8 and 8.9. They address the memories for the CRC module to write to, and for the SAR module to read from. The actual transfer of the data is co-ordinated by the address control unit (discussed below). The WAR differs from the RAR, in that the former increments normally, until the *jump_war* control signal

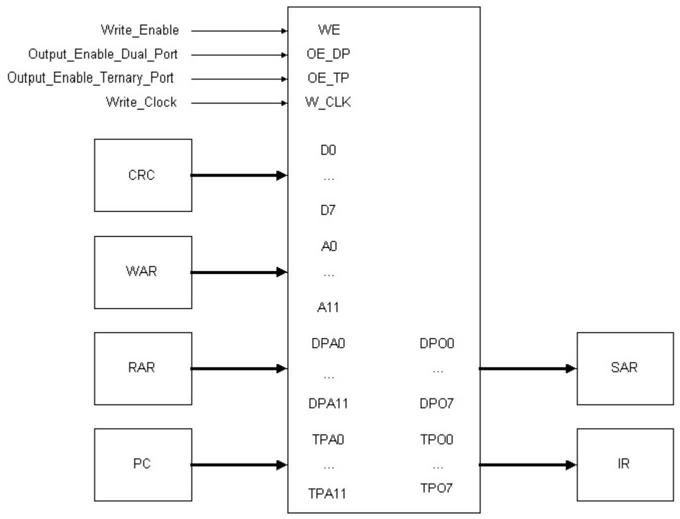


Figure 8.6: Final memory interface

is asserted, whence a switch is made to accept the pad register (PR) value into the adder circuitry. This will perform the padding algorithm, in that, upon command, a certain amount of memory locations will be effectively skipped, and since the packet memory is initialized with zeroes, a certain amount of pad bytes will be inherently appended to the packet. This will work as long as zeroes are re-written into the memory location just read out to the SAR module. Conversely, the RAR only has to increment after each read (and its trigger to start reading), in order to address the bytes of the packet to be read out next. The control signals designated in the previous three diagrams will be discussed in the control units section below.

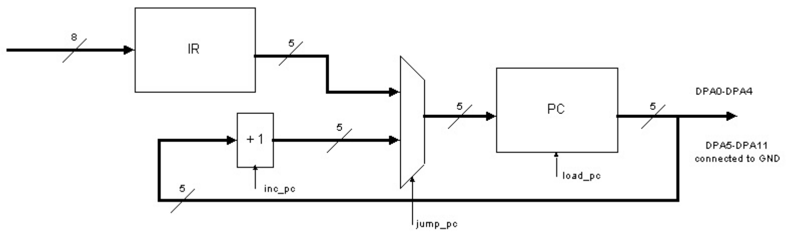


Figure 8.7: PC architecture

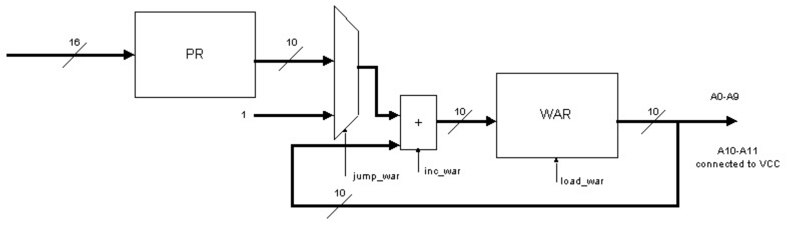


Figure 8.8: WAR architecture

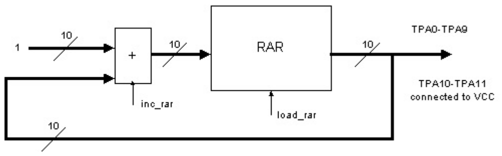


Figure 8.9: RAR architecture

8.3.3 Control Units

Proknet contains a microprocessor which performs all of the previously mentioned tasks. However, the CPU is partitioned into three main units: the **instruction control unit (ICU)**, the **address control unit (ACU)** and the **main control unit (MCU)**. The instruction control unit is the least complex of all three, and will be the first presented.

ICU design

The ICU was partially modified from the original state machine to look like figure 8.10. The assertion of global reset causes the **finite state machine (FSM)** to enter its reset state, where all the machine's outputs are reset. The PC is loaded with its initial value, and the *inc_pc* signal is asserted to indicate the increment of the PC after fetching the first instruction. The *Fetch* state loads the IR (by assertion of *load_ir*) with the data from the memory (the actual instruction to be decoded), and the *Decode_and_Execute* state actually performs the decoding and the execution, as all four instructions do not require more than one clock cycle (no operands to fetch) to execute. Depending on the received instruction, certain control signals are asserted, and then de-asserted. The instructions are all stored in ROM, and hence are fetched on start up. At this moment, it is worth mentioning that the instructions and the instruction control unit do not add or hinder the overall process. The functionality that was supposed to be an advantage was mostly replaced by the introduction of the other two control units, and hence this unit has nearly become obsolete. It is however interesting to note that this unit is functioning, and if need be, can be utilized in future releases. The only utilized function is the *TLL_CU_startC* signal, which is the ICU's signal to the TLL to commence counting (for the very first time). The latter is asserted upon decoding of the *start_counter* instruction.

