NTNU – Trondheim
Norwegian University of
Science and Technology

Programmable Microcontroller Peripherals

Anders Ruden

Master of Science in Electronics
Submission date: June 2013
Supervisor: Bjørn B. Larsen, IET
Co-supervisor: Tor Erik Leistad, Atmel Norway AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Statement

Microcontroller hardware peripherals implement fixed, specific functions or protocols which cannot be changed once silicon is made. A typical microcontroller will have several peripherals multiplexed on the same pins, leaving some of the microcontroller hardware unused in the end application.

A study is proposed to investigate if peripherals can be substituted by embedded programmable CPUs, which emulate features normally supported by a hardware module, such as SPI, UART, TWI, or USB. The study should propose a specific solution and implement this in HDL. One or more of the protocols should be implemented to demonstrate the selected solution. The resulting area, power consumption, and performance should be compared to a traditional hardware solution.

Acknowledgements

I would like to acknowledge the people that have helped me to complete this master thesis.

First, I would like to thank my supervisor at Atmel, Tor Erik Leistad. He have been extremely helpful by explaining all my questions in the smallest detail, in a manner that is easily understandable. He have also given me good feedback on my work and guided me when i have been stuck. Without him it would have been impossible for me to finish this thesis.

In addition, I would like to thank my supervisor at NTNU, Professor Bjørn B Larsen for the help and feedback he have given me throughout this school year on both the preliminary work and the master thesis.

Last, I will thank my family and friends who have been supporting me throughout the years of my master's studies.

Abstract

This thesis is a continuation of work done in a specialization project. The result from the preliminary work have been used to implement a programmable peripheral processor in HDL that can replace non-programmable hardware modules. The implemented solution have then been tested to find out if it is capable of doing the most basic operations that a UART protocol require to do parallel-to-serial and serial-to-parallel conversions. The results of the implementation and testing have been analysed and the performance, area and power consumption have been presented. The resulting performance and area have also been compared to traditional hardware solutions.

The results from the tests demonstrates that the presented peripheral processor is capable of doing the basic operations that is required to do parallel-to-serial and serial-to-parallel conversions. However, the area of the peripheral processor is significantly larger than the total area of multiple non-programmable hardware modules. The result of this is that the cost of utilizing a peripheral processor will be greater than with existing solutions.

Sammendrag

Denne oppgaven er en fortsettelse på arbeidet gjort i et fordypningsprosjekt. Resultatet fra forarbeidet har blitt brukt til å implementere en proggrammerbar perifer prosessor i HDL som kan erstatte ikke-programmerbar maskinvare. Den implementerte løsningen har blitt testet for å finne ut om den er i stand til å gjøre de mest elementære operasjonene som behøves for å gjøre seriell-til-parallell og parallell-til-seriell datakonvertering på samme måte som en UART protokoll. Resultatet av implementasjonen og testingen har deretter blitt analysert og ytelsen, størrelsen og strømforbruket til prosessoren har blitt presentert. Ytelsen og strømforbruket har også blitt sammenlignet med tradisjonelle ikke-programmerbare løsninger.

Resultatene fra testene viser at den gjeldene prosessoren er i stand til å gjøre de mest elementære operasjonene som kreves for seriell-til-parallell of parallell-til-seriell convertering. Imidlertid, viser resultatene også at prosessoren er betydelig større enn størrelsen til flere ikke-programmerbare maskinvare moduler sammenlagt. Dette fører til at også kostnaden til den perifere prosessoren vil bli betydelig større en eksisterende løsninger.

Contents

List of Figures

List of Tables

Abbreviations

AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASB	Advanced System Bus
ATPG	Automatic Test Pattern Generator
CPU	Central Processing Unit
DMA	Direct Memory Access
FIFO	First In First Out
HDL	Hardware Description Language
IRQ	Interrupt ReQuest
LCR	Line Control Register
LIFO	Last In First Out
LSB	Least Significant Bit
LSR	Line Status Register
MSB	Most Significant Bit
NAND2	Not AND gate with 2 inputs
PPU	Peripheral Processing Unit
RTL	Register Transfer Level
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TWI	Two Wire Interface
UART	Universal Asynchronous Receiver and Transmitter
USART	Universal Synchronous and Asynchronous Receiver and Transmitter
USB	Universal Serial Bus

Chapter 1

Introduction

This thesis is a continuation of the preliminary work done in a specialization project the autumn 2012. Some parts of this thesis will therefore include the same content as given in the report for the preliminary work. It will be informed when the content is from the preliminary work. The motivation is the same for this thesis and the preliminary work [4].

1.1 Motivation

Data is often needed to be transferred between a microcontroller and other devices. A common way to do this is by using a data transfer protocol that do a parallel to serial conversion and then sends the data serially between the external device and the microcontroller. There are several different protocols that support this type of parallel to serial conversion. USART, SPI, TWI and USB are four common protocols. Each protocol have separate advantages and limitations and is therefore not suitable in the same situations.

A microcontroller have to support several different transfer protocols. Today this is done on Atmel microcontrollers by having different hardware modules to control the various data transfer protocols [2]. The disadvantage of this is that the microcontroller is locked to only support the predefined data transfer protocols because these modules

are not programmable. If a user want to do data transfer with a protocol that is not supported, the user have to program the CPU to control the data transfer. This is complicated and uses a lot of CPU resources which could have been used for other tasks, and is therefore not desirable. An alternative solution can therefore be to implement a programmable peripheral processor to control the data transfer instead of non-programmable hardware.

1.2 Approach

Based on the problem description and the result of the preliminary work, the following bullet points summarizes the work that is done in this thesis:

- **A peripheral processor is implemented in HDL.** The results from the preliminary work are used as a specification to implement a peripheral processor in HDL.
- **Simulations are run on the Implemented Solution.** A test program that can simulate the behaviour of a peripheral protocol is created.
- **The Implemented Solution is analysed.** The results are analysed to present the performance, area and power consumption of the peripheral processor.
- **The implemented Solution is compared to Existing Modules.** The performance, area and power consumption of the implemented solution are compared to existing hardware modules.

1.3 Outline

The report presents first the theory required to understand the content of the thesis. Next a summary of the preliminary work is presented in chapter 3. The summary includes the most important results needed to give a specification of the peripheral processor. Furthermore, chapter 4 gives a detailed description of the different steps in

the work towards the presented solution. The results of the implemented peripheral processor is presented in chapter 5. Chapter 6 describes the functionality and explains the reasons for the choices made. Last, the conclusions is given in chapter 7.

Chapter 2

Theory

The theory in chapter 2.1, 2.2, 2.3 and 2.4 is the same as in the preliminary work. Chapter 2.5 summarizes the AMBA APB bus specification.

2.1 Hardware Versus Processor

There are two different methods to do data processing. One is by using a processor and the other is by using non programmable hardware modules. Non programmable hardware modules consists of logic gates such as AND, OR, XOR and NOT gates. These logic gates are set up in order to do one specific task or several tasks simultaneously. The advantage of a non programmable hardware module is that it is significantly faster than a processor and can do several tasks simultaneously. It is also considerable cheaper to produce non programmable hardware modules if the task to be solved are simple. However, when a non programmable hardware module is produced, it can not be changed. If it is a requirement for another or a different function, a new module have to be produced. This is where a processor have it biggest advantage.

A processor has the same building blocks as non programmable hardware, that is logic gates. The difference is that the processor can perform different tasks decided by the program on the processor. The advantage of this is that a processor can do lot of

different tasks on the same circuit by loading a different program to it. Nevertheless, a processor is huge and costs a lot compared to simple hardware modules. In addition, the processor can only do one task at a time and will often need long time to complete a task.

Because of the drawbacks of a processor, many microcontroller producers have concluded that using non programmable hardware is the best solution for serial communication with external devices. This Includes, among others, Atmel [2], Texas Instruments [5] and Energy Micro [6].

2.2 Processor Architecture

There are many different types of processors, but they all have some mutual components that are essential for the processor to be able to operate. Figure 2.1 gives an example of a simple processor.

The Arithmetic Logic Unit (ALU) is the component that does calculations and operations on the data. The two inputs A and B are the data inputs and FS is the function select input. The FS input selects what function to be done with the data on input A and B. This function can for example be to add data A with data B. The result is given on the output F. The V, C, N and Z signals are status bits that give information about the data on the output F.

V: Overfow Indicator
C: Carry Flag
N: Negative Flag
Z: Zero Flag

The register file is the temporary memory where the data currently being processed is stored. AA and BA gives the data address of the data to be placed on the data bus A and B. DA gives the address of where in the register to write the data from the ALU. WE is the write enable signal and is set high only if the data on the D Data bus is to be stored. The A Data, B Data and D Data bus all have the same width that is given by the width of the registers.

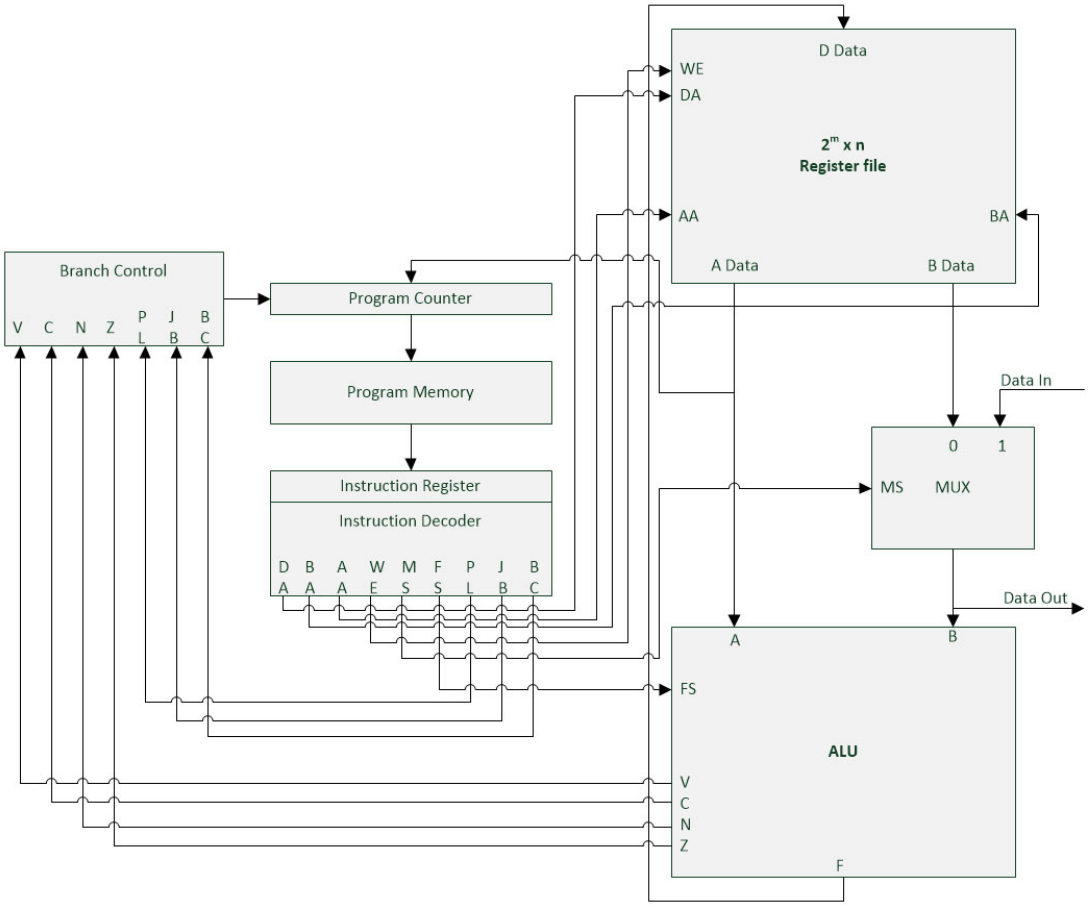

FIGURE 2.1: A Simple Processor

The instruction register receives an instruction by the program memory. This instruction is then decoded by the instruction decoder and then the instruction decoder sets all the pins to the value corresponding to the instruction to be executed. For example, an instruction could be to add register 1 with register 2 and store the added value in register 3. This would result in the following events:

1: The instruction register will be loaded with the instruction.
2: The instruction decoder will decode the signal in the.
instruction register and set its outputs.
3: The outputs are: DA=3, BA=2, AA=1, WE=1, MS=0, FS=ADD
PL=0, JB=0, and BC=0.
4: The instruction will be executed.

The program memory is where the program to be executed is stored. The program is as a list with all the instructions stored in the order they are to be executed in. The program counter is a pointer that points to the instruction that is to be executed. For each instruction executed, the program counter is incremented with one and the

next instruction will be executed.

The branch control is controlling the program counter. If PL is set to 1, a branch or jump is called. JB determines if it is a branch or jump. If JB is set to 0, the program counter is loaded with the data on the A Data bus. If JB is set to 1, a conditional branch is called. BC will then select the branch condition from the status bits.

2.3 First Peripheral Processor

The first computer that used a peripheral processor to control Input/Output signals was the CDC 6600 [7] [1]. The CDC 6600 was a supercomputer developed by Seymour Cray for the firm Control Data Corporation in 1967. This supercomputer had one CPU and ten identical peripheral processors. The Peripheral processors was much smaller than the CPU. Each peripheral processor had a register that could store 4096 12-bit words and a repertoire of 62 instructions. Each peripheral processor also had access to the central storage and access to all the 12 peripheral channels. Figure 2.2 gives an illustration of how the Peripheral Processors are included in the CDC 6600 computer.

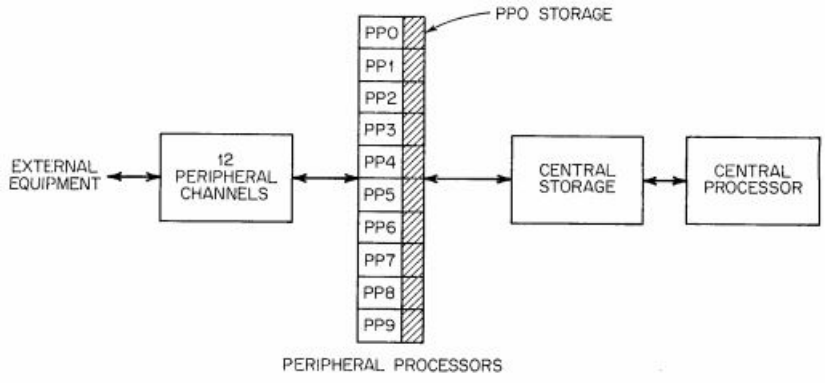

FIGURE 2.2: Peripheral Processors in the CDC 6600 supercomputer [1]

Having ten independently constructed processors would use a lot of area and cost a lot. Therefore did each peripheral processor actually share one ALU. This could be done because of the time needed to read and write to the registers. For example each peripheral processor would use 1000 nano seconds to add the value register in register A to the value in register B and store it in register C. But the ALU would actually

only use 100 nano seconds to perform the add operation. For that reason, one ALU could be shared between 10 processors to maximize its efficiency. Each peripheral processor was given a time slot where it had access to the ALU.

2.4 USART

The functionality of the USART is extracted from [2] and [8]. Universal Synchronous and Asynchronous Receiver and Transmitter (USART) is a popular method for transmitting serial data. Its main features, among others, is that it can switch between synchronous and asynchronous communication and it can have different length on the data frames. Synchronous operation uses a clock and data line while there is no separate clock accompanying the data for asynchronous transmission. Both transmission and reception can occur at the same time. This is known as full duplex operation.

The USART consists of three main modules that is a transmitter, a receiver and a baud generator. Additional the USART also have registers for storing status flags and data. A overview of a USART is given in figure 2.3

Clock Generation

The USART can operate on several different clock frequencies and baud rates. The clock frequency decides the baud rate of the USART. When the USART operates as a master, the user have to set the baud rate of the USART before it can be activated. This is done by writing a value to the Divisor Register. The Baud Generator will then use this value to calculate the baud rate. The baud rate is a function of both the value in the Divisor Register and the CPU clock speed. Equation 2.1 gives the formula for calculating the Divisor value if the preferred baud rate is known. The baud rate can commonly range from 2400 bit/s to 1 Mbit/s depending on the clock frequency the master is operating on.

$$DivisorValue = \frac{Clk_{CPU}}{16 * (BaudRate)} - 1 \tag{2.1}$$

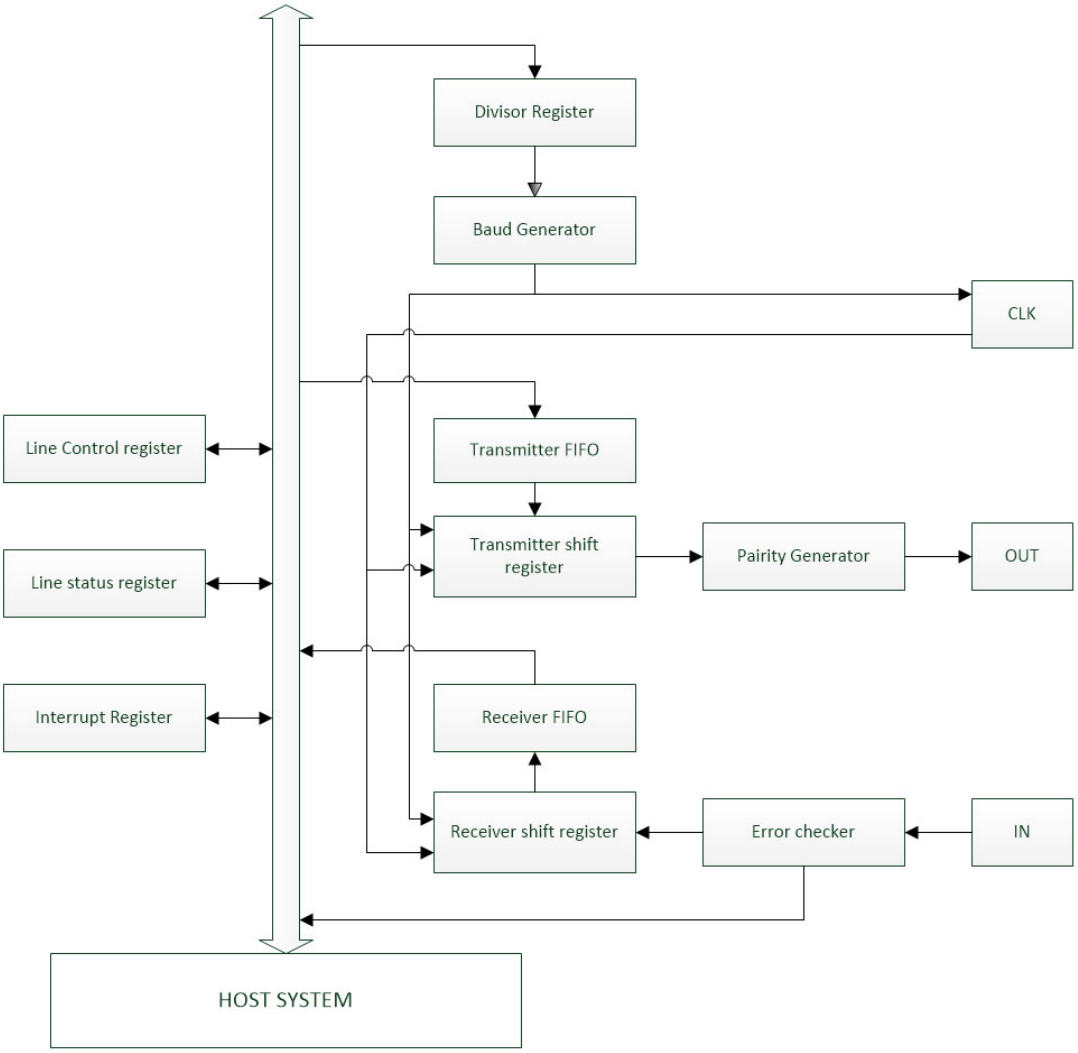

FIGURE 2.3: Block diagram of a USART module

Registers

There are several registers in the USART for storing the different control signals and status signals. The different registers are listed in table 2.1, to table 2.4.

LCR	
Bit number	Function
7	**Transmit Data Bit 8** When bit 5 in LCR is set this bit is the 9th data bit of the transmitted character.
6	**Receive Data Bit 8** When bit 5 in LCR is set this bit is the 9th data bit of the received character.
5	**9 Bit Characters** 0 = 5 to 8 bits character length. 1 = 9 bits character length.
4	**Even Parity** 0 = enable odd parity 1 = enable even parity
3	**Parity enable** 0 = send no parity bit after data bits 1 = generate or check a parity bit after data bits.
2	**Stop bits** 0 = transmit 1 stop bit after data bits are sent 1 = transfer 2 stop bits.
[1:0]	**Character length** 00 = transmit 5 bits of data per frame 01 = transmit 6 bits of data per frame 10 = transmit 7 bits of data per frame 11 = transmit 8 bits of data per frame

Table 2.1: Line Control Register

LSR	
Bit number	Function
7	**Error flag** 0 = The receive FIFO has no errors. 1 = the receive FIFO has parity, farming, or break condition errors.
6	**Transmitter empty flag** 0 = transmitter is shifting data out to the serial line. 1= the transmit FIFO and transmit shift register are both empty.
5	**Transmit FIFO empty flag** 0 = the transmit FIFO has at least 1 peace of data remaining. 1 = the transmit FIFO is empty
4	**Break indication** This is set to 1 if the receiver detects a string of 0 for longer than a full word transmission time. Under this condition the FIFO is loaded with a 0x00 character and the receiver remains idle until it detects a valid start bit.
3	**Farming error** This is set to 1 when the receiver detects an invalid stop bit.
2	**Parity error** This is set to one when the receives character has an incorrect parity bit.
1	**Overrun error** This is set to 1 when the receiver FIFO is full and a completed character in the receive shift register is destroyed.
0	**Receiver data ready** 0 = the receive FIFO is empty 1 = data is ready to be read from the receive FIFO.

Table 2.2: Line Status Register

Interrupt Register	
Bit number	Function
7	**USART Mode Select** 0 = Asynchronous operation 1 = Synchronous operation
6	**Enable Receiver Line status Interrupt** Set this bit to 1 to enable this interrupt.
5	**Enable Transmit FIFO Empty Interrupt** Set this bit to 1 to enable this interrupt.
4	**Enable Receive Data Available Interrupt and Timeout Interrupt** Set this bit to 1 to enable this interrupt.
[3:1]	**Interrupt ID** These bits are used to identify the highest priority of the interrupts
1	**Interrupt pending** 0 = an interrupt is pending 1 = there are no interrupts

Table 2.3: Interrupt Register

Divisor Register	
Bit number	Function
[15:0]	**Baud Rate**

Table 2.4: Divisor Register

Frame Format

When there is no data to be sent, the serial line is high. The first bit of a frame is the start bit. This bit is always low and one clock cycle long. The next bits are the data bits and these bits can vary from 5 to 9 bits. The least significant bit is sent first and the most significant bit is sent last. The bit following after the data bits is a parity bit only if parity is enabled in the Line Status Register. Last there is a stop bit and can be defined as one or two clock cycles long. The stop bit is always high. The stop bit can immediately be followed by a new start bit if there is more data to be transferred. If there is no more data to be transferred the line will stay high.

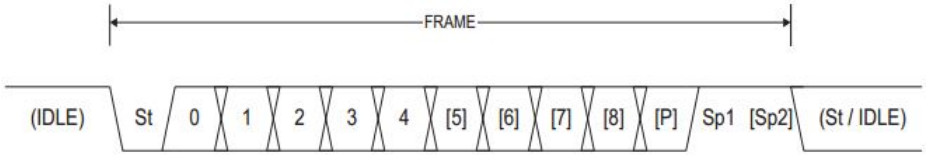

FIGURE 2.4: Frame Formats [2]

St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit, always high.
Sp	Stop bit, always high.
IDLE	The line is high when there is no transfer.

Transmitter

For transmitting signals with the USART the processor first have to set the flags in the LCR and IR registers so that the USART operates whit the correct mode. The baud rate also have to be set before a transmission can be started. When the baud rate and the control flags has been set, the processor can transmit data by writing to the transmitter address as it was a memory space.

When the USART receives data it places the data into a FIFO buffer. The transmitter pops data from the FIFO to a shift register and the shift register performs a parallel to serial conversion by shifting data out on the serial line for transmission.

The transmitter also generates the start bit, parity bit, and stop bit. When the transmission is complete, the transmitter sets the Transmitter Empty flag to 1 to indicate that the transmission is complete.

Receiver

The receiver is more complex than the transmitter. The receiver constantly monitors the receive serial line. When the receiver notices that the line goes low, it starts a sampling process to verify that it is a valid start bit. This sampling is done with a frequency that is 16 times faster than the baud rate. When the start bit goes from high to low the receiver will do 3 samples of the start bit. These samples will be done on sample 8, 9 and 10. the receiver will then compare the 3 samples and if two or more of the samples are logical low, the start bit is considered as a valid start bit. If only one of the samples are logical low, the start bit is considered as a noise spike and the receiver starts looking for a the next high to low transition. An illustration of the sampling process is given in figure 2.5.

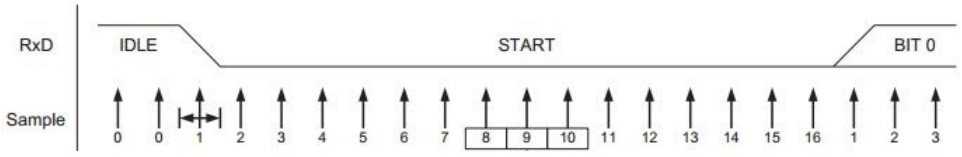

FIGURE 2.5: Start Bit Sampling [2]

If the start bit is considered as a valid start bit, the data recovery can begin. For each data bit received, the same sampling that was done with the start bit is done with all the bits received. The sampling of data bit and parity bit is the same and is illustrated in figure 2.6

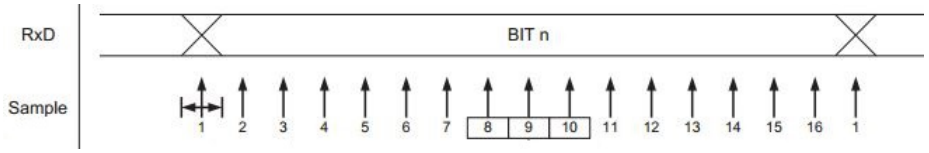

FIGURE 2.6: Data and Parity Bit Sampling [2]

The same sampling is done for the stop bit. But there is a difference after the last sample. After the 10th sample is done, the receiver is immediately ready for a new

high to low transition that indicating a new start bit. Figure 2.7 illustrates the sampling of the stop bit and the earliest possible beginning of a new start bit. The first point where a new start bit can occur is on the point marked (A), and (B) marks a stop bit of full length.

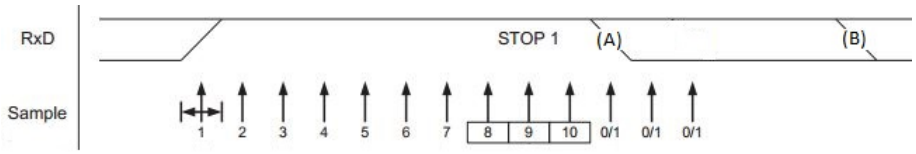

FIGURE 2.7: Stop Bit Sampling and Next Bit Sampling [2]

The reason for the variable length for the stop bit is because of the possibility for asynchronous transmission. The transmitter and receiver does not necessary operate at the exact same clock frequency. If the transmitter is sending frames at too fast bit rate, the variable stop bit length will minimize the difference. If the transmitter is sending frames too fast or too slow compared to the internally generated baud rate, the receiver will not be able to synchronize the frames. This will only be noticed if the receiver expects a stop bit, but the received bit is logical low. When that happens the Farming Error flag will be set in the Line Status Register. If the USART is using synchronized transmission, the transmitter will send a clock to the receiver. The data reception will have the same protocol for sampling data, but there will not be any mismatch between the transmitter and receiver baud rate.

After one frame is shifted in to the receive Shift Register, the received frame is pushed in to the receive FIFO and the Receiver Data Ready flag is set to one. This tells the processor that at least one data frame is ready to be read. The receiving operation can continue and new frames can be pushed into the FIFO. If the FIFO gets full and a frame is written over a frame that yet is to be read by the processor, the Overrun Error flag is set. If the parity bit does not match the received data the Parity Error flag will be set.

2.5 AMBA Bus

The Advanced Microcontroller Bus Architecture (AMBA) is a standard designed for on-chip communication on high performance embedded microcontrollers [3]. The AMBA specification is divided into three different buses:

- Advanced High-performance Bus (AHB)
- Advanced System Bus (ASB)
- Advanced Peripheral Bus (APB)

The AMBA AHB and ASB are in general used for the main bus system in a microcontroller. The AHB is for high-performance, high clock frequency system modules and is the most advanced bus of the three different AMBA buses. The ASB is an alternative for the AHB where the high performance of the AHB is not required. The AMBA APB is for peripheral modules and is optimized for minimal power consumption. Figure 2.8 gives an overview of how a typical AMBA system is connected.

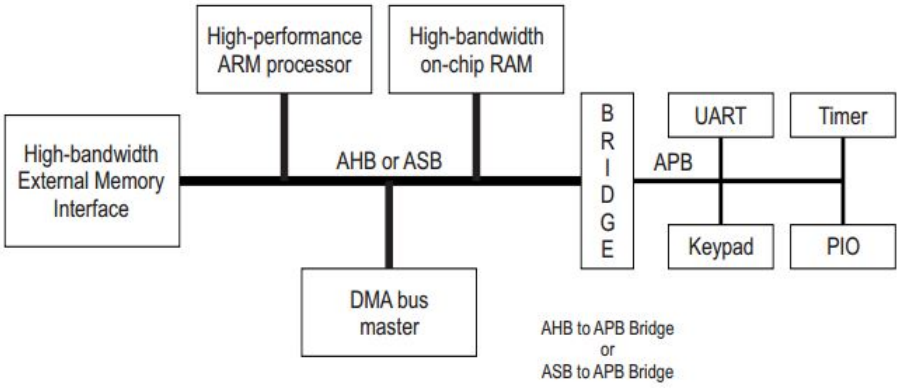

FIGURE 2.8: A typical AMBA system [3]

AMBA APB

The AMBA APB is a bus optimized to be used with peripheral modules and is the most relevant AMBA bus for this thesis. The APB is connected to the main bus via a bridge that handles the communication between the main bus and the APB. All the signals on the APB is controlled by a master connected to the main bus and all the peripheral modules are connected as slaves to the APB. A big advantage with the APB is that it is static when not in use and will therefore use minimal power in this situation.

The signals used in the APB bus is named with the single letter P prefix. Table 2.5 shows the list of signals in the bus and a description.

Name	**Description**
PCLK	Bus clock
PRESETn	Reset signal for bus, active low.
PADDR	Address bus, 32 bit wide
PSELx	A signal to each bus slave to indicate that the slave is selected. There is a different PSELx signal for each bus slave.
PENABLE	Used to indicate the second cycle of a data transfer.
PWRITE	When set high, the signal indicates a write access, when low a read access.
PRDATA	Read data. Driven by the selected slave, 32 bit wide.
PWDATA	Write data, 32 bit wide.

Table 2.5: AMBA APB signals

The operation of the APB can be represented with three states: IDLE, SETUP and ENABLE. The idle state is the default state of the bus. When a transfer is required the bus moves to the SETUP state, stays in this state for one clock cycle and then moves to the ENABLE state. The ENABLE state lasts for one clock cycle and if a new transfer is required, the bus will move directly to the SETUP state after the ENABLE state. If no new transfer is required, the bus will move back to the IDLE state. Figure 2.9 shows both a write transfer and a read transfer. The first clock cycle, T1->T2, shows the IDLE state of the bus, T2->T3 shows the SETUP state and T3->T4 shows the transfer state.

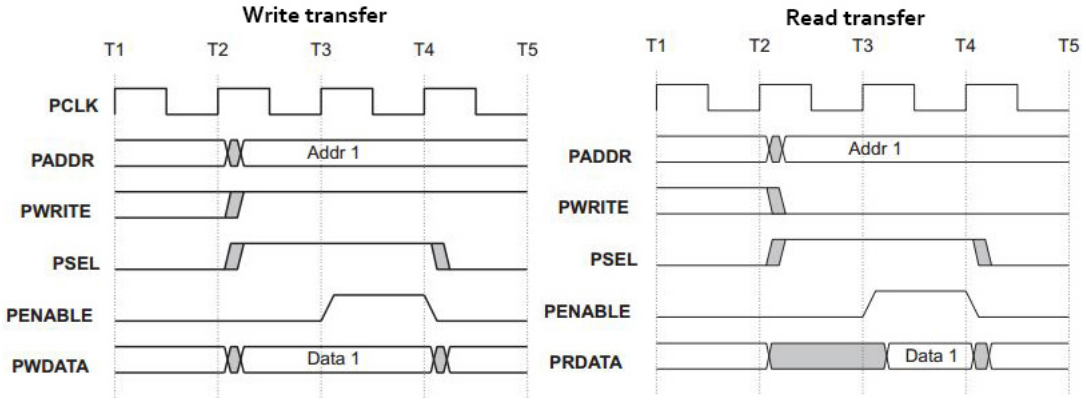

FIGURE 2.9: Write and read transfer for APB bus [3]

The write transfer starts with the SETUP state. In this state PADDR, PWRITE, PSEL and PWDATA will be set. The next clock cycle the ENABLE state will be asserted. The signals set in the SETUP state will remain the same and the PENABLE signal will now be set high.