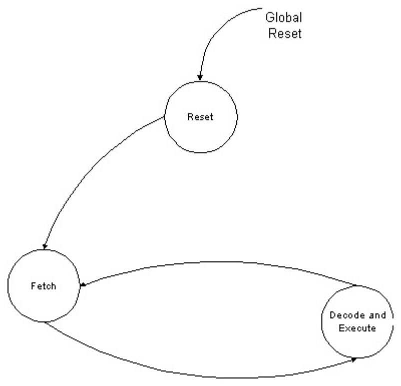


Figure 8.10: ICU FSM

ACU design

The ACU is a more vital control unit, and its corresponding FSM is shown in figure 8.11. As we can see from the figure, the assertion of the global reset puts us directly in the *Reset* state. The assertion of the *TLL_CU_startC* signal from the ICU, will trigger a move to the *CRC_start* state, where the WAR begins to increment (from zero), in order to address valid locations to write bytes out to (assertion of *load_war* and *inc_war* control signals). The FSM transitions to the *Idle* state on the next clock edge (remember that all our FSMs are sequentially designed around our 25 Mhz clock). The FSM remains in the *Idle* state unless one of the following occurs:

1 The TLL module finished counting the number of bytes in the packet (assertion of *TLL_eop*)

- The *jump_war* signal is asserted to execute a jump according the amount of padding bytes we have to append;

2 The CRC module finishes writing a packet out to memory (assertion *CRC_eop*)

- The *load_rar* and *inc_rar* signals are asserted to load the RAR and allowing it to start incrementing to feed the SAR module with its bytes of packet information (the *oe_tp* signal is also asserted to allow ternary-port reads, while the *we* signal is de-asserted to disallow writes from the CRC module to the packet memory);

3 The SAR module completes the segmentation of a packet (assertion of *SAR_pd*)

- The *inc_rar* and *oe_tp* signals are de-asserted to disallow reads from the ternary-port memory;

4 A new packet comes into the pipeline (assertion of *stimulus_startC*)

- The *we* and *inc_war* signals are re-asserted to allow writes from the CRC module again.

As a side note, upon reset none of the control signals are asserted. Also, the *CRC_restart* state was added in order to allow the pipeline to handle more than one packet simultaneously. Finally, all secondary states (the ones following the *Idle* state) are transitions states, in that they *transition* back to the *Idle* state after one clock cycle.

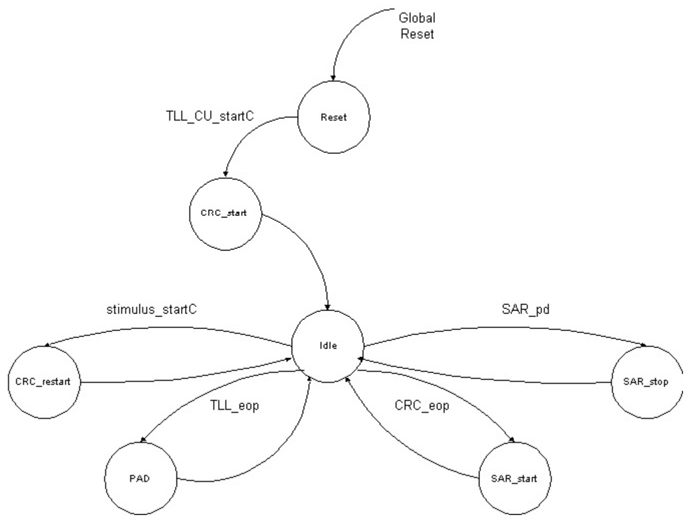


Figure 8.11: ACU FSM

MCU design

The MCU is the nucleus of Proknet. Its corresponding FSM is shown in figure 8.12. Instead of attempting to explain the packed FSM, it would be easier to outline the main functions of the MCU:

1- TLL

- Start the TLL counter through the assertion of *TLL_startC*;
- Handle the *TLL_eop* signal indicating the end of the packet.

2- Trailer

- Drive read and write signals to the trailer FIFO buffer;
- Drive a form of *TLL_eop* to the trailer called *CU_T_eop*;
- Drive a *T_clr_cntr* signal to clear the trailer counter;
- Handle *T_eop* signal indicating the end of the trailing for the packet;
- Handle *Empty* and *Full* signals indicating the status of the FIFO buffer.

3- CRC

- Drive read and write signals to the CRC FIFO buffer;
- Drive a form of *T_eop* to the CRC called *CU_CRC_eop*;
- Handle *CRC_eop* signal indicating the end of the write to memory of the packet;
- Handle *Empty* and *Full* signals indicating the status of the FIFO buffer.

4- SAR

- Drive *SAR_startC* upon reception of *CRC_eop*;
- Drive *Start_Address* and *End_Address* which indicate the start and end of the packet in memory;
- Handle *SAR_pd* signal indicating the completion of the segmentation process for one packet.

The MCU operates on global reset as well, whereas it resets all of its corresponding output signals. The control unit is also provisioned to handle multiple packets on the pipeline as indicated in the FSM. The transition states were carefully selected, after numerous sample scenario runs. For example, a *T_eop* transition can only occur when the machine is in its *TLL* or *Trailer* states, because otherwise the trailer has already been completed, and hence, the *T_eop* signal has already been asserted and de-asserted.

8.4 Hardware Specification

The microprocessor module is implemented using the Verilog [DET95][Smi96] language. Its main components are as described above in the Functional Specification section. The source code can be revised by looking at Appendix D (or in directory /proknet/ucpu, on the accompanying CD). There are three main files: 'inst_cntrl_unit.v', 'address_control_unit.v' and 'main_control_unit.v'. The microprocessor functionalities (PC, IR) are implemented in the ICU design, while the network-processor functionalities (WAR, RAR) are implemented in the ACU. The MCU controls the pipeline, and hence has many other functionalities. All three files contain standard Verilog state machines, with one caveat. The MCU contains a "previous_state" registered value, which aids in the implementation of the FSM. This is necessary because of different transitions within the state machine, hence we need a certain signal to keep reminding us where the previous state, and hence, how we got to be in the present state.