The timing of PADDR, PWRITE, PSEL and PENABLE is the same for the read transfer, but PRDATA will not be set until the ENABLE state is reached.

Chapter 3

Summary of Preliminary Work

Some of the content in this chapter is the same as in the preliminary work.

3.1 Emulation

The preliminary work was a joint effort between the author and Joacim Dybedal. This was the specialization project and was done the autumn 2012. The main goal of the work was to find a possible solution for how a peripheral processor could be implemented and compare this solution to existing hardware modules. This was done by first emulating an SPI and UART module by writing software with the programming language C, and then run this software on an Atmel AVR. The software was then used to extract the instruction set used by the AVR to explore what operations an implementation of a peripheral processor should be able to perform. The instruction set extracted was then used to create a new and optimized instruction set for a peripheral processor. Also, a solution of how the peripheral processor could be implemented was suggested. This solution was then compared with existing hardware modules with respect to area, performance and power consumption.

When running the software to emulate a UART module on the AVR, a significant issue with having a processor controlling the data transfer was found. The AVR was running with a clock frequency of 11.05925 MHz. This resulted that the maximum

baud rate of the UART could be 2400 bits/s. A non-programmable hardware module can support a baud rate up to 250 KHz with the same clock frequency. In addition to the problem with the baud rate, it was also a problem that full duplex mode not was possible. The problem could be solved by simulating full duplex mode by switching between transmitting and receiving, but this would result in a lower baud rate.

3.2 Instruction Set

The instruction set extracted from the UART and SPI software may not include all the instructions needed. This is because the UART program did have less functionality than a fully functional UASRT. Furthermore, the peripheral processor is also supposed to run other serial communication protocols, such as TWI and USB. However, the time limit made it impossible to investigate all the protocols and UART and SPI where therefore prioritized. The UART was the most resource-demanding program and used a total of 26 registers in the AVR. However, the compiler does not care about reusing registers if it is not necessary. An estimation that 16 registers are enough should therefore be valid.

Two suggestions of an optimized opcode was given based on the instruction set extracted from the UART and SPI software. The first suggestion was a reduction of the opcode to 16 to 14 bits. A reduction of the opcode length from 16 to 14 bit will reduce the amount of logic needed to decode the opcode. However, it will not reduce the space needed to store the instructions. This is because the registers have a size of 8 bit and the instructions will therefore be stored in two registers anyway. Therefore, a second suggestion was made. This suggestion was to divide the instruction set into two parts. One part with 8 bit instructions and one part with 16 bit instructions. With this solution, no bits will be wasted when the instructions are stored and the instructions with 8 bits will be transferred to the instruction register faster. Table 3.2 and 3.1 gives the instructions included in the second solution.

Instruction	Opcode	Instruction	Opcode
MOVW	0000 0000 0ddd 0rrr	STS	0011 0kkk dddd kkkk
CPC	0000 0001 dddd rrrr	RJMP	0100 kkkk kkkk kkkk
SBC	0000 0010 dddd rrrr	RCALL	0101 kkkk kkkk kkkk
ADD	0000 0011 dddd rrrr	LDI	0110 KKKK dddd KKKK
CPSE	0000 0100 dddd rrrr	BRCS	0111 00kk kkkk k000
CP	0000 0101 dddd rrrr	BRLO	0111 00kk kkkk k000
SUB	0000 0110 dddd rrrr	BREQ	0111 00kk kkkk k001
ADC	0000 0111 dddd rrrr	BRMI	0111 00kk kkkk k010
TST	0000 1000 dddd dddd	BRVS	0111 00kk kkkk k011
AND	0000 1000 dddd rrrr	BRLT	0111 00kk kkkk k100
CLR	0000 1001 dddd dddd	BRHS	0111 00kk kkkk k101
EOR	0000 1001 dddd rrrr	BRTS	0111 00kk kkkk k110
OR	0000 1010 dddd rrrr	BRIE	0111 00kk kkkk k111
MOV	0000 1011 dddd rrrr	BRBS	0111 00kk kkkk ksss
IN	0000 1100 dddd aaaa	BRCC	0111 01kk kkkk k000
OUT	0000 1101 rrrr aaaa	BRSH	0111 01kk kkkk k000
SBR	0001 KKKK dddd KKKK	BRNE	0111 01kk kkkk k001
COM	0010 0001 dddd 0010	BRPL	0111 01kk kkkk k010
NEG	0010 0001 dddd 0011	BRVC	0111 01kk kkkk k011
SWAP	0010 0001 dddd 0100	BRGE	0111 01kk kkkk k100
INC	0010 0001 dddd 0101	BRHC	0111 01kk kkkk k101
ASR	0010 0001 dddd 0110	BRTC	0111 01kk kkkk k110
ROR	0010 0001 dddd 0111	BRID	0111 01kk kkkk k111
DEC	0010 0001 dddd 1000	BRBC	0111 01kk kkkk ksss
JMP	0010 0001 kkkk 110k	BLD	0111 1000 dddd 0bbb
	kkkk kkkk kkkk kkkk	BST	0111 1001 dddd 0bbb
CALL	0010 0001 kkkk 111k	SBRC	0111 1010 rrrr 0bbb
	kkkk kkkk kkkk kkkk	SBRS	0111 1011 rrrr 0bbb
ADIW	0011 1kkk kddd kkkk	SUBI	0010 01KK dddd KKKK
MUL	0010 0010 dddd rrrr	SBCI	0010 10KK dddd KKKK
SER	0010 0001 dddd 1001	CPI	0010 11KK dddd KKKK
LDS	0011 0kkk dddd kkkk	ORI	0111 11KK dddd KKKK

Table 3.1: 16 bit opcode

Instruction	Opcode	Instruction	Opcode
NOP	1000 0000	SEC	1001 1sss
ICALL	1000 0001	SEH	1001 1sss
IJMP	1000 0010	SEI	1001 1sss
RET	1000 0011	SEN	1001 1sss
RETI	1000 0100	SES	1001 1sss
SLEEP	1000 0101	SET	1001 1sss
CLC	1001 0sss	SEV	1001 1sss
CLH	1001 0sss	SEZ	1001 1sss
CLI	1001 0sss	POP	1010 dddd
CLN	1001 0sss	PUSH	1011 rrrr
CLS	1001 0sss	LSL	1100 dddd
CLT	1001 0sss	ROL	1101 dddd
CLV	1001 0sss	TST	1110 dddd
CLZ	1001 0sss	LSR	1111 dddd

Table 3.2: 8 bit opcode

3.3 Architecture

When studying the possibilities of how the architecture of the peripheral processor could be, one question arisen. This was what parts of the processor that could be shared between the CPU and the peripheral processor. It would be a benefit to the cost of the peripheral processor to share the program memory with the main CPU. However, if the program memory was to be shared between the CPU and the peripheral processor, all the instructions to the peripheral processor had to be transferred between the memory and the peripheral processor with the main bus in the system. The problem with this is that the CPU will be unable to do anything if the main data bus is occupied by the peripheral processor.

It was also looked into the sizes of the different memories and what extra logic that was needed to control the peripheral processor. A suggestion of how the peripheral processor could be is given in figure 3.1.

The core of the peripheral processor will primarily consist of a register file, ALU, instruction decoder, SRAM and other control logic. The core is then connected to external modules with an internal bus that is specifically designed for this peripheral

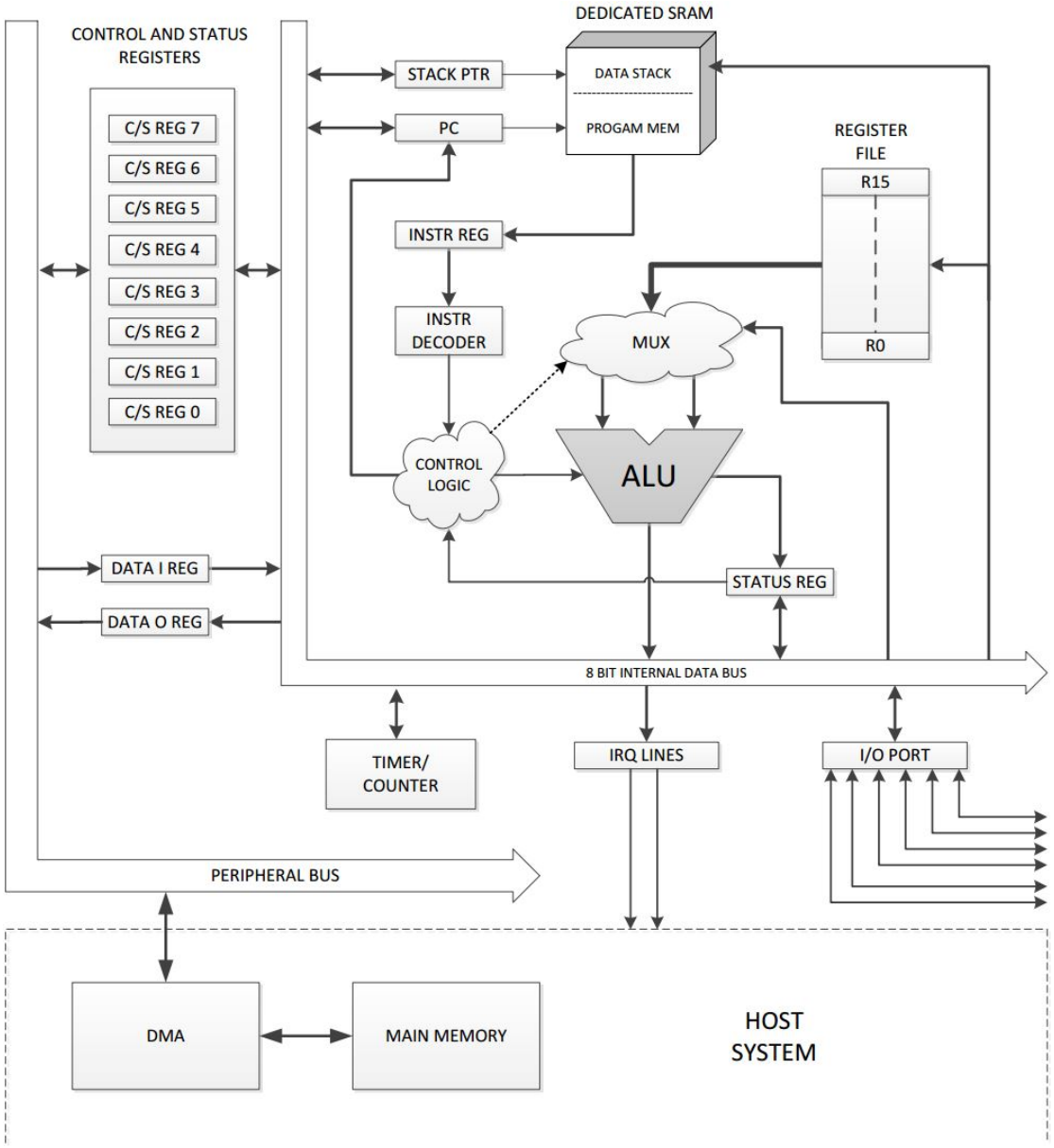

FIGURE 3.1: PPU Block Diagram [4]

processor. C/S REG 0 to 7 are the control and status registers. These registers are to set the settings and to read the status of the transfer protocol running on the processor. DATA I REG and DATA O REG are the registers that the CPU writes the data to be transmitted to and reads the data that is received. The IRQ LINES are the interrupt and request signals from the peripheral processor to the CPU. The DATA STACK and PROGRAM MEMORY have a size of 4096 bytes. Furthermore, the peripheral processor will have direct connection to the input and output ports. Additionally, the peripheral processor will need a 16 bit timer to control the transfer

baud rate. The host system will be connected to the peripheral processor with an AMBA APB bus, shown as the peripheral bus.

3.4 Performance

During the programming of the UART two issues about running a UART on a processor was found. The first one is that full duplex mode is not possible. This is because a processor cannot do multiple tasks simultaneously. The other issue is that the baud rate have to be significantly lower than it can be on non programmable hardware. As mentioned in chapter 2.4 the receiver have to have a sampling frequency that is 16 times higher than the baud rate. In addition to this the processor will have to use several clock cycles between each sample.

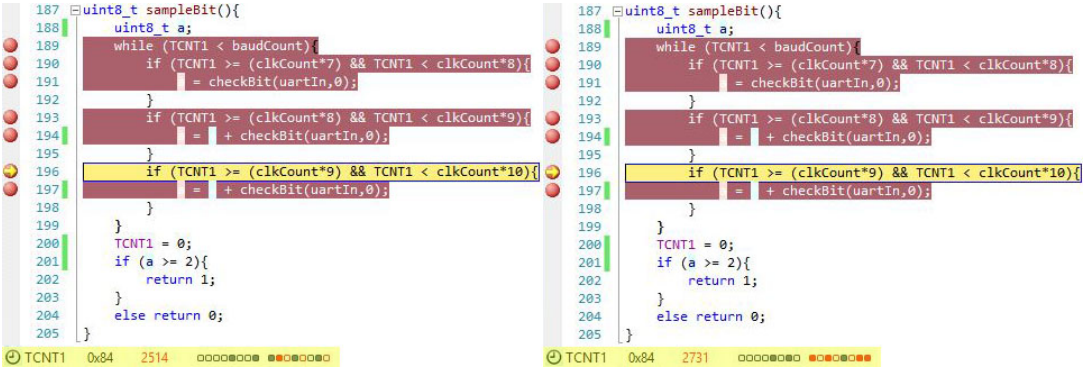

FIGURE 3.2: Excerpt from the UART C-code

Figure 3.2 gives an illustration of how many clock cycles needed from the second sampling to the third. In this example the divisor value (clkCount) is set to 287 and therefore the third sample will be done when TCNT1 is between 2583 (clkCount*9) and 2870 (clkCount*10). When the program reaches line number 196 the first time TCNT1 will be less than 2583 and the while-loop will have to do one more loop. The next time the program reaches line 196 TCNT1 is 2731. Now the criteria for the third sample is fulfilled, but the loop needed a total of 217 clock cycles to circle one time. In this example divisor value was chosen to be 287. This value is taken from [2] and results in a commonly used baud rate at 2400 bits/s.

3.5 Cost Analysis

The suggestion given in chapter 3.3 was then used to estimate the cost of the peripheral processor. The total cost was given in NAND2 equivalents and was calculated separate for the registers in each module. The formula used for calculating the number of NAND2 gates per register is given in equation 3.1. This is a formula Atmel has provided. A list of the different registers with each corresponding number of bits and NAND2 equivalents is given in table 3.3.

$$TotalNAND2 = NumberOfBits * 10 * 2 \tag{3.1}$$

Register	Number of bits	Number of NAND2 gates	Percent
Register file	128	2560	2.4%
C/S Registers	64	1280	1.2%
Data I/O	16	320	0.3%
Stack Ptr	8	160	0.2%
Program Counter	12	240	0.2%
Instruction Register	16	320	0.3%
I/O Ports	8	160	0.2%
Status Register	8	160	0.2%
Timer/Counter	104	2080	2.0%
Sum without SRAM	364	7280	6.9%
SRAM	32768	98304	93.1%
Total	33132	105584	100%

Table 3.3: Sum of logic

For comparison, the USART module in the ATmega128 microcontroller have a total of 11 8-bit registers. This results in a total of about $11 * 8 * 10 * 2 = 1760$ NAND2 gates. Furthermore, the SPI module have a total of 4 8-bit registers. This results in a total of about $4 * 8 * 10 * 2 = 640$ NAND2 gates. Last, the TWI module have a total of 5 8-bit registers that results in a total of about $5 * 8 * 10 * 2 = 800$ NAND2 gates. These three modules will therefore require a total of about $1760 + 640 + 800 = 3200$ NAND2 gates.