The implementation scheme for this module was chosen to be the behavioural one. Although most of the project was implemented in RTL-style code, it would have been fruitless to attempt to do the same to this module, as this module contained entities that were better-suited for a state machine type of design, and hence, the behavioural representation in Verilog. The registers (PC, IR, WAR and RAR) were also implemented behaviourally (using *always* @ blocks) to follow with the top level structure of this module. Other notables are the fact that addition was also implemented behaviourally. We had at our disposal a fully functional ALU, capable of adding/subtracting two 16 bit numbers, but it was tightly controlled by the pipeline itself, and hence could not have its control shifted towards the MCU without paying a price.

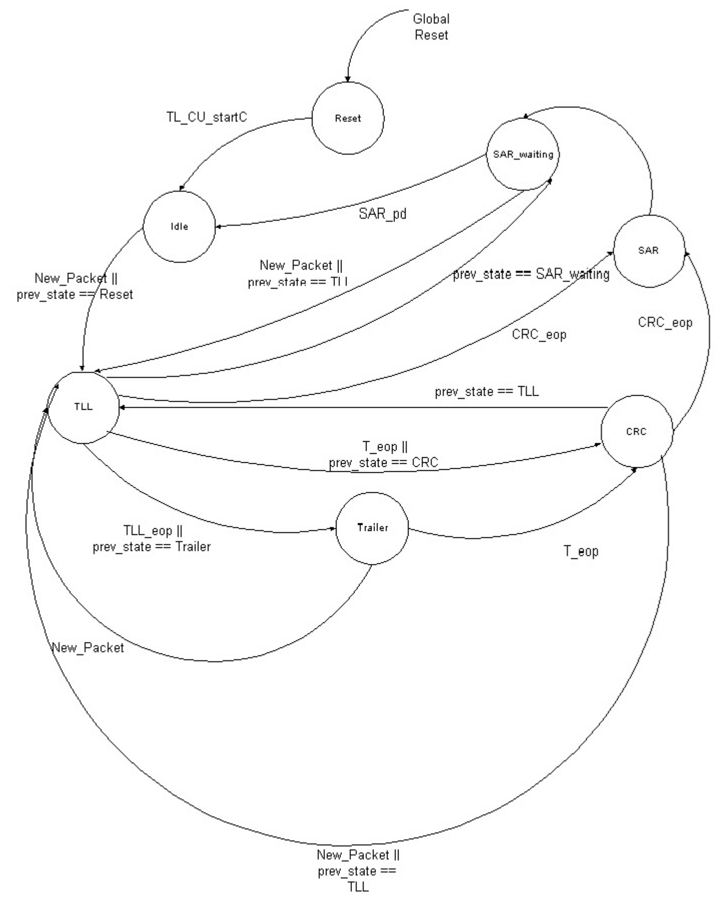

Global Reset
Reset
TL_CU_startC
SAR_waiting
Idle
SAR_pd
SAR
New_Packet || prev_state == Reset
New_Packet || prev_state == TLL
prev_state == SAR_waiting
CRC_eop
CRC_eop
prev_state == TLL
TLL
CRC
T_eop || prev_state == CRC
Trailer
TLL_eop || prev_state == Trailer
T_eop
New_Packet
New_Packet || prev_state == TLL

Figure 8.12: MCU FSM

8.5 Functional Simulation

Functional simulation was run on the microprocessor top level module, and the corresponding results are shown in figure **??**. Functional simulation of this module included the instantiation of the memory module. Now, although the latter has not been formally introduced (following), it has been discussed enough to appreciate its projected objectives. The top level module of the microprocessor is termed *ucpu_top*, while the test bench module is termed *tb_ucpu*. The former is instantiated in the latter, and subsequently tested by the variation of the inputs to the module. As we can see from the functional simulation, the MCU starts out in the *Reset* state (state 0), and upon reception of a *start_again* signal indicating a new packet to be processed, proceeds to the *Idle* (state 1) and then the *TLL* (state 2) state right after. In between, a *tll_startC* signal is sent to the TLL module instructing to commence the counting process. The trailer and CRC module read and write signals also are driven. They actually have the same characteristics as the system clock, since we need to write and read bytes to each of the trailer and CRC module FIFO buffers every clock cycle, in order to fulfill our pipline requirements. The MCU remains in the *TLL* state until it receives feedback from the module that it has completed the counting process (*tll_eop*), upon which the MCU transitions to the *Trailer* (state 3) state, and sends the trailer module a signal (*cu_to_trailer_eop*) to indicate TLL completion, and TLR and PR validity. The machine remains in the *Trailer* state until it receives feedback from the module that it has completed the trailing process (*t_eop*), upon which the MCU transitions to the *CRC* (state 4) state, sends the CRC module a signal (*cu_to_crc_eop*) to indicate trailer completion, and clears the trailer module's counter by asserting *clear_trailer_counter*. The MCU remains in the *CRC* state until it receives feedback from the module that is has completed the processing of the packet, which is safely stored in packet memory, through the assertion of *crc_eop*. The MCU then transitions to the *SAR* (state 5), and subsequently to the *SAR_waiting* (state 6) state, clears the CRC FIFO buffer by resetting both pointers (*clear_read_ptr* and *clear_write_ptr*), and sends a signal to the SAR module (*sar_startC*) indicating to it that there is a packet in memory ready for segmentation. The SAR proceeds in the segmentation process, and upon completion, signals the MCU with the assertion of *sar_pd* (*sar_packetdone*), whence the MCU transitions back to the *Idle* state awaiting a new packet to come into the pipeline (signaled by *start_again*).

8.6 Synthesis

The microprocessor module was synthesized using "Design Analyzer" provided by Cadence. This software allows the designer to input Verilog source files, and generates schematics for each of the modules utilized within the design. Referring to appendix D, we can see the synthesis outputs for all the modules in the microprocessor module. By looking at the top level module, *ucpu_top*, we can see the symbol and schematic diagrams that represent and implement the module, respectively. The symbol diagram presents us with a black-box view of the module, only showing the inputs and outputs of the module, as well as the width of each input or output. The schematic diagram on the other hand presents us with the underlying structure (a white-box view) of the module. First, we can see that the module contains four other module instantiations, which is according to specification. The four internal modules are: *instr_cntrl_unit*, *address_cntrl_unit*, *main_cntrl_unit* and *tp_ram_memory*. The former three will be discussed below, but the latter will be left for the synthesis discussion of the memory module.

The three FSMs discussed earlier control the entire system. The Verilog source code was written behaviourally, and for that matter, it is not simple to recognize the output of the synthesis process. That said, there a few components that should be easily spotted, including all of the state flip-flops, and the output construction logic. For example, looking at the *main_cntrl_unit* schematic, we can trace the *crc_read_signal* output and how it was generated. It is the output of an 2-input AND gate. One of the inputs is the global system clock, while the other is the output of an internal latch. The latch has an internal logic generated input, but also has an enable tied to it. The enable is used to disable reading from the CRC FIFO buffer. This is again according to specification.

8.7 Timing Simulation

This section is yet to be completed. The issue here is that the microprocessor module instantiates the memory module. That said, the latter was written in a brute-force behavioural fashion, and thus has caused the synthesis tool to halt because of a lack of host machine memory. The memory module is quite a large design, including 64 kB of RAM cell components. Included in the memory module are various address decoder instantiations, which do not help the size of the module. The synthesis tool handles the reading of all Verilog source files, and a first-attempt compilation. The synthesis tool orders the designer to "uniquify" all multiply-declared instantiations. Upon the completion of that request, the designer attempts a final compilation step, whereas the synthesis tools utilizes around 1.18 GB of memory, but still manages to *crash*. This issue is a major one, as the designer needs to revert back to the design of the memory, and investigate other options to create the memory. A simple solution would be to interface to external memory, which would actually be the ideal situation. The reason for not performing that in the first place, is that it was thought easier to create a memory that is less than 5 kB in size within the design itself.

Chapter 9

Memory Module Design

9.1 Theory

The last topic of discussion is the memory module design. Memory is a large array of storage cells, each capable of holding one bit of information [Kli82]. Traditionally, the CPU accesses the memory by altering a number of bits at a time, and thus memory is usually byte addressable. The number of bits that could be altered at any one time is termed the *width of the memory*, which changes from computer to computer. An address is associated with each byte of memory, because the latter is not usually accessed in a linear fashion, and thus requires a mechanism of control. The number of addressable bytes in the address space will be a power of 2, calculated by taking the power of 2 to the number of address lines present on the address bus. For example, for a 16 bit address bus, there are 2^{16}, or 65,536 different memory locations that could be written to or read from. The size of the memory is usually described by two numbers: the number of total different memory locations and the width of the memory. Thus, typical values for memory sizes include 2kB x 8 or 16kB x 4 (the unit kB equaling 1024 bytes).

Normal memory is designed with one write enable line and one output enable line, in order to be able to read or write to memory. There is another type of memory that allows for the reading and writing out to memory at the same time called dual-port memory. If on the other hand, two writes were to occur at the same time, some extra logic would have to be built in order to make sure that both writes do not occur on the same memory location. The write enable line is a single wire driven by the microprocessor to control the function of the memory. If the write enable control line is asserted as a logical one, i.e., "true", then the memory performs a write operation. If it is asserted as a logic zero, i.e., "false" then the memory cannot write out to memory. The same logic holds for the output enable line, only that it controls read operations.

The system memory is usually described by a memory map, which serves to section off the different types of memory. The read/write memory, the read-only memory, and the write-once memory (if present) have to be delineated in terms of address space, in order to correctly address them internally in the microprocessor's memory interface. Traditionally, it is advantageous to place the program space at the opposite end of the memory compared to the data or user space. An address decoding mechanism also has to be put into place in order to easily address a memory location by not having to decode all the address lines on the bus. It is worth noting that a **chip_select (CS)** signal is usually used when multiple devices are interfaced to the microprocessor, but in this scenario only the memory is interfaced, requiring one CS signal. Delving further, the designed memory is not external, and hence will actually not even require a CS signal to interface to, however, does require a write enable and output enable(s) in order to correctly read and write to memory locations.

There are numerous types of memories: asynchronous, synchronous, dual-port, first-in-first-out. A brief discussion of each follows. Asynchronous memory allows for new data to be written to it just by changing the address lines, and hence the write enable lines. For this characteristic, a major restriction is put into effect, and that is that the new address line must remain stable for a certain period of time before the actual write enable line changes. Synchronous memory on the other hand, handles these types of situations much more efficiently, as it is quite easier to analyze signal changes for setup and hold-time violations. All signals are synchronized to a global clock, thus are easier to analyze. Dual-port memory has already been discussed. Finally, first-in-first-out (FIFO) buffer memory is used to accept short bursts of high-speed data from the input side, and output them to the receiver at a rate that the latter can handle. FIFO buffers are mostly utilized in communication systems, and have already been discussed in the trailer and CRC module design sections [dB98].