Chapter 4

Procedure

The block diagram of the peripheral processor given in figure 3.1 was used as a specification together with the block diagram given in figure 2.1 to implement the peripheral processor in HDL. It is important to notice that figure 2.1 and 3.1 was only used as a guideline to how the peripheral processor was to be implemented and was not used as a complete specification.

The main HDL language in Amel is Verilog. It was therefore natural to use Verilog to implement the processor in HDL. The main advantage of using this language was that Atmel then could give the author access to all the necessary tools that Atmel utilizes.

4.1 Verilog Implementation

All the modules discussed have a reset signal. This signal will reset all modules when set to logical 0, but it will not be mentioned when discussing the different modules. Also, some names of the input and output signals have been changed from their original names in figure 2.1 and 3.1. Therefore, all names mentioned in this chapter will have the same name as the final result given in chapter 5.

When starting on the code for the processor, the theory in chapter 2.2 was the starting point. First the register file and the ALU was implemented. The register file was given the same inputs and outputs as given in figure 2.1 but with a small name change. When implementing the ALU, it was decided that functions important for the processor to work would be implemented first, and that more functions would be added later if time allowed it.

After the first draft of the register file and the ALU was implemented, the implementation of the instruction decoder was started. Before implementing the instruction decoder, a decision of what size and layout of the opcode to be used had to be made. It was chosen to use the same opcode layout as Atmel have on their 8-bit AVR. The reason for the choice will be discussed in chapter 6. Similarly as the ALU, only a few instructions where included in the instruction decoder at first.

The branch control and the program counter was implemented after the instruction decoder was tested and working. The program counter was not implemented as a separate module, but was included in the branch control. Also, two extra signals, bset and offset, was added between the instruction decoder and the branch control.

As concluded in the preliminary work, an SRAM would be used as program memory and stack to save area. The SRAM module was provided by Amel, but needed a wrapper to be able to communicate with the rest of the system. First it was planned to have a shared SRAM between the program memory and the stack, but this was changed during the implementation to two separate SRAM modules for the program memory and the stack.

The stack was moved out of the CPU core and the read and write procedure to the stack was set to be done via the data bus. A timing issue occurred when reading and writing to the stack. To correct this, two signals where added to the branch control and instruction decoder: hold_pc and done_hold. hold_pc was also added to the program memory.

After reading and writing to the stack was working properly, all the other external modules where implemented. And the internal data bus was extended each time a new external module was added.

4.2 Test Programs

When all the modules were working and connected to the system, two test programs were made. The first test program was made to check that all the instructions of the peripheral processor was function properly. This was done with a test bench that inserted the opcode for all the different instructions into the program memory at the beginning of the simulation and then let the processor run trough all the instructions.

The second test program was made to be more realistic. This test program was loaded into the peripheral processor by simulating the same method the host system would use to program it. To do this, the test bench was made to load the instructions into the memory one by one each clock cycle by using the input signals from the APB bus into the peripheral processor.

The task of the test program was to behave as a simple UART module to verify that the processor was able to do the most basic operations needed by a UART module. For the transmitter part, this included checking the status register for new data on the input register, loading 8 bit from the input register, calculating parity bit and transmitting one data frame including start bit and stop bit.

The receiver part of the test program was to recognise a start bit on the input port, shift 8 bits into a register, transfer this data to the output register and then sending an interrupt to the host system.

4.3 Synthesis

When the peripheral processor was working on the RTL level, Spyglass was used to check for problems with the design that is not possible to synthesize. The errors

and warnings where corrected and a synthesize was done. Synthesis was done with Design Compiler by using a script that Atmel provided. This script did also analyse the peripheral processor and printed several reports of the specifications of the circuit. The most important reports where area, timing and fault coverage.

After the synthesis was done, Formality was run to check that all the modules did behave the same way before and after the synthesis. When this test was succeeded, a clock tree synthesis was done. This synthesis did a layout of the circuit so that more specific reports could be presented.

Last, a power simulation was done. This was done by simulating the UART program running on the peripheral processor with the same clock tree and layout made in the previous step.

Chapter 5

Results

5.1 CPU Core

Figure 5.1 gives an illustration of the core of the peripheral processor. It has several common features as the example processor given in chapter 2.2, but there are also some major differences. Table 5.1 and 5.2 gives a short description of all the in/out signals connected to each module including the bit size of the signals.

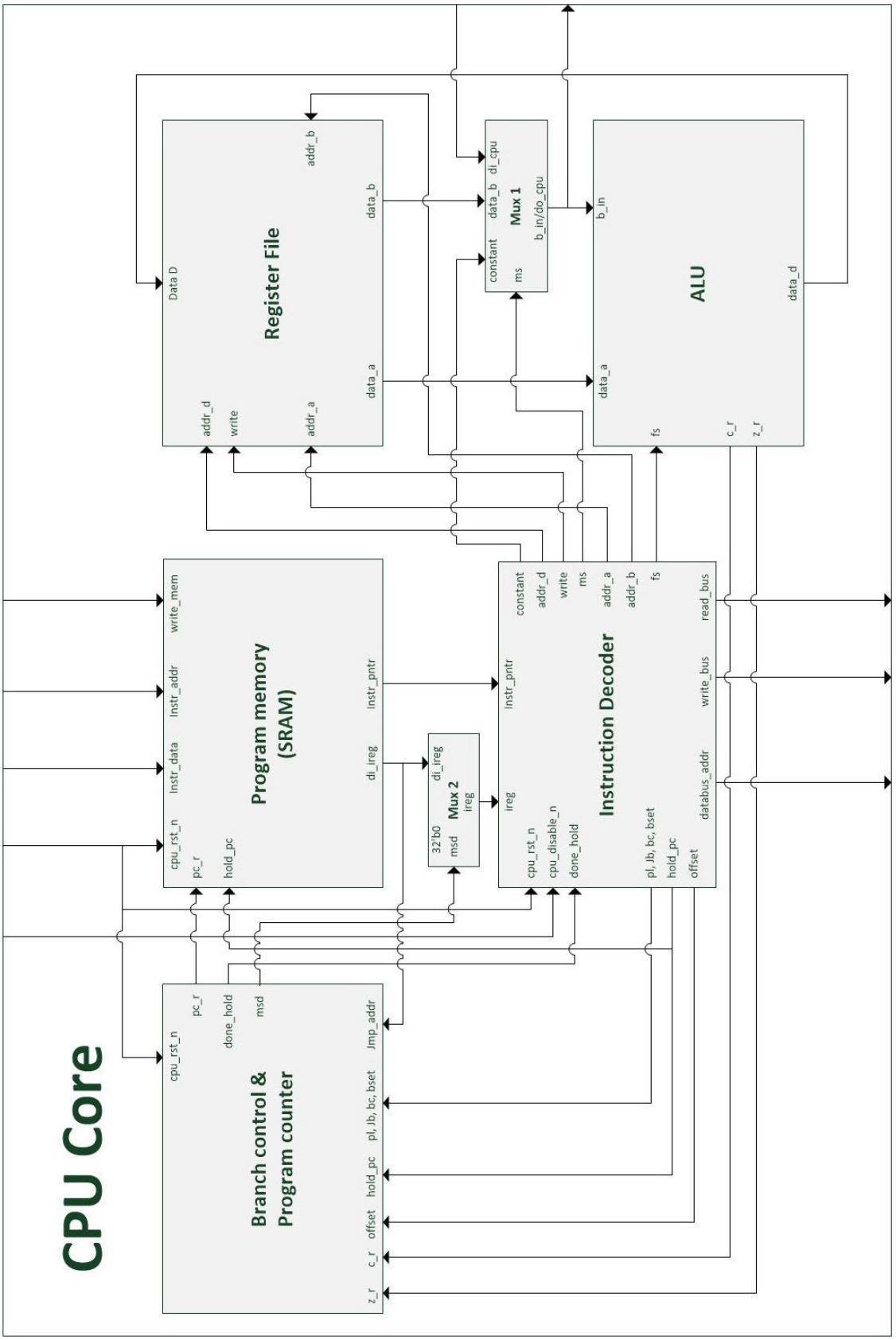

Figure 5.1: PPU Core Block Diagram

Signals from the register file:		
Signal	Bit Size	Description
data_a	8	Data to be transmitted from the register pointed to by addr_a to ALU.
data_b	8	data to be transmitted from the register pointed to by addr_b to Mux.
Signals from the mux:		
b_in	8	Data to be transmitted from the mux to the ALU.
do_cpu	8	Data to be transmitted to external modules.
Signals from the ALU:		
data_d	8	data to be written to the register pointed to by addr_d.
c_r	1	Carry flag
z_z	1	Zero flag
Signals from the instruction decoder:		
addr_a	4	Points to the register that is the output on data_a.
addr_b	4	Points to the register that is the output on data_b.
addr_d	4	Points to the register that the content on data_d is to be written to.
write	1	Is set high when the content on data_d is written to a register.
constant	8	Contains a constant given in an instruction.
ms	2	Signal to select input on the mux.
fs	5	Selects ALU function.
read_bus	1	Is set high when the internal data bus is to be read.
write_bus	1	Is set high when data is written to the internal data bus.
databus_addr	7	Gives the address of the location that data is to be written to or read from via the internal data bus.
offset	7	Used when branching. Gives the number of steps the program counter will jump. Is given in two's complement and can be both positive and negative.
hold_pc	1	When held high, the program counter will not increment. Used for instructions that need more than one clock cycle to complete.
pl	1	When set high, a branch or jump is called.
jb	1	Determines if a branch or jump is called.
bc	2	Determines what status flag that triggers a branch.
bset	1	Determines if the branching is triggered on high or low status flag.

TABLE 5.1: Signal description

Signals from the branch control:		
pc_r	11	Program counter
done_hold	1	This signal is set high one clock period after hold_pc is set high. Used when writing to and reading from the stack.
msd	1	Used when jumping. This signal decides if the signal to the instruction decoder comes from the program memory or if it is set to zero.
Signals from the program memory:		
di_ireg	32	The instructions to be decoded. Is 32 bit wide and contains two instructions.
instr_pntr	1	A pointer that points to the part of the instruction register that is to be decoded.
Signals from the exterior:		
Signal	Bit Size	Description
instr_data	32	Used when programming the memory. Contains the data to be written to the program memory.
inst_addr	9	Used when programming the memory. Points to the memory location where the instruction data is to be written.
write_mem	1	Is set high if the program memory is to be programmed.
cpu_disable_n	1	When held low, the CPU will be paused.
cpu_reset_n	1	When set low, the CPU will jump to the start of the program memory. No registers will be reset.

TABLE 5.2: Signal description

Register File

The register file have sixteen registers each with the size of eight bit. Writing to the register will occur at the rising edge of the clock and when write is set to logical one. The input on data_d will then be written to the address given by addr_d. The output on data_a and data_b is not driven by the clock, but will always give the data where addr_a and adddr_b is pointing to.

ALU

The ALU is primarily combinatorial and not driven by the clock. The only elements of the ALU that is sequential is the status flags c_r and z_r. These two signals will at the rising clock edge be given the value that was generated by the operation done at that clock edge.

Instruction Decoder

The instruction decoder have been changed a lot compared to chapter 2.2. The instruction register and instruction decoder are now in the same module. Also, the data into the instruction register, di_ireg, is 32 bit, but each instruction is only 16 bit. Therefore an extra signal, instr_pntr, is added to indicate what part of the 32 bit data that is to be decoded. The instruction decoder has no clock and is therefore 100% combinatoric. The signal hold_pc is used when instructions that need two clock periods is executed. Because the instruction decoder do not have a clock, a signal from the branch control is needed to indicate the start of the second clock period. This is the done_hold signal. The two signals cpu_rst_n and cpu_disable_n have the same functionality in the instruction decoder. Both signals will set the instruction decoder to do NOP-operations.

The instruction decoder is using the same opcode layout and addressing mode used in the AVR Instruction Set[9].

Branch Control and Program Counter

The branch control and program counter have been included in one module. The offset signal is used when branching and is given in two's complement. This is because the program counter have to be able to branch in both directions. The msd signal is used when a jump is called. This signal is necessary to prevent the instruction decoder to decode the jump address as an instruction. When a jump is called, the 32 bit in the next program memory location is used as the address for the jump.

This address can accidentally have the same binary number as a valid instruction for the instruction decoder and will cause the processor to execute a random instruction while the branch control is jumping to an address. Therefore the msd signal will set the ireg signal from Mux 2 to be zero when a jump is called.

Program Memory

The Program memory is based on an SRAM module provided by Atmel and have a size of 512 x 32 bits. This size will give room for a program with 1024 16-bit instructions. The program memory consists of a wrapper and the SRAM module. The wrapper is controlling the signals to and from the program memory so that the SRAM module is compatible with the rest of the system. Among others, the wrapper is inverting the clock signal to the SRAM. This is to prevent reading from it to require two clock cycles.

5.2 External Modules

Figure 5.2 gives an illustration of the whole peripheral processor with the bus system and all the modules connected to it.

Internal Data Bus

The 8-bit internal data bus is the bus that connects the CPU core together with all the external modules. The CPU core is the master of the bus and no reading or writing operations are done without the CPU core controlling it. When writing is to be done, the CPU core will send three signals to the data bus: databus_addr, do_cpu and write_bus. The signal databus_addr is a 7-bit signal that is divided into two parts. The first three bits gives the address of the external module where the data is to be sent and the last four bits gives what register in the external module

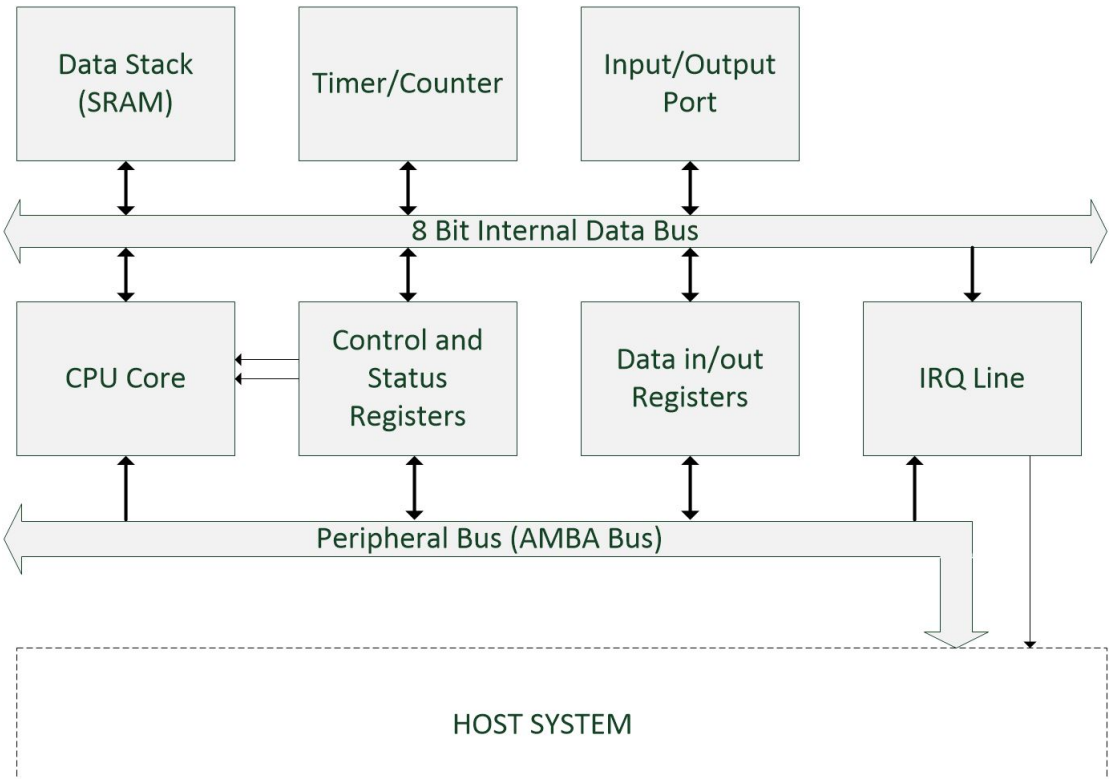

FIGURE 5.2: PPU Block Diagram

that is to be written to. The signal do_cpu is the data signal from the CPU core to the external modules.