9.2 Design History

The memory was one of the last modules to be designed and implemented. For this matter, it was already outlined in terms of functionality, inputs and outputs as presented in the microprocessor module design section. As can be seen in figure 8.5, the input buffer was thought to be contained within the system memory. A buffer control unit would be controlled by the MCU, and thus the pipeline would be fed with the bytes of the packets accordingly. The initial design for the memory was for a single-port synchronous memory, as shown in figure Initial memory interface. The design included one input/output bus that would allow for read operations (into the IR), and write operations (from the CRC module). This solution quickly became disbanded for many reasons. First, as mentioned before, we will need to perform three different operations simultaneously. One issue to mention is that we will not have to introduce any external circuitry to disallow two writes to the same memory location, because the three operations are one write, and two read operations (Two reads to the same memory location are allowed). For this matter, the implemented memory is a **ternary-port memory (TPRAM)**. It is, as mentioned, synchronized to the global 25 Mhz clock, and hence, the memory location values can only be altered at the edge of the global clock, and not at the edge of the write enable lines (as in asynchronous memories).

The memory itself is a 4kB large, and its size is represented by the following notation: 4kB x 8, meaning that there are 4,096 total addressable memory locations with a memory width of 8 bits. The total valid memory locations came about by taking the power of 2 to the number of address lines, the latter being 12. Thus, 2^{12} equates to 4,096 memory locations. The address bus width was chosen to be 12 bits, because packet memory is defined to be 1 kB large, and the program space is defined to be 32 bytes long. Hence, for further expansion, a sensible amount of unused memory was placed in, and thus the 12 bit addressing scheme was selected. The memory has one write enable line (*w_en*) to allow for the writes from the CRC module, and two output enable lines, one to enable reads to the SAR module (*tp_rd*), and the other to enable reads to the IR (*dp_rd*). Each of these signals are actually buses that should be 4,096 bits wide, but because of address decoding are less than that. This point will be delved into in the next section.

The need memory being only a few kilobytes large, it was unnecessary to interface to external memory, and thus it was chosen to implement the memory internally. That said, an initial design of a RAM cell is shown in figure 9.1. This RAM cell is only used to instantiate two of the ports of the TPRAM. The third port is actually the dual-port which feeds the IR, and is presented in the ROM design section. The RAM cell will be the basis used to make all of the system memory, and will hold all of the packet information. It will be the lowest entity that will be visible in terms of system memory.

An address decoding mechanism will have to be used to address the different types of memory, but its discussion will also be left for the next section, as no major design iterations were spent on it.

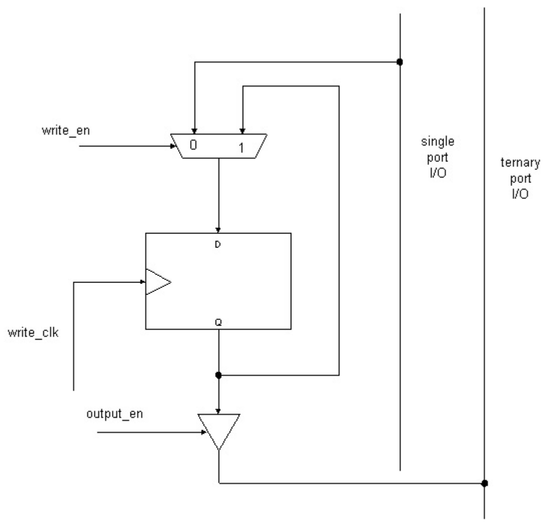


Figure 9.1: RAM cell

9.3 Functional Specification

The new memory map is shown in figure 9.2. The only addressable spaces are the packet memory and program spaces. The program space is actually a 32 byte ROM implementation, while the packet memory is a 1 kB RAM implementation. The latter, as previously mentioned, took its shape from the basic RAM cell. From the small building blocks (RAM cells), a whole memory system was generated. More on this topic in the hardware specification that follows.

Another major block of the memory module is the address decoding scheme. There were two major types: the microprocessor address decoding scheme, and the network processor address decoding scheme. The former is illustrated in figure 9.3. It can be shown that from the address DPA5-DPA0, and the oe_dp enable line, the 32 output enable lines can be generated, so as to read the correct memory location. The network processor address decoding scheme is much more complex and will not be shown here, as it is just an extension of the ideologies presented in the MCPU address decoding scheme.

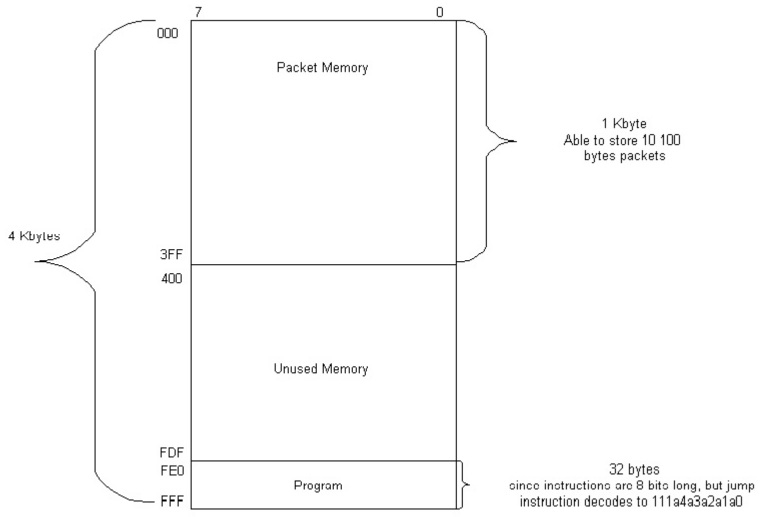


Figure 9.2: Final memory map

One last point is the input buffer implementation. It was decided, that since the input buffer usually feeds the IP over ATM segmenter, it would be logical for it to be the output of a buffer or another chip sitting on the board. That said, it would be quite unrealistic to store the buffer in the main memory as depicted in the initial memory map. or this reason, the input buffer will be moved from the main memory, and into the final test bench of the top level module. A brief period of time was spent in order to attempt to place the input buffer into a file, which could be read into the Verilog simulation environment, but was not implemented due to time constraints.

9.4 Hardware Specification

In order to generate the system memory, eight RAM cells were instantiated to make a byte, then eight bytes were instantiated to make 8 bytes, eight of which were instantiated to make 64 bytes, eight of which were instantiated to make 512 bytes. Now, the memory space was modified during the address decoding, since the WAR and RAR only used the least significant 10 bits to address their common space, the packet memory, while the PC only used the least significant 5 bits to address its space, the program space. The PC was picked as the dual port, while the WAR was the single port, and the

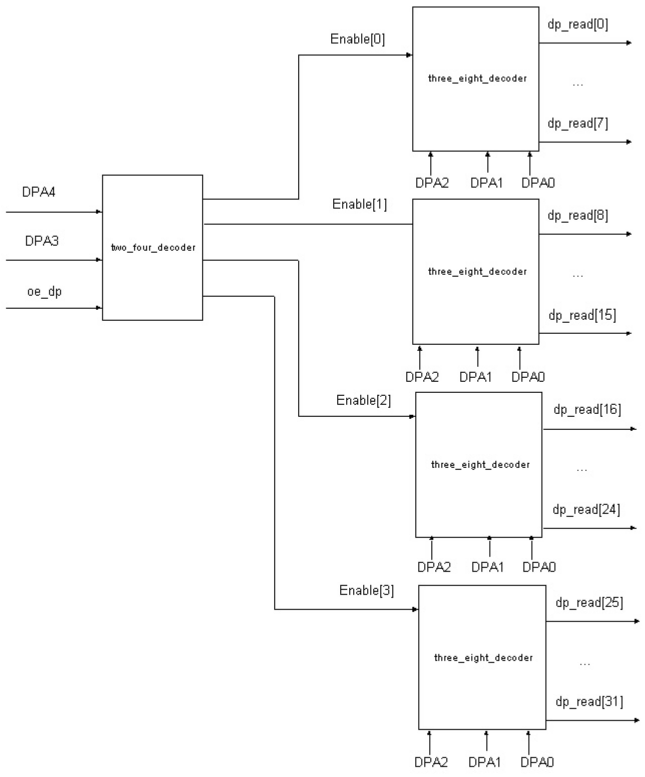


Figure 9.3: Micro-CPU address decoding

RAR was the ternary port. That said, we come to an earlier point that there less than 4,096 w_en/dp_rd/tp_rd lines. This holds true because of the address decoding simplifications. There are only 32 dp_rd lines, as there are only 32 bytes of ROM to read from, while there are 1,024 w_en/tp_rd lines to encompas the 1 kB of packet memory. One major design change is the reduction of the address space to 2 kB, as indicated by the 10 bit address decoding of the WAR and RAR. That said, to instantiate 2 kB of memory (32 bytes of which is ROM, and the rest RAM), we included three 512 byte RAM instantiations, seven 64 byte RAM instantiations, one 32 byte RAM instantiation, and of course, one 32 byte ROM instantiation (3*512 + 7*64 + 2*32 = 2,048).

The ROM block was written in behavioural code. Its mode of operation is quite simple: when queried with an address, decode it and drive the correct dp_rd line, whereas the data bus connected to the IR, will be filled with the ROM byte at that memory location (see accompanying source code, or CD directory /proknet/memory/rom_32bytes.v).

The address decoding block was implemented according to the truth table implementations of each decoder. There are three decoder blocks: the two-to-four decoder, the three-to-eight decoder and the four-to-sixteen decoder. All three are utilized in the decoding scheme, and their instantiations and interconnections can be seen by looking at the source code (see accompanying source code, or CD directory /proknet/memory/address_decoder.v).

9.5 Functional Simulation

Functional simulation was run on the memory top level module, and the corresponding results are shown in figure Memory functional simulation. The top level module of the memory is termed *tp_ram_memory*, while the test bench module is termed *tb_memory*. The former is instantiated in the latter, and subsequently tested by the variation of the inputs to the module. As we can see from the functional simulation, the outputs match our desired results. Upon global reset, our ROM begins to output bytes of data to the *IR*, whence the *oe_dp* output enable line to the dual port is enabled. The ROM will output one instruction to the IR every clock cycle. These instructions were initialized in the ROM source file. We can also see that the PC is incrementing every clock cycle to address the next memory location. Another observation to make is that the CRC module writes two bytes out to memory, the first with a value of 0x12 to address 0x008, while the second is 0x0F to address 0x004. The *write_en* signal is asserted each time a write is initiated. On the other side, the *oe_tp* output enable signal is asserted once the SAR module is ready for its data, which happens to be 0x12 and 0x0F, stored at addresses 0x008 and 0x004, consecutively. This demonstrates the correct functionality of the TPRAM.

9.6 Synthesis

The memory module was synthesized using "Design Analyzer" provided by Cadence. This software allows the designer to input Verilog source files, and generates schematics for each of the modules utilized within the design. Referring to appendix E, we can see the synthesis outputs for all the modules in the memory module. By looking at the top level module, *tp_ram_memory*, we can see the symbol and schematic diagrams that represent and implement the module, respectively. The symbol diagram presents us with a black-box view of the module, only showing the inputs and outputs of the module, as well as the width of each input or output. The schematic diagram on the other hand presents us with the underlying structure (a white-box view) of the module. First, we can see that the module contains two other module instantiations, which is according to specification. The two internal modules are: *address_decoder* and *ram_2kbytez*.

The *address_decoder* module contains numerous instantiations of the three main address decoding modules: *two_four_decoder*, *three_eight_decoder* and *four_sixteen_decoder*. Please refer to section 9.2 for a more detailed discussion of the design history of this module. The interesting note to observe when referring to the address decoding synthesis outputs is that only NOT and AND gates are utilized to implement the modules. This was expected as the modules' Verilog code only mapped out the truth table of each decoder into source code. The truth tables are easily implemented using NOT and AND gates, which was actually the truth.