For example, if data is to be written to the first register of the control and status register. The CPU core will set the 7 bits in the databus_addr signal to 0010001. The first three bits (001) is the address of the control and status register and the last four bits (0001) is the address of the first register in the control and status register. Also the CPU core will set write_bus high. Because databus_addr points to the control and status register and write_bus is set high, the data bus will set the signal w_cs_reg high, forward the last four bits of databus_addr to cs_reg_addr and forward the data on do_cpu to di_cs_reg. The control and status register will notice that w_cs_reg is set high and therefore write the data on di_cs_reg to the address on ca_reg_addr.

Reading from an external module works with the same principle, however the CPU core does only need to set the address to the register it will read from. The signal read_bus is only needed when reading from the data stack and the timer.

Table 5.3 gives a short description of the signals from the CPU core and the data bus.

Signals from the CPU core:		
do_cpu	8	Data out from the CPU core to the internal data bus.
databus_addr	7	Gives the address where the data on the data bus is sent to or read from.
write_bus	1	Is set high when the data on the data bus is to be written to an external module.
read_bus	1	Is set high when the CPU core is to read data from the data bus.
Signals from the internal data bus:		
di_cpu	8	The data to the CPU core.
write_stack	1	Is set high when data is to be written to the stack.
read_stack	1	Is set high when data is to be read from the stack.
di_stack	8	The data to the stack.
w_cs_reg	1	Is set high when data is to be written to the control and status register.
cs_reg_addr	4	Gives the address of what register in the control and status register to read from or written to.
di_cs_reg	8	The data to the control and status register.
w_io_reg	1	Is set high when data is to be written to the input/output register.
di_io_reg	8	The data to the input/output register.
w_timer	1	Is set high when data is to be written to the timer.
r_timer	1	Is set high when data is to be read from the timer.
timer_addr	4	Gives the address of what register in the timer that is to be read from or written to.
di_timer	8	The data to the timer.
w_irq	1	Is set high when data is to be written to the IRQ line.
di_irq	8	The data to the IRQ line.
w_io_port	1	Is set high when data is to be written to the input/output port.
io_port_addr	4	Gives the address of whet register in the input/output port that is to be rad from or written to.
di_io_port	8	The data to the input/output port.

TABLE 5.3: Signal description

APB Bus

The APB bus described in chapter 2.5 is the bus that controls the communication between the host system and the peripheral processor. The APB bus have not been implemented, but the signals from the APB bus to the modules have been named to match the APB bus. Table 5.4 describes the signals from the APB bus.

Signals from the APB bus:		
paddr	32	The address of to the internal registers of the active module.
pwrite	1	Is set high for doing a write operation and low for reading.
psel	1	Is set high activate the modules.
penable_mem	1	Is set high to enable the program memory.
penable_cs	1	Is set high to enable the control and status register.
penbale_io	1	Is set high to enable the input/output register.
penable_irq	1	Is set high to enable the interrupt request register.
pwdata	32	The data to the enabled module.

TABLE 5.4: Signal description

Data Stack

The data stack is the external storage for the CPU core and consist of the same SRAM type as the program memory and a wrapper. Writing and reading to the stack works with the LIFO (last in, first out) principle and therefore no internal address is needed when reading from and writing to the data stack.

When writing to the data stack, the stack will use its internal stack pointer to store the data in the correct location. Each time data is stored to the stack, the stack pointer will increment with one and give the position of the next storing location. When reading from the stack, the read_bus signal have to be set high. The read_bus signal will then be forwarded to the read_stack signal and the data stack will decrement the stack pointer with one.

Control and Status Registers

The control and status registers consists of eight 8-bit registers and is used to store both control and status signals. The main CPU can use these registers to control the behaviour of the program running on the peripheral processor. The peripheral processor can use these registers to store information regarding the status of it self.

How the peripheral processor and the host processor utilizes these registers will be decided by the software running on the peripheral processor and host processor. However, two bits in the first register are reserved. These two bits are the LSB (least significant bit) and the LSB+1 in the first register (cs_reg[0][1:0]). These two bits are respectively connected to cpu_reset_n and cpu_disable_n given in table 5.2.

Both the internal data bus and the APB bus can write to the control and status registers at the same time, as long as they don't write to the same internal register. If both the internal bus and the APB bus writes to the same internal register simultaneously, the APB bus will have first priority and the data from the internal bus will be lost.

Data In/Out Registers

The data in/out registers consists of one input register and one output register. The CPU core have only access to write to the output register and read from the input register. Furthermore, the host processor has only access to write to the input register and read from the output register. Reading and writing to both registers can be done simultaneously.

Table 5.5 describes the signals from the data stack, the control and status registers and the in/out registers

Signals from the data stack:		
do_stack	8	The data to the internal data bus.
Signals from the control and status register:		
do_cs_reg	8	The data to the internal data bus.
cpu_reset_n	1	When this signal is low, the CPU core will be reset.
cpu_dissable_n	1	When this signal is low, the CPU core will be set on pause.
prdata_cs	32	The data to the AMBA bus. Only the last 8 bits will contain data.
Signals from the input/output registers:		
do_is_reg	8	The data to the internal bus.
prdata_io	32	The data to the AMBA bus. Only the last 8 bits will contain data.

TABLE 5.5: Signal description

IRQ Line

The IRQ line is the interrupt request line from peripheral processor to the host system. The 1-bit value stored in this module is directly connected to the interrupt unit in the host system. If both the APB bus and the internal bus writes to the IRQ line at the same time, both values will pass through an or gate and then be stored. This is to prevent the host system to clear the interrupt the same time as the CPU core writes a new interrupt to make sure that no interrupt is lost.

Timer/Counter

The timer/counter is a 16 bit counter and does have two modes of operation, that is, compare-on and compare-of. If compare is turned off, the timer will count until it is reaching the hexadecimal number FFFF and then start over. If compare is turned on, the counter will count until it reaches the same number that is stored in the compare register. When this number is reached, a compare flag is written to the MSB of the internal register c_r and the counter starts over.

The CPU core have access to read the data in all the registers in the counter, but all the registers are divided into two and only one part of each register is possible to read at a time. This is because the internal data bus only is 8 bit wide while the

counter register and the compare register is 16 bit wide. Also, the control register is divided. This is because reading from the control bits not shall interfere with the compare flag. If the compare flag is read, the compare flag will be reset to 0. Table 5.6 gives an overview of the different registers in the counter. Table 5.7 describes the functionality of the control register.

Register description		
Register[Bit part]	Read/Write access	Internal address
counter_r[7:0]	Read only	0000
counter_r[15:8]	Read only	0001
compare_r[7:0]	Read/Write	0010
compare_r[15:8]	Read/write	0011
c_r[3:0]	Read/write	0100
c_r[7]	Read only	0101

TABLE 5.6: Description of registers in the timer

Control Register (c_r)	
Bit number	Function
7	Compare flag.
[6:3]	Unused
2	Compare on/of, 0 = compare off, 1 = compare on
1	Reset counter, reset is active low
0	Counter on/off, 0 = counter off, 1 = counter on

TABLE 5.7: Description of Control Register

Input/Output Port

The input/output port is the module that controls the signals to the input/output pad on the microcontroller. The input/output pad is the circuit that directly controls the pins on the microcontroller and the input/output port is connected to this pad.

The input/output port have four outputs to the pad and one input. The outputs to the pad is paired in two sets. One pair is pull up (PU) and pull down (PD) and the other pair is OUT and output enable (OE). Which pair to use is decided by the device that is connected to the pins. The outputs are described in table 5.9.

Writing to the different outputs in the input/output port have four different modes. These modes are clear, set, toggle and force. If the clear mode is used, the zeros in

the data from the CPU core will set a zero on the corresponding output from the input/output port. The bits in the data from the CPU core that is set to 1 will not change the output. If the set mode is used, only the ones in the data from the CPU core will set a 1 on the corresponding output from the input/output port. The bits in the data from the CPU core that is set to 0 will not change the output. If the toggle mode is used, the bits in the data from the CPU core that is set to 1 will toggle these bits on the corresponding output. The force mode will set the corresponding output to the same as the data from the CPU core.

The address bus into the input/output port is used to choose what output port to write to and what mode to use. The first two bits in the address are used to choose what output to write to, and the last two bits are used to choose what mode to use. Table 5.8 gives the address used for different modes and outputs.

Modes and output addresses		
io_port_addr	Mode	Output
xx00	clear	-
xx01	set	-
xx10	toggle	-
xx11	force	-
00xx	-	OUT
01xx	-	OE
10xx	-	PU
11xx	-	PD

TABLE 5.8: Gives the address used for different modes and outputs

Signals from the Timer/Counter:		
do_timer	8	The data to the internal data bus.
Signals from the input/output port:		
do_io_port	8	The data to the internal data bus.
OUT	8	Output to the pad
OE	8	Output enable, is set high to activate OUT signal to the pad.
PU	8	Pull Up, Used to set the output pin high.
PD	8	Pull Down, Used to set the output pin low.

TABLE 5.9: Signal description

5.3 Instructions Implemented

Table 5.10 lists all the instructions implemented including the opcode for each instruction and a description. The destination register is marked with "d", and the source register is marked with "r". "K" is marking a constant, "k" is marking an address and "x" is used for don't-care bits.

Instruction	Opcode	Description
NOP	0000 0000 0000 0000	No operation
ADC	0001 11xx dddd rrrr	Add with Carry
ADD	0000 11xx dddd rrrr	Add without Carry
AND	0010 00xx dddd rrrr	Logical AND
SBC	0000 10xx dddd rrrr	Subtract with Carry
SUB	0001 10xx dddd rrrr	Subtract without Carry
LDI	1110 KKKK dddd KKKK	Load Immediate
LSL	0000 11xx dddd dddd	Logical Shift Left
LSR	1001 010x dddd 0110	Logical Shift Right
ROL	0001 11xx dddd dddd	Rotate Left trough Carry
ROR	1001 010x dddd 0111	Rotate Right trough Carry
OR	0010 10xx dddd rrrr	Logical OR
XOR	0010 01xx dddd rrrr	Exclusive OR
CP	00001 01xx dddd rrrr	Compare
CPC	0000 01xx dddd rrrr	Compare with Carry
CPI	0011 KKKK dddd KKKK	Compare with Immediate
SUBI	0101 KKKK dddd KKKK	Subtract Immediate
SBCI	0100 KKKK dddd KKKK	Subtract Immediate with Carry
DEC	1001 010x dddd 1010	Decrement
INC	1001 010x dddd 0011	Increment
MOV	0010 11xx dddd rrrr	Copy Register
POP	1001 000x dddd 1111	Pop Register from Stack
PUSH	1001 001x dddd 1111	Push Register to Stack
STS	1010 1kkk dddd kkkk	Store Direct to Data Space
LDS	1010 0kkk dddd kkkk	Load Direct from Data Space
BRCS	1111 00kk kkkk k000	Branch if Carry Set
BRCC	1111 01kk kkkk k000	Branch if Carry Cleared
BREQ	1111 00kk kkkk k001	Branch if Equal
BRNE	1111 01kk kkkk k001	Branch if Not Equal
JMP	1001 010x xxxx 110x kkkk kkkk kkkk kkkk	Jump Jump address

Table 5.10: Included instructions

5.4 Simulation

The waveforms of the simulations is given in appendix A and the program code for each simulation is given in appendix B. Appendix A.1 shows a check of all the instructions when running the program in appendix B.1. Table 5.10 can be used together with the waveforms to verify that all the instructions are executed correctly.

Appendix A.2 displays the result of the UART simulation. The program for this simulation is given in B.2 Figure A.5 shows the part of the simulation where the program is waiting for an input from APB bus. The data in control register 7 is used to check for new data in the input register. The data in the input register is loaded into register 7 in the register file. The next figure, figure A.6 shows the part of the program where the parity bit is calculated. Register 7 is used to calculate the parity bit and the parity bit is stored in the LSB in register 6. Figure A.7 shows the that the same value loaded from the input register is shifted out. This includes a start bit, the data, a parity bit and the stop bit. Each bit is set to be held on the output port for two clock cycles.

The last figure, figure A.8, in appendix A.2 shows the serial-to-parallel conversion part of the UART program. The program is waiting for a start bit on the input port and is then shifting in the data from this port into register 4. This data is then written to the output register and the interrupt request to the host system is set. No parity bit is included in this part.

5.5 Synthesis Reports

Table 5.11, 5.12, 5.13 and 5.14 gives the relevant reports printed by the synthesis tool.

Area	
Number of ports:	199
Number of nets:	334
Number of cells:	75
Number of combinational cells:	67
Number of sequential cells:	0
Number of macros:	0
Number of buf/inv:	65
Number of references: 65	12
Combinational area:	7755 (72.5%)
Noncombinational area:	2935 (27.5%)
Total cell area:	10690 (100%)

Table 5.11: Area report

Setup timing	
Startpoint:	C1/PM/U_SRAM_MEM (rising edge-triggered flip-flop clocked by clk')
Endpoint:	C1/A1/z_r_reg (rising edge-triggered flip-flop clocked by clk)
Path Group:	clk
Path Type:	max
Data required time	23.51
Data arrival time	-23.51
slack (MET)	0.00
Clock period	24.00

Table 5.12: Setup timing report

Clock Gating	
Number of Clock gating elements	33
Number of Gated registers	272 (93.15%)
Number of Ungated registers	20 (6.85%)
Total number of registers	292

Table 5.13: Clock Gating report

Uncollapsed Stuck Fault Summary Report		
Fault Class	Code	#Faults
Detected	DT	20569
Possibly detected	PT	37
Undetectable	UD	543
ATPG untestable	AU	468
Not detected	ND	469
total faults		22086
test coverage		95.56%

Table 5.14: Design For Test Coverage

5.6 Net List Reports

Table 5.15, 5.16 and 5.17 gives the relevant reports printed by the clock tree synthesis tool.

Setup Timing	
Startpoint:	rst_n
Endpoint:	C1/R1/reg_r_reg_1_5
Path Group:	async default
Path Type:	max
Data required time	25.03
Data arrival time	-3.37
slack (MET)	21.66
Startpoint:	C1/PM/U_SRAM_MEM
Endpoint:	C1/R1/clk_gate_reg_r_reg_5_/latch
Path Group:	clock gating default
Path Type:	max
Data required time	23.16
Data arrival time	-24.87
slack (VIOLATED)	-1.70
Startpoint:	write_mem (input port)
Endpoint:	STACK1/U_SRAM_STACK
Path Group:	REGIN
Path Type:	max
Data required time	12.53
Data arrival time	-10.94
slack (MET)	1.59
Startpoint:	C1/PM/U_SRAM_MEM
Endpoint:	C1/R1/reg_r_reg_1_6
Path Group:	clk
Path Type:	max
Data required time	24.62
Data arrival time	-26.12
slack (VIOLATED)	-1.50

Table 5.15: Setup Timing report

Hold Timing	
Startpoint:	rst_n (input port)
Endpoint:	TIMER/counter_r_reg_15
Path Group:	async default
Path Type:	min
Data required time	0.61
Data arrival time	-0.68
slack (MET)	0.07
Startpoint:	penable_io (input port)
Endpoint:	IO1/clk_gate_data_in_r_reg/latch
Path Group:	clock gating default
Path Type:	min
Data required time	0.17
Data arrival time	-0.21
slack (MET)	0.04
Startpoint:	pwdata[3] (input port)
Endpoint:	IO1/data_in_r_reg_3
Path Group:	REGIN
Path Type:	min
Data required time	0.40
Data arrival time	-0.40
slack (VIOLATED)	0.00
Startpoint:	C1/B1/jmp_ctrl_reg
Endpoint:	C1/B1/jmp_ctrl_reg
Path Group:	clk
Path Type:	max
Data required time	0.42
Data arrival time	-0.72
slack (MET)	0.29

Table 5.16: Hold Timing report

Power		
Net Switching Power	9.433e-04W	58.13%
Cell Internal Power	6.796e-04W	41.87%
Cell Leakage Power	1.150e-08W	0.00%
Total Power	1.623e-03W	100.00%
Total Power per MHz	3.892e-05	

Table 5.17: Time Based Power

Chapter 6

Discussion

6.1 Instruction Set

In the preliminary work it was given two solutions to an optimized instruction set. This was done because the peripheral processor do not need as many instructions as the 8-bit AVR. Due to that, the opcode for the instructions can be simplified. The best solution was found to be a combination of both 8-bit instructions and 16-bit instructions. However, it has been chosen to use the same opcode layout for the instructions in the peripheral processor as the instructions in the 8-bit AVR. The reason for this is that the same compiler can then be used for the peripheral processor and the 8-bit AVR. If a new instruction set were to be used, a compiler would have to be written for it.