The *ram_2kbytez* module contains numerous instantiations of the smaller memory modules: *ram_512bytez*, *ram_64bytez*, *ram_32bytez*, *ram_8bytez*, *ram_byte_z*, *ram_cellz* and *rom_32bytez*. The basic module is the *ram_cellz*, which is implemented using a flip-flop and other combinational logic. The main specifications of the memory can be reviewed in section 9.3.

9.7 Timing Simulation

Please refer to section 8.7 for a discussion of this task.

Part IV

Conclusions

Chapter 10

System Integration

10.1 Proknet©in all its might

10.1.1 System Integration

This part of the project was divided into three parts:

- TLL and Trailer Integration. (see Appendix G, tll_trailer_top.v)
- TLL, Trailer and CRC integration. (see Appendix G, tll_trailer_crc_top.v)
- TLL, Trailer, CRC, SAR and Micro CPU and Memory integration which is know as Proknet. (see Appendix G, final_top.v)

The reason for this is to assure that the modules are carefully studied and that their behaviors well understood; that way it becomes easier to debug as the integration progresses.

10.1.2 Functional and Hardware Specification

As far as functional and hardware specifications go, nothing was changed in the designed modules of the pipeline. In each integration phase, the appropriate modules were instantiated, and valid inputs and outputs were passed as was designed in the pipeline.

10.1.3 Functional Simulation

For TLL and trailer integration, a test bench (see Appendix G, tll_trailer_top_tb.v) was written in order to model the control unit's actions (see Appendix G), and to test their functionality together (see Appendix G for timing diagrams). In the first timing diagram (Tll_Trailer_Integration(1/2)), a packet was received on the pipeline(In_data_bus) and *StartC* (start counting) was asserted to allow the TLL module to extract the length of the packet and to compute the padding bytes. As shown, the bytes of the packet were put out to the pipeline (out_data_bus), a clock cycle later from when it was received. This cycle keeps going until the TLL module sets the *eop* signal high, which causes the control unit to set *cu_eop* high for 8 clock cycles, which allows the trailer module to output the trailer fields (see Appendix G, Tll_Trailer_Integration (2/2)).

Knowing that the TLL and trailer modules functioned properly, the CRC module was merged and integrated along with (see Appendix G, tll_trailer_crc.v). A test bench was written to test the functionality of all three modules (see Appendix G, tll_trailer_crc_top_tb.v). The main concern in this part was to see if the CRC module will output the 4 bytes of CRC at the given signal *cu_c_eop* (it is set by the control unit when signal *doneT* is received from the trailer module), this signal will stay high for 4 clock cycles.

Having done so many integrations and feeling very comfortable with the outputs and timing diagrams, it was time to pick up the speed and merge the rest of the modules (SAR, MicroCPU and Memory). Please refer to Proknet Top Level Functional Simulation 1,2 and 3 as the discussion progresses. A packet of 16 bytes was put on the pipeline and the signal *start_again* is asserted to indicate the arrival of a new packet. The packet goes through the TLL module and the following occurs:

- Extractor extracted the TL (total length of the packet) and stored in TLR (16).
- Pad Unit computed the number of pad bits needed and displayed by PR (24).
- Packet kept flowing through the TLL, trailer (because *CU_T_eop* is low), CRC (because *CU_CRC_EOP* is low) until the *TLL_eop* signal is set high.

Just a reminder that 16 (length of packet) + 24 (output of pad algorithm) bytes of padding + 8 bytes of Trailer = 48 bytes. Therefore the packet is divisible by 48.

When *TLL_eop* was set high by the TLL module, the Control Unit set *CU_T_eop* high for eight clock cycles in the Trailer Module which in return:

- Switches the 2 to 1 mux and triggers the 3-bit counter to start counting;
- The 3-bit counter is the selector of the 8 bytes trailer, therefore as it counts, it is outputting the trailer fields to the pipeline. When the 3-bit counter is done counting, it sends a *T_eop* back to the MCU.

When *T_eop* is received by the MCU, *CU_CRC_EOP* is asserted for four clock cycles which causes the following to occur in the CRC module:

- Switches the 2 to 1 mux and triggers the 2-bit counter to start counting;
- The 2-bit counter is the selector to the 4 bytes of valid CRC that were computed by the CRC-32 unit, so as the 2-bit counter counts, it is selecting the CRC bytes and putting them out onto the pipeline. (Please see *crc_data_out*, notice the Trailer and CRC). When the 2-bit Counter is done counting it sends *CRC_eop* (it means CRC has been appended) to the MCU.

Upon receiving the *CRC_eop*, the main control unit sends a *SAR_startC* signal to the SAR module, which causes the following to occur:

- Counter is triggered (see *atmcounter*), and the counter keeps counting up to 48 then;
- *Cell_Done* is asserted, meaning that 1 48 byte cell has been sent.
- *Packet_done* goes high which means that 1 IP packet has been processed.

10.1.4 Synthesis

Synthesis was previously separately done on each module (please refer to the individual modules for more information).

10.2 Proknet©limitations

The following section is brief liaison of all the major hurdles that have been left behind because of a major lack of resources, be it time, man-power or development tools. The major task to be handled is the completion of the padding algorithm. The latter requires a module to write back zeroes into the just-read bytes of memory. The original thought was that the SAR would accomplish this task, but it turned out too memory intensive, as a record of the start and end addresses of the *previous* packet have to be kept. The other module that could have taken care of this is the micro-processor module, but the latter already was interfaced to a TPRAM, and was not able to handle more ports, or multiplex the data on one of the existing ports.

A major limitation of Proknet is that it cannot handle more than 2 packets on the pipeline. It can on the other hand handle 1 packet (the size being smaller or greater than 48 bytes), and 2 packets. The limitation is that when more than 2 packets are present on the pipeline, the TLR and PR registers, as well as the Start_Address and End_Address lines all latch or store values of the next packet, while the previous one is being processed by the SAR module.

An ATM specification change was made during the design, and that is to only perform the byte-wise CRC calculations on the packet and the trailer, excluding the padding bytes. The reason for that is simplicity of design. In order to make up for this specification change, an assumption about the reassembly side was made: the reassembler will strip the padding bytes first, and then calculate its own CRC to see if it matches the packet's sent CRC information (4 bytes of it).

As previously mentioned, a queuing and scheduling module was supposed to add extra functionality to Proknet, but it was scrapped as soon as the priorities were handed down by the members of the group. Queuing and scheduling was seen as a luxury to have, rather than a necessity for the correct operation of the system.

As aforementioned as well, the timing simulation phase of the project did not succeed as anticipated. Numerous problems were faced from design methodologies, to library problems to memory constraints. Please see each module design section for more detail on this topic.

Finally, by looking at the final top level functional simulation, we can see that there are two bytes that are not propagated to the output of the pipeline. The reason for that is found in a design error. The FIFO buffers were designed to hold exactly the same amount of bytes as each module (trailer and CRC) was transmitting. For this reason, the buffer would get full, and then would have to be emptied by one byte first, before allowing a write operation on the buffer. For this reason, a byte is lost in each buffer, and is not saved, but actually overwritten in the buffer. A resolution to this problem would be to increase each buffer's size by one, making the trailer FIFO buffer a 9-byte buffer, and the CRC FIFO buffer a 5-byte buffer.

10.3 Future Work

Proknet has managed to successfully segment variable-sized IP packets into fixed-sized ATM cells. That basic functionality was implemented and functioned correctly. What is missing are all the extra caveats that were thought up in the preliminary design phase of the project. Queuing and scheduling is a great luxury to have, and would have to be the first module implemented if Proknet 1.0 were to come out. Other major improvements would be a greater pipeline throughput, a correct (according to specification) CRC calculation, and a correct implementation of the padding algorithm. Reviewing the current limitations of Proknet, it would be a great improvement to successfully simulate the timing characteristics, and place and route the design. Proknet would then become an entity ready to be implemented on any programmable chip.

Proknet could also benefit from a parallel architecture, where multiple pipelines could be executing on various packets. This system would have to be a massively-parallel system, but would increase the throughput by leaps and bounds. This idea could be beneficial in the re-engineering of today's SARs on the market, as they operate on a queuing and scheduling algorithm, but do not include multiple pipelines.

10.4 Project Conclusions

This project started off as a migration from traditional microprocessor design to a more innovative network processor design. The idea was a raw one, with only a few references to go by. The design work accomplished proved that this technology is of great benefit to the networking world. Today's major (communication) industry players use SARs as commonly as they use resistors on their printed circuit boards. Proknet was born in a small design room in McDonald Hall at the University of Ottawa, but its birth gave true design experience to its parents. It has also come a long way from the block diagrams that were "supposed" to segment a packet into smaller chunks, and has flourished into a working Verilog model, as well as a synthesized netlist.

To make the really long story long, the project's main objectives have been met. The designers believe that they have acquired a great deal of knowledge by doing everyday designer tasks. One point to be stressed is the fact that the implementation phase should have been guarded against, as the designers did spend quite a long time in the design phase. As a result though, a straightforward implementation eased the flow into synthesis.

Proknet is alive and well. It is awaiting its total completion, but stands before you today, as a valid IP over ATM segmentation entity.

Bibliography

[1.697] Wordnet 1.6. Org from ftp://clarity.princeton.edu/pub/wordnet/wn1.6unix.tar.gz. Technical report, Princeton University, 1997.

[Cle92] Alan Clements. *Microprocessor Systems Design: 68000 Hardware, Software, and Interfacing.* PWS Publishing Company, 20 Park Plaza, Boston, Massachusetts 02116, 2nd edition, 1992.

[dB98] Dave Van den Bout. *The practical Xilinx Designer Lab Book.* Xilinx Inc., 1998.

[DET95] Philip R. Moorby David E. Thomas. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, Reading, Massachussetts, 1995.

[Hal92] Fred Halasall. *Data Communications, Computer Networks and Open Systems.* Addison-Wesley Publishers Limited, 1992.

[Ibe97] Oliver C. Ibe. *Essentials of ATM Networks and Services.* Addison Wesley Longman Inc., 1997.

[Kli82] Edwin E. Klingman. *Microprocessor Systems Design, Microcoding, Array Logic, and Architectural Design.* Prentice Hall Inc., 1982.

[Lee00] Sunggu Lee. *Design of Computers and Other Complex Digital Devices.* Prentice Hall, 2000.

[Pry95] Martin De Prycker. *Asynchronous Transfer Mode Solution for Broadband ISDN.* Prentice Hall International (UK) Limited, 1995.

[Smi96] Douglas J. Smith. *HDL Chip Design.* Doone Publications, 1996.

[Tan96] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall PTR, 1996.

[Tre87] Nick Tredennick. *Microprocessor Logic Design.* Digital Equipment Corporation, 1987.

Appendix A

A - TLL module

Appendix B

B - Trailer module

Appendix C

C - CRC module

Appendix D

D - Microprocessor module

Appendix E

E - Memory module

Appendix F

F - SAR module

Appendix G

G - System Integration

Appendix H

H - Design Documents

Appendix I

I - Project Presentation