A negative effect of this is that it will be parts of the opcode that is not needed when giving instructions to the peripheral processor. The unused parts are the bits marked as don't-care bits in table 5.10.

6.2 CPU Core

Pipelining

The CPU core was primarily implemented by using the theory given in chapter 2.2, but several decisions had to be made during the implementation of the different modules. First of all, the processor does only have two steps in the pipeline. Instruction fetch and execute are in the same step and the last step is write back. As a consequence of this, the ALU and instruction decoder had to be fully combinatorial. Also, the output from the register file is combinatorial. The main reason for this choice is that a processor with few pipelined steps is more easy to implement and to debug.

However, there are some disadvantages with this solution. One disadvantage is that the clock frequency have to be lower than in a processor with more pipelining. Additionally, if the results show that the processor is too slow it will be complicated to add a more pipelined structure without changing the whole processor.

Negative Numbers

Most processors have the functionality to support negative numbers. It have been chosen that this peripheral processor will not have that functionality. This is because this processor is made for doing parallel-to-serial and serial-to-parallel conversion and be able to run the different protocols that handle this type of data conversion. Most protocols that do this, do not need negative numbers, therefore the support for negative numbers have not been included. An advantage of this is that the processor can handle bigger positive numbers when the support for negative numbers is excluded.

Shared Program Memory and Data Stack

In the preliminary work it was suggested that the program memory and data stack would share the same SRAM. This was suggested because the size of the data stack

can be much smaller than the program memory. Therefore only one SRAM module is needed if the data stack is implemented as a small part of the program memory. During the implementation it was found that a shared program memory and data stack was difficult to implement and it would also decrease the performance of the processor.

The reason for the implementation difficulties was that it is only possible to do one read or one write operation at a time to the SRAM. Therefore, it is impossible to read the next instruction from the program memory if the data stack is active. One solution to solve this problem was to include an instruction buffer in the decoder that contained a certain amount of instructions. This buffer was then used to fetch the next instruction when PUSH/POP instructions to the data stack was executed. However, both the branch control and the instruction decoder was already implemented and this solution needed huge changes in both modules. Furthermore, it was several other issues with this solution and it was uncertain that it would give the required performance.

Because of these reasons, it was chosen to have separate program memory and data stack. This will increase the cost and the area of the processor, but the performance would be better because of the possibility to read the program memory at the same time as the data stack is active.

PUSH and POP Instructions

When the data stack was implemented, it was found a problem when reading and writing to it. This problem did only occur when the PUSH and POP instruction was stored in the second half of a 32 bit program memory location (bit 0 to 15). In this situation the data stack did not respond. This was happening because reading and writing to an SRAM module have a delay and the delay when reading the program memory was added to the delay in the data stack. This total delay was longer than a half clock cycle and the data stack could therefore not respond before a new clock

cycle was reached. The half clock cycle time limit is because of the inverted clock in the SRAM.

It was thought of a couple of different solutions for this problem, but it was found that the best solution was to extend the PUSH and POP instructions to last for two clock cycles. To do this, the hold_pc and done_hold signals was added between the instruction decoder and the branch control. If a PUSH or POP instruction is given, the instruction decoder will set the hold_pc signal high. The next clock cycle the branch control will set the done_hold signal high and then the PUSH or POP instruction will be executed. The operation in the first clock cycle will then only consist of loading the instruction into the instruction register. The next clock cycle, the instruction register will already have the instruction stored and the program memory is not needed to be read. The second clock cycle, the PUSH or POP instruction can be executed without the delay from the program memory.

The disadvantage with this solution is that the PUSH and POP instructions will require twice as long time to be executed and the performance of the processor will therefore be decreased.

JUMP Instruction

The branch instructions do a relative branch in the program memory form the current position in both positive and negative directions. However, these instructions can not be used to branch more than 64 steps in either direction. Therefore a JUMP instruction is needed to be able to reach the whole program memory. This instruction is different from the other instructions because it is a 32 bit instruction. The first 16 bits is used for the opcode of the instructions while the last 16 bits are the jump address.

Because the JUMP instruction was divided into two 16 bit parts, a problem occurred when this instruction was implemented. If the jump address was the same as the opcode for an instruction, the instruction decoder would decode the address as if it was an instruction. This is the reason Mux 2 in figure 5.1 is implemented. If a jump is

called, the branch control will set the msd signal in mux 2 to give a NOP instruction to the instruction decoder while the branch control is executing the jump.

6.3 Exteral Modules

Internal Data Bus

The internal data bus is designed to be both fast and have low power consumption. It is fully combinatorial and the delay in the internal data bus will therefore only depend on the amount of logic the data have to pass from the input to the output. Because of the bus design, the data have to only pass one multiplexer or demultiplexer to go from the input to the output. The longest data path in the bus was originally given in the report presented in table 5.15, but the complete data path had to be removed due to corporate secrets at Atmel. However, this report showed that the longest data path in the internal data bus would use about 2 ns from the input to the output. This is not much compared to the 24 ns long clock period.

The power consumption will also be low in the internal data bus. This is because the bus does not have any clock and is only active when it is triggered by the CPU core. Additionally, there is only the relevant outputs that will toggle when the bus is active. The inputs to the unused modules will be held low during a transmission.

Data Stack

The data stack was previously defined to be a part of the same SRAM used by the program memory. But because of the problems discussed in chapter 6.2 it was chosen to have a separate SRAM module for the data stack. The result of this is that the cost of the processor will increase because an extra SRAM module have to be implemented and the SRAM module implemented is much bigger than what is required for the data stack. Nevertheless, it is easy to change the implemented

SRAM and that is something Atmel may do if they chose to do further work with the peripheral processor.

Control and Status Registers

The decision to have eight control and status registers was taken in the preliminary work. The software for a USART will typically only need five control and status registers and SPI require even less, but the peripheral processor is supposed to run other protocols as well, for example USB. Therefore as much as eight control and status register are needed. The programmer does also have the alternative to use the unused control and status registers as a storage place. The unused registers will therefore not be wasted.

The cpu_reset_n and cpu_disable_n signals are added to give the host processor more control of the peripheral processor. First of all, the cpu_reset_n signal have to be used when the host processor is programming the peripheral processor. This will reset the program counter and hold it so that the program memory can be written to without the peripheral processor is trying to read the program memory. The cpu_disable_n signal is to pause the peripheral processor. If this signal is active, the peripheral processor will do nothing, and will not be reset. The peripheral processor will continue from its current state when cpu_disable_n is not active any more.

Data In/Out Registers

The data in/out registers could have been implemented as a part of the control and status registers and let the software running on the peripheral processor decide what registers to use for data and what registers to use for control and status bits. However, having a separate module controlling the data transfer to and from the peripheral processor would make it more easy for the programmer to keep track of the different registers.

The data in/out registers are implemented to behave as only one register. The peripheral processor can only read the input register and only write to the output register. Therefore, no internal addressing is required when reading and writing to the in/out registers. This will prevent the peripheral processor to overwrite input data and the host processor cannot overwrite output data from the peripheral processor.

Timer/Counter

It was experienced that a 16-bit counter was necessary when writing the software to emulate an UART in the preliminary work. The timer/counter have therefore been implemented with 16 bits. The functionality of the counter have been limited to only include the most basic functions for a counter. This is because a peripheral processor should be as simple as possible and the functions implemented was only the functions used in the preliminary work.

One argument to implement a more advanced counter is because the processor should be able to run other protocols than those investigated in the preliminary work. However, all the extra functionality a counter can have, can also be made with software.

IRQ Line

Both the peripheral processor and the host processor have access to write to the IRQ line. This is to make the peripheral processor able to set an interrupt request and the host processor is able to clear an interrupt request. It is no limitation for any of the two processors to do both reading and writing to the IRQ line. Therefore the programmer have to be careful to not write a wrong bit to the IRQ line.

Input/Output Port

The input/output port was implemented to give the peripheral processor several methods to write data to the output port. The clear, set, toggle and force modes

can be used in different situations where different writing modes are necessary. The different modes and outputs of the output port was the reason for needing four bits to address the internal registers of the external modules. However, the extra functionality to the output port gave a bigger advantage than having one less bit in the address of the internal data bus.

6.4 Simulation

Test of Instructions

The waveforms in appendix A.1 shows the result of running the program given in appendix B.1. This program does not do anything useful other than showing that each separate instruction is doing what it is supposed to do. Figure A.1 shows that the processor is able to branch and jump. The red line in the figure shows the point where the first branch instruction is executed. The opcode is f42a which indicates a branch-if-carry-set instruction with a distance of 5 steps in the program counter. One can see that the carry bit is set and the program counter jumps from 31 to 36 which is a distance of 5.

By going through all the other instructions given in appendix A.1 together with the program given in appendix B.1, one can confirm that all the instructions are doing what they are supposed to do and that all the external modules are responding correct when they are accessed.

UART Test Program

The waveforms in appendix A.2 shows the result of running the program given in appendix B.2. This program was made to test if the processor was able to run a program that did the most basic operations of what is expected by an UART. The operation done by the program is to use the status register to check for new data in the data in register. If new data is found, the processor will first calculate a parity

bit for the data and then shift the data out with the same frame format the UART protocol is using. The program will also check for a start bit on the input port and shift data in if a start bit is recognized.

Figure A.5 shows the first part of the program where the status register is checked. The first red line indicates the time the host processor writes to the status register to indicate that new data is written to the data in register. The time between the first and second line is the part where the status register is analysed. At the second line, it has been registered that new data is available and the peripheral processor does a jump to the function for calculating parity bit. The third line indicates the time the data in the data in register is loaded into the internal register of the peripheral processor and the calculation of the parity bit is started.

The third line in figure A.5 is the same line as the first in figure A.6. Register 6, 7 and 13 is used for the parity bit calculation and the calculation procedure is given with comments in appendix B.2. Register 7 is used to load the data from the data in register and register 6 is the register that stores the parity bit. If register 6 ends up containing an even number the parity bit is 1 and opposite if register 6 ends up containing a odd number. Bit 0 in register 6 can therefore be used as a parity bit. The second line in figure A.6 indicates the time where the peripheral processor has calculated the number of ones in register 7. At this time register 6 contains the hexadecimal number 5 and the data in the data in register is the hexadecimal number 6d that is equivalent to the binary number 01101101. This binary number have 5 ones and confirms that the calculation is correct. The data in register 6 is then incremented with one so that bit 0 in register 6 can be used as the parity bit.

The third line in figure A.6 is the same as the first line in figure A.7. This line indicates the start of the shift out procedure. First the data in the data in register is loaded into register 0. Then the status register is updated so that the host processor can write new data to the data in register. After that, the shifting out starts. First a start bit is loaded to the output followed by the data in register 0. The first bit to be shifted out is the LSB and the data is set to be held for two clock cycles before it shifts to the next bit. Last, the parity bit is loaded to the output after the MSB in

register 0. The parity bit is followed by a stop bit and the output will be held high until the next shift out procedure. The last red line on figure A.7 indicates the end of the shift out procedure and the peripheral processor jumps back to the start to check for new inputs.

Figure A.8 shows a situation where the peripheral processor recognizes a start bit on the input port. A jump to the shift-in procedure is done at the time given by the first red line. The data is shifted in with the LSB first. At the second red line the shifting is done and the data in register 4 is loaded into the data out register. The next clock cycle an interrupt request is sent to the host processor to indicate that new data is available. The last red line indicates the time the processor acknowledges the interrupt and resets the interrupt request.

It is important to notice that this program does not behave as a valid UART. It is only made to test and demonstrate that the processor is capable to do the most basic operations required by the UART protocol and other serial-to-parallel conversion protocols. However it gives a good indication that the peripheral processor is capable to emulate a fully functional SPI or UART module. This is because most of the operations in these protocols are reading and writing to the status registers and doing fault checking on received data in addition to shifting data in and out. All of these operations are demonstrated in the UART test program.

6.5 Performance

The performance of the peripheral processor is decided by two elements: The clock frequency and how fast the processor can do operations frequently required by the program running on the processor. For example, a processor that is frequently required to do multiplications, but do not have a multiplication instruction, will have to utilize the other instructions in its instruction set to do multiplications. The result of this is that one multiplication procedure will require several clock cycles to be performed and the overall performance of the processor will decline in accordance with how many multiplications operations needed.

Synthesis Timing

The timing report from the synthesis tool given in table 5.12 is giving a good indication of what the clock period have to be. The timing report presents the longest path in the design, how fast the data is required to pass through the longest path and how fast the data is passing trough the longest path. With a clock period of 24 ns The data arrival time and the required time was the same (23.51 ns). This resulted that the clock period could not be shorter than 24 ns and gives a clock frequency at 41.7 MHz.

The start point in the longest path was the SRAM in the program memory. Because of the lack of pipelining, it was chosen to invert the clock in the SRAM. This would let the SRAM respond within the same clock cycle it is accessed. However, this had a huge affect on the timing in the processor. Because the start point of the longest path was the SRAM and this was clocked on the negative clock edge, the data would leave the SRAM first after one half clock period (12 ns). The consequence of this is that the first half of the clock period is lost and the current longest path is 12 ns longer than actually necessary. If an extra instruction-fetch step in the pipeline had been included the inverting of the clock in the program memory would not have been necessary and the longest data path could potentially have been 12 ns shorter.

Because the peripheral processor is connected to the host system with an APB bus, it is restricted to the clock frequency of the bus, which again is restricted to the clock frequency of the host processor. The maximum frequency of the peripheral processor can therefore not be higher than the clock frequency of the host processor. However, it is possible to implement handshaking between the control and status registers in the peripheral processor and the host processor. This will allow the peripheral processor to have an asynchronous clock that can be faster than the clock in the host processor.

Netlist Timing

As described in chapter 4.3, the timing reports presented after the clock three synthesis was more detailed than the earlier timing report. The setup and hold timing

reports are the result of four different tests done by the clock three synthesis tool. The tests are divided into four different groups: async default, clock gating default, regin and clk. The async default test checks the longest path for an input signal. The clock gating default test checks the longest path for a signal to reach the input to a clock gate. The regin test checks the longest path for an internal signal to reach a register. Last, the clk test checks the longest path for the clock input. The numbers in the tests are given in nano seconds.

The first report, named Setup Timing, gives the results for a test that checks if the data signal reaches its end point before the clock signal at a worst case scenario. The two tests that belong to the path group async default and regin are both passed. In both tests, the data arrives before the clock and no problem will arise in these situations. However, the two tests that belong to the path group clock gating default and clk was not passed. In these situations the signal will arrive later than required and this can cause the peripheral processor to malfunction. However these tests checks the peripheral processor at a worst case scenario. For example at high temperatures. Therefore, the peripheral processor will probably function correctly in normal circumstances.

Nevertheless, it is optimal if all the tests is passed. The easy solution is to decrease the clock frequency, but there are also other methods that can be utilized. As mentioned in chapter 6.3, the complete data path is removed from the timing reports. However, the original report shows that having a clocked instruction register between the program memory and instruction decoder as an extra pipelined step could save up to 4 ns. The result would then have been that both the violated tests in table 5.15 was passed. A third method to correct the violated tests is to manually make changes to the data path so the data signal will arrive earlier. This process require a lot of experience and takes long time and is therefore not desirable.

The second report, named Hold Timing, gives the result for a test that checks if the data is held stable on a register long enough after the clock arrives. If these tests are violated, a register can store an incorrect value and the peripheral processor can malfunction. These tests are more critical and have to pass. The report in table 5.16

shows that all the tests are passed except the test in the regin group. The report shows that the data required time is the same as the data arrival time but because of the inaccuracy with only two decimals the test tool assumes that the test is violated. If there is a timing violation in this test, a solution to the violation can be to manually add a buffer between the flip-flops in the current register.

Speed of Operations

As discussed in the start of this chapter, it is not only the clock frequency that defines the performance of the processor. How many clock cycles the processor need to perform an operation is also essential. This is decided by the instructions available and how a program is written. The test program given in appendix B with its simulation results given in appendix A.2 is not optimal for doing a performance analysis, but can be used to give an estimation of the performance.

One can see from figure A.5 that it is 14 clock cycles between the first and the second red line. This is the time from the status register is updated to the parity calculation is started. The first and the last line in figure A.6 shows that the parity calculation require 36 clock cycles. These two numbers indicates that approximately 50 clock cycles are required from the time new data is available to be transferred to the data is ready to be transferred. The UART protocol is defined to be able to transfer a new data frame immediately after the stop bit at the end of the frame. Therefore, the peripheral processor have to be capable of checking for new data and calculate parity bit within the period of one stop bit. If the stop bit is set to be the same size as the rest of the bits, only one bit can be transferred each fiftieth clock cycle. Because the clock period is set to be 24 ns, the maximum baud rate will be 833Kbit/s as given in equation 6.1. The typical baud rate of a UART range from 2.4Kbit/s to 1Mbit/s [2].

$$BaudRate = \frac{1s}{(24 * 50)ns} = 833Kbit/s \tag{6.1}$$

In this case, the parity bit was calculated separately. It could have been calculated while shifting out to improve the performance. Nevertheless, this is the absolute

maximum if the peripheral processor is only to transmit data and no data reception is to be done. If data reception is done available simultaneously as data reception, the baud rate will decrease significantly. This is because the UART receiver includes fault checking when receiving data as given in chapter 2.4. This fault checking will require several clock cycles within each sample and the baud rate will therefore decrease. How much the baud rate will decrease is hard to estimate without having a fully functional UART program to run on the peripheral processor.

Under optimal circumstances, a fully functional UART, SPI and TWI program should have been programmed to run on the peripheral processor to be able to investigate the performance. Nevertheless, this was not possible because of the time limit.

Because this processor is only to be used as a peripheral processor, specialised instructions could be added to reduce the required clock cycles needed to do operations that are frequently requested. For example, a calculate-parity-bit instruction can be added so that the parity bit calculation can be done in only one clock cycle. This would increase the area of the ALU, but at the same time probably increase the maximum baud rate for a UART transmission. If fully functional UART, SPI and TWI programs had been made, these programs could be used to analyse what operations that would be beneficial to include as separate instructions.

Power

When synthesising, the synthesis tool will add clock gates in front of registers that are activated with equal conditions to save power. Adding a clock gate is beneficial if three or more registers are activated by the same condition. The report presented in table 5.13 show that more than 93% of all the registers are clock gated. This is a good result and it will be hard so save more power by adding additional clock gates.

The power report given in table 5.17 is the result of a power analysis done when running the UART program in appendix B.2. It is hard to compare this value to other circuits, but it can be calculated how long a normal battery can power the peripheral processor when it is running the tested program.

The ATmega128 microcontroller can operate at a voltage ranging from 2.7V to 5.5V. It is therefore reasonable to assume that the peripheral processor can operate at 3V that is the same as two AAA batteries connected serially. A typical AAA battery can have a capacity of 750mAh. At 3V the peripheral processor will have a current consumption of $1.623mW/3V = 0.541mA$. The peripheral processor can therefore run the UART program for $750/0.541 = 1386$ hours. This is obviously an unrealistic value because the peripheral processor cannot function without the rest of the AVR. Nevertheless, it gives a number to associate with the power consumption.

Area and Cost

Table 5.11 presents the total area of the peripheral processor without the SRAM. This is because the SRAM modules implemented are only implemented to make testing of the peripheral processor possible. The SRAM modules can easily be changed without having to do fundamental changes in the architecture of the peripheral processor.

The area is given in NAND2 equivalents and presents both the combinatorial and noncombinatorial area. Because of the lack of pipelining, it is reasonable that the combinatorial area is 72.5% of the total area. If extra pipelined steps is to be added, the combinatorial area will presumably be unchanged, but the noncombinatorial area will increase.

The total area for the peripheral processor is reasonable compared to the estimation given in the preliminary work in table 3.3. The extra area is the result of a combination of the unreliability in the area calculations and extra logic added during the implementation of the peripheral processor.

The total area given in table 5.11 consists of 10690 NAND2 gates. The total area of the non programmable USART, SPI and TWI modules is 3200 NAND2 gates all together, as given in chapter 3.5. The peripheral processor is therefore more than 3 times bigger and more expensive than these thee modules together.

In addition to this the SRAM have to be included as well. The area and cost of the SRAM will vary depending of what SRAM is chosen and the size of it. The

SRAM size and type is a choice Atmel have to take if they decide to continue the development of the peripheral processor.

Design For Test Coverage

Table 5.14 presents the DFT coverage of the peripheral processor. This result should be as high as possible and preferably above 98%. The 98% limit is a requirement for units to be qualified to be used in automotive areas, such as cars, trains and other industrial elements. It would therefore be desirable to have a DFT coverage above 98%. However, one can argue that the peripheral processor is only a small part of a huge system. Therefore, given that the rest of the system is much larger than this module, a test coverage of 95.56% will not have a significant impact of the DFT coverage of the whole system.

The reason that the result is below 98% is, among others, because of some combinatorial logic on the inputs and outputs that not is testable. However, these problems will disappear if the peripheral processor is included into the host system and then tested. The synthesis tool do not give a detailed DFT coverage report, and it would be easier to analyse the DFT coverage if an ATPG tool had been used after the synthesis. This is not done because of the time limit.

Chapter 7

Conclusions

In the preliminary work it was found that a processor is significantly slower than non programmable hardware. Therefore utilising a processor instead of non programmable hardware to control serial communication with external devices would have a huge affect on the maximum baud rate. The result from the implementation confirms this, but the affect on the baud rate can be much smaller than first anticipated. This is because a peripheral processor can have a higher clock frequency than the host system and it is easier to optimize the program when the instruction composition is programmed manually instead of using a high level programming language, such as C.

The peripheral processor implemented can have a maximum clock frequency of 41.7 MHz. This is almost four times higher than the clock frequency used in the preliminary work that was 11.1 MHz. In addition to this, the implementation is done with only two pipelined steps. This is not an optimal solution if high clock frequency is desired and was only chosen to simplify the design of the peripheral processor due to the time limit. The timing reports from the synthesis tools show that the clock frequency can be doubled if a more pipelined structure is chosen. Furthermore, several small adjustments can be made to further increase the clock frequency. The baud rate given in equation 6.1 is unrealistically high because of the lack of functionality in the UART program. However, if the clock frequency is increased to around 80 MHz a baud rate of around 0.5Mbit/s may be possible. Nevertheless, a fully functional

UART program should be made and tested on the peripheral processor if further work is to be done with it. A fully functional USART program will give a good feedback about the maximum baud rate, also at different clock frequencies.

Another problem found in the preliminary work was the area of the peripheral processor. This problem is still significant. The fact that the peripheral processor is more than three times larger than the total area of the non programmable USART, SPI and TWI modules together cannot be ignored. The area of the SRAM will also be added to this. For that reason, the cost of the peripheral processor have to be valued against the advantages of its ability to be reprogrammed.

The value of having a reprogrammable peripheral processor is huge if the existing protocols are being updated and if new communication protocols are released after the microcontroller have been produced. A microcontroller with a programmable peripheral processor will then last much longer than a microcontroller with non-programmable peripherals.

It is hard to give a sturdy conclusion if a peripheral processor is a good solution or not. A peripheral processor will both have significant advantages and disadvantages. However, this thesis illustrates that it is possible to substitute non-programmable peripherals with a programmable peripheral processor.

Appendix A

Simulation results

A.1 Simulation of instructions

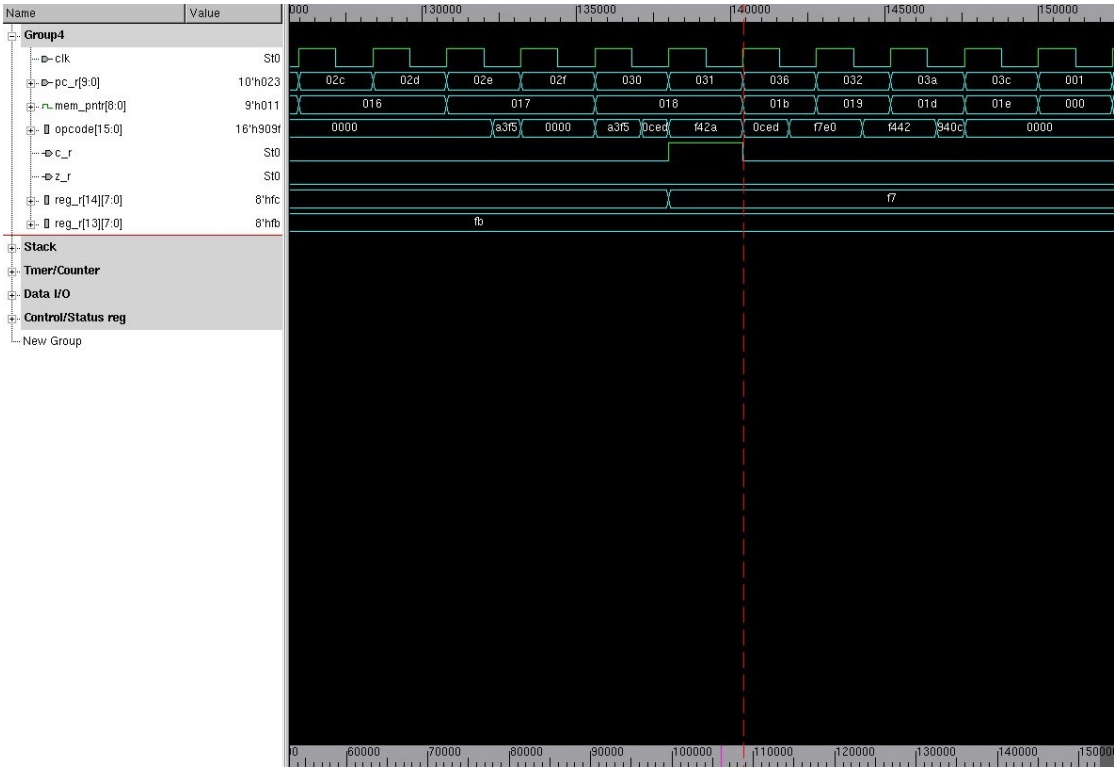

FIGURE A.1: Branch and jump waveform

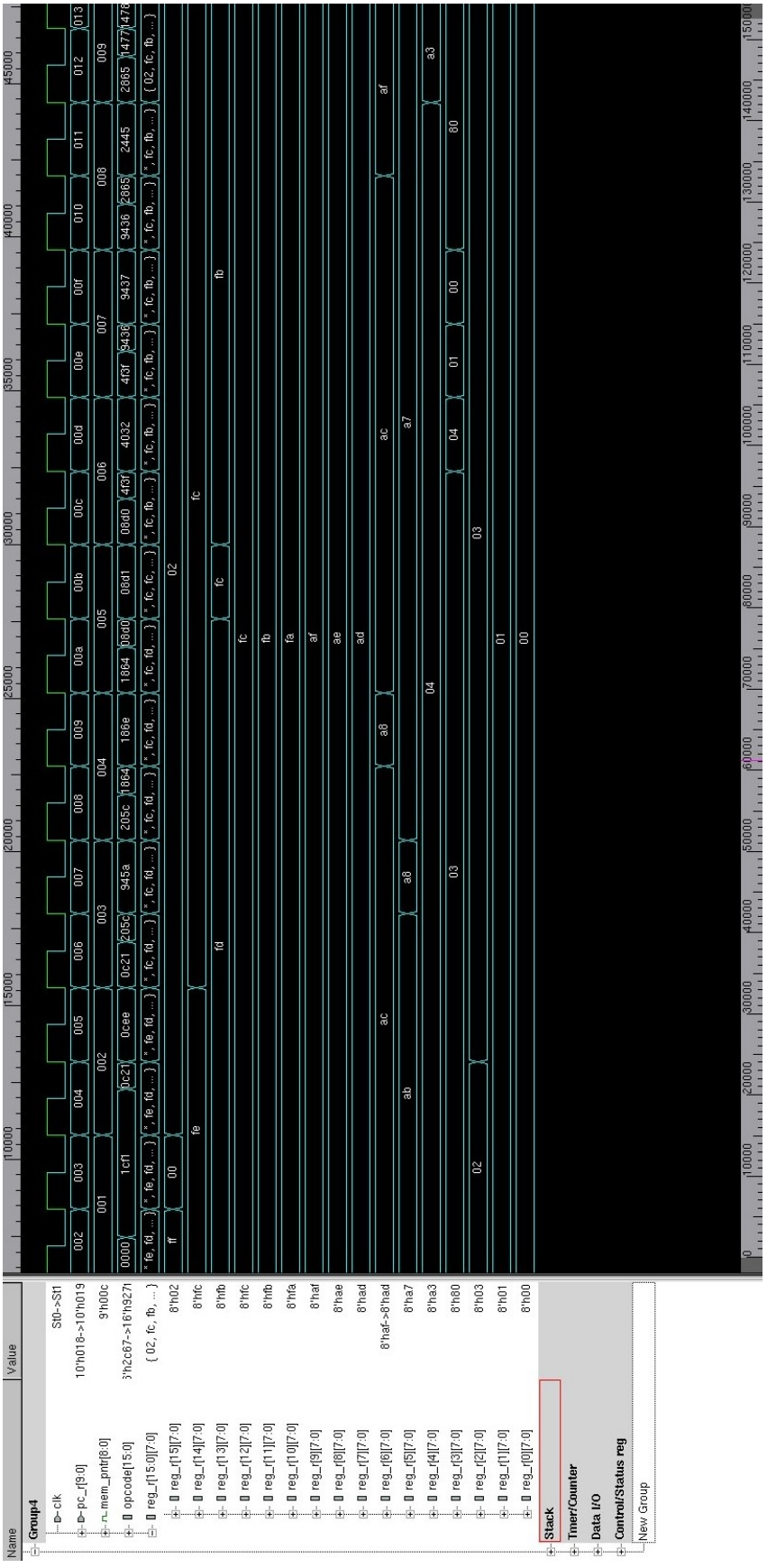

FIGURE A.2: Register operations waveform

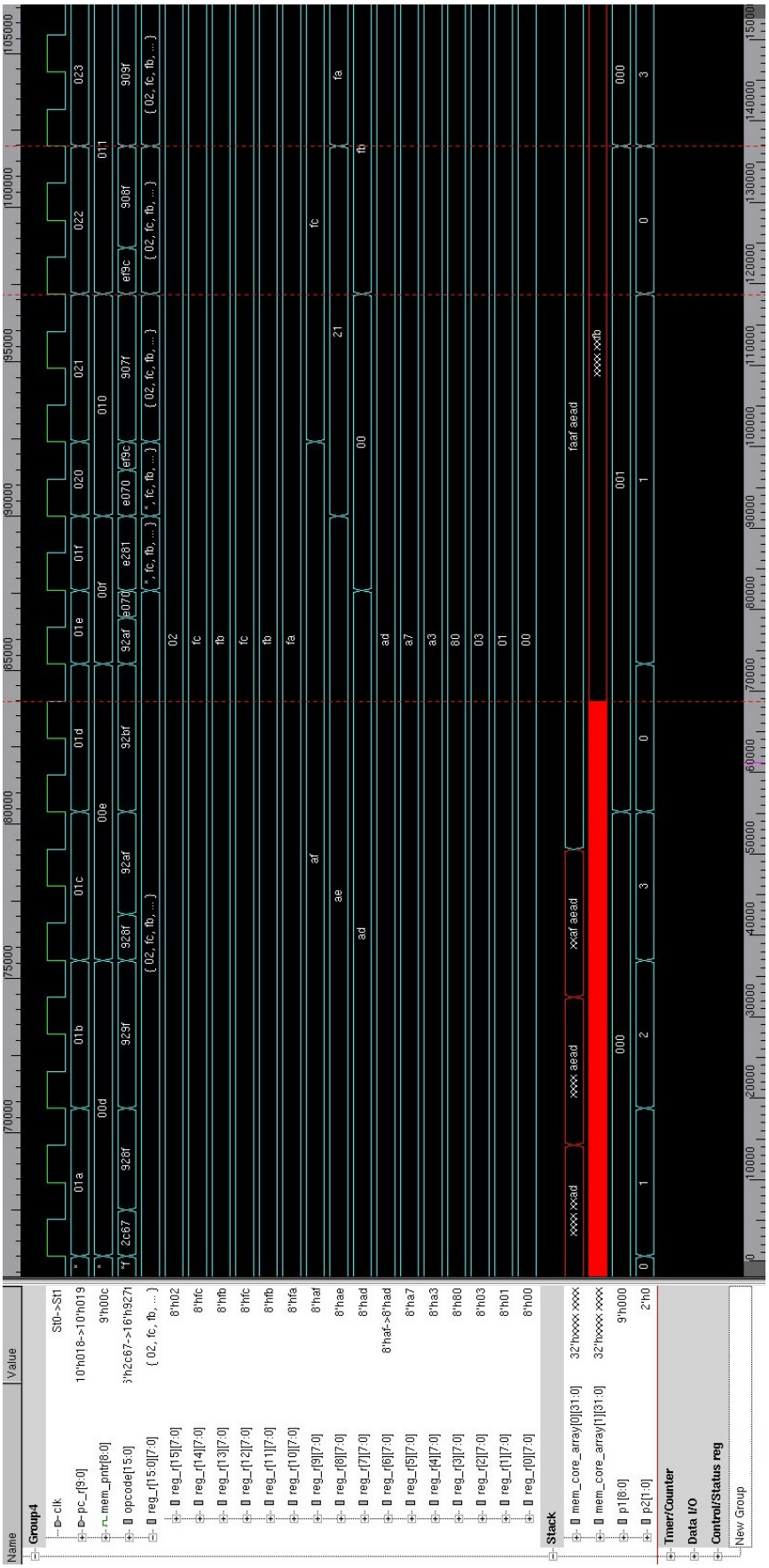

FIGURE A.3: Stack operations waveform

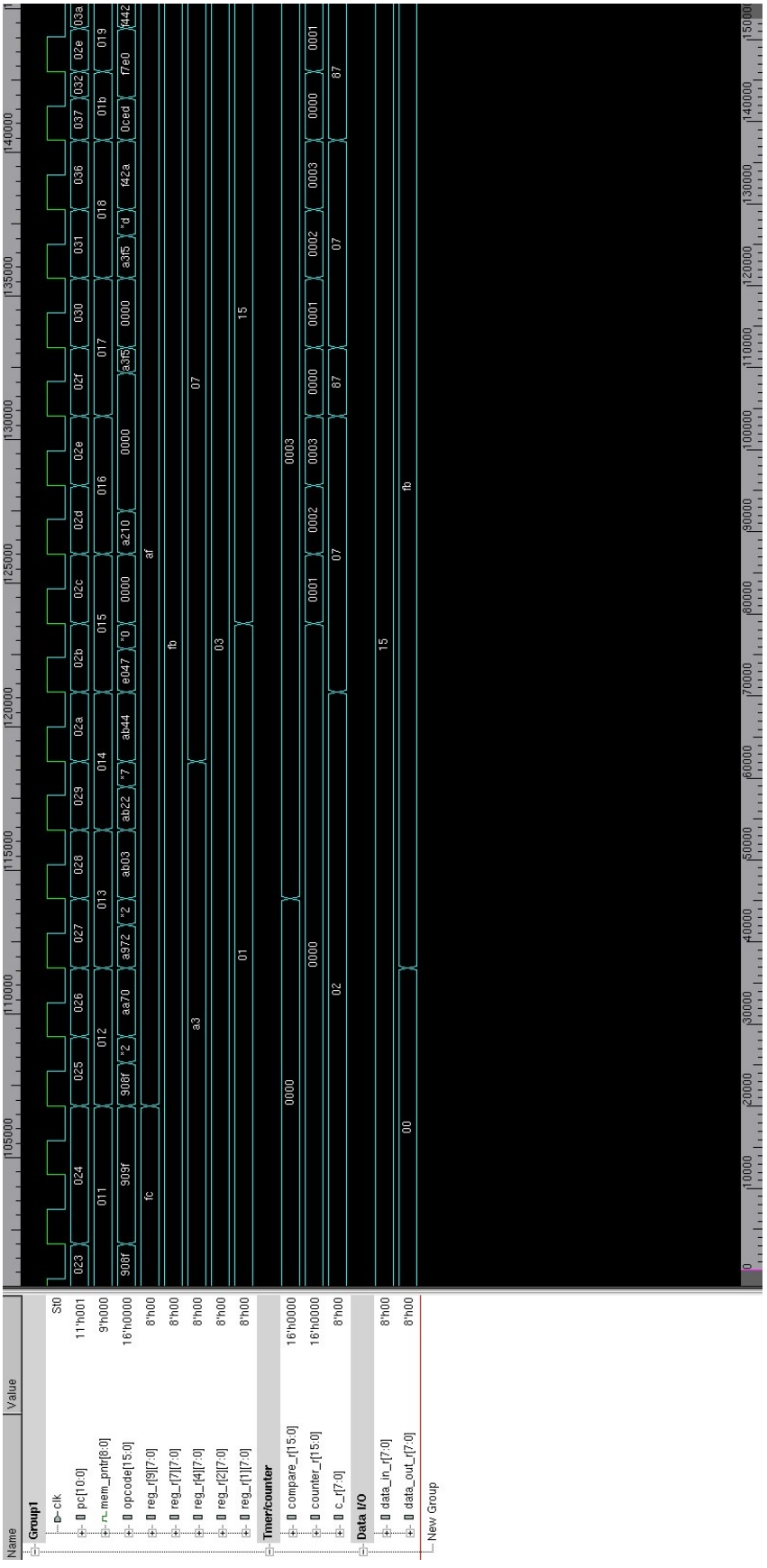

FIGURE A.4: In/out and timer operations waveform

A.2 UART simulation

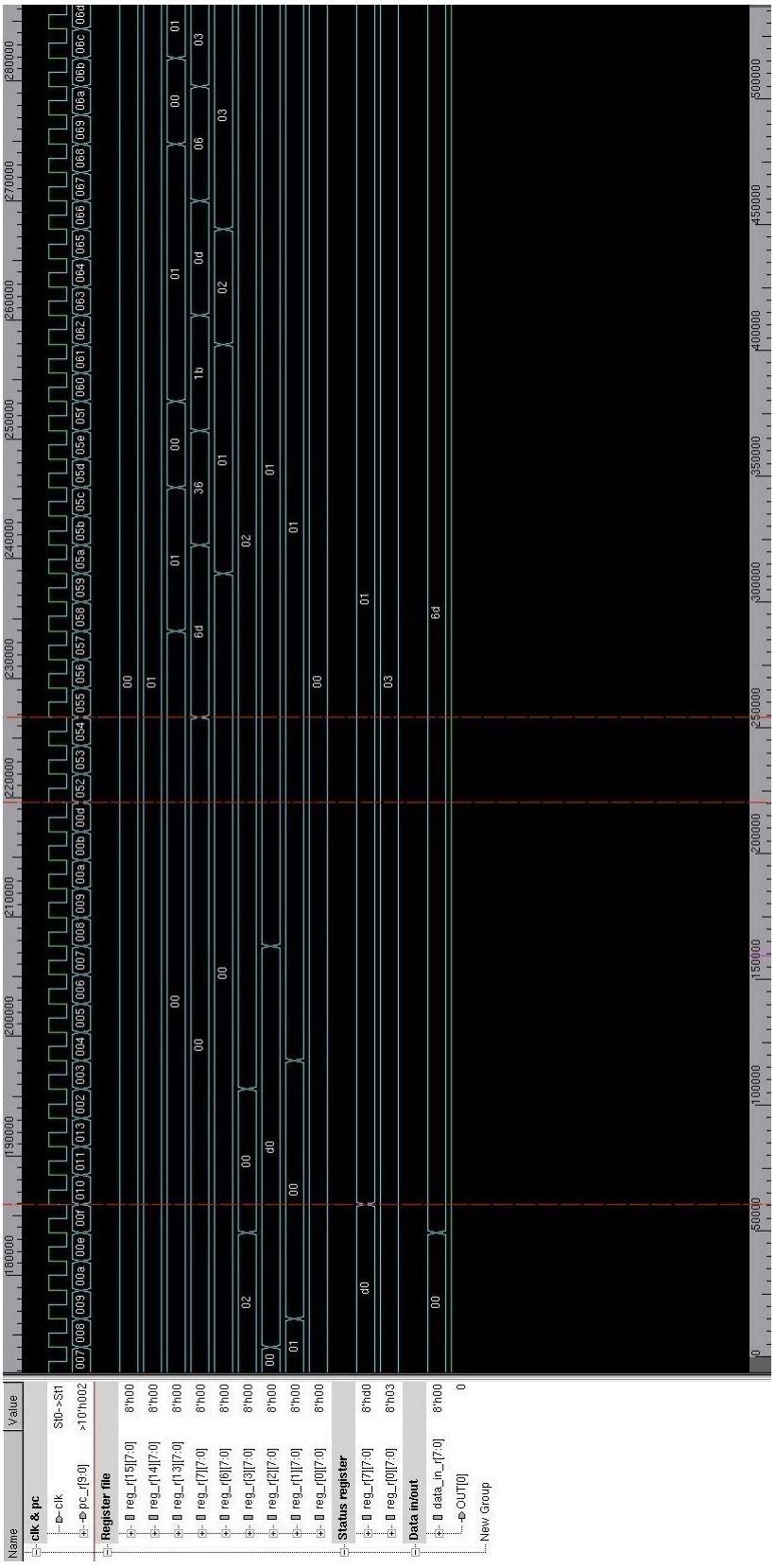

FIGURE A.5: Checking for status bit

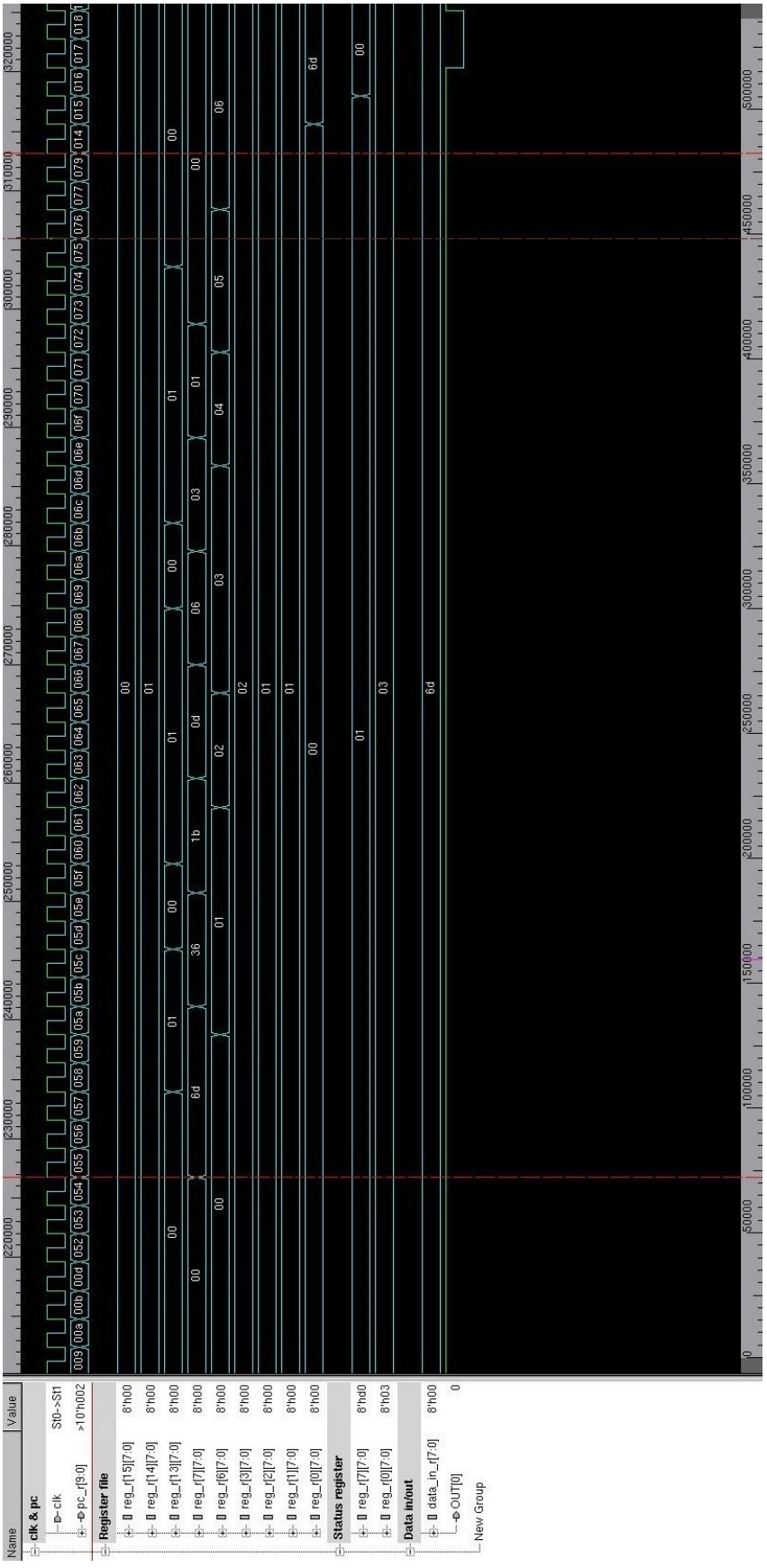

FIGURE A.6: Calculating parity bit

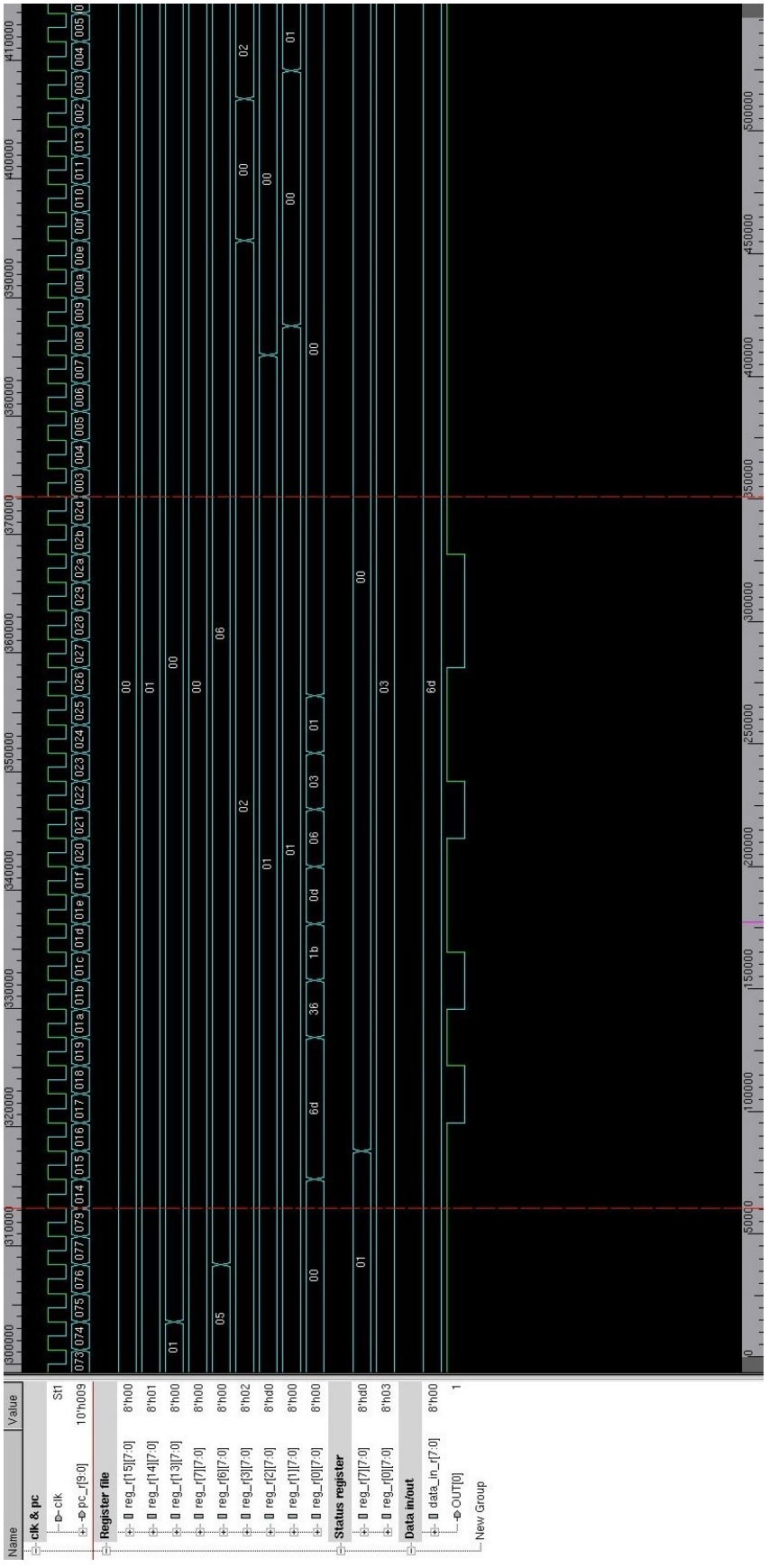

FIGURE A.7: Shifting data out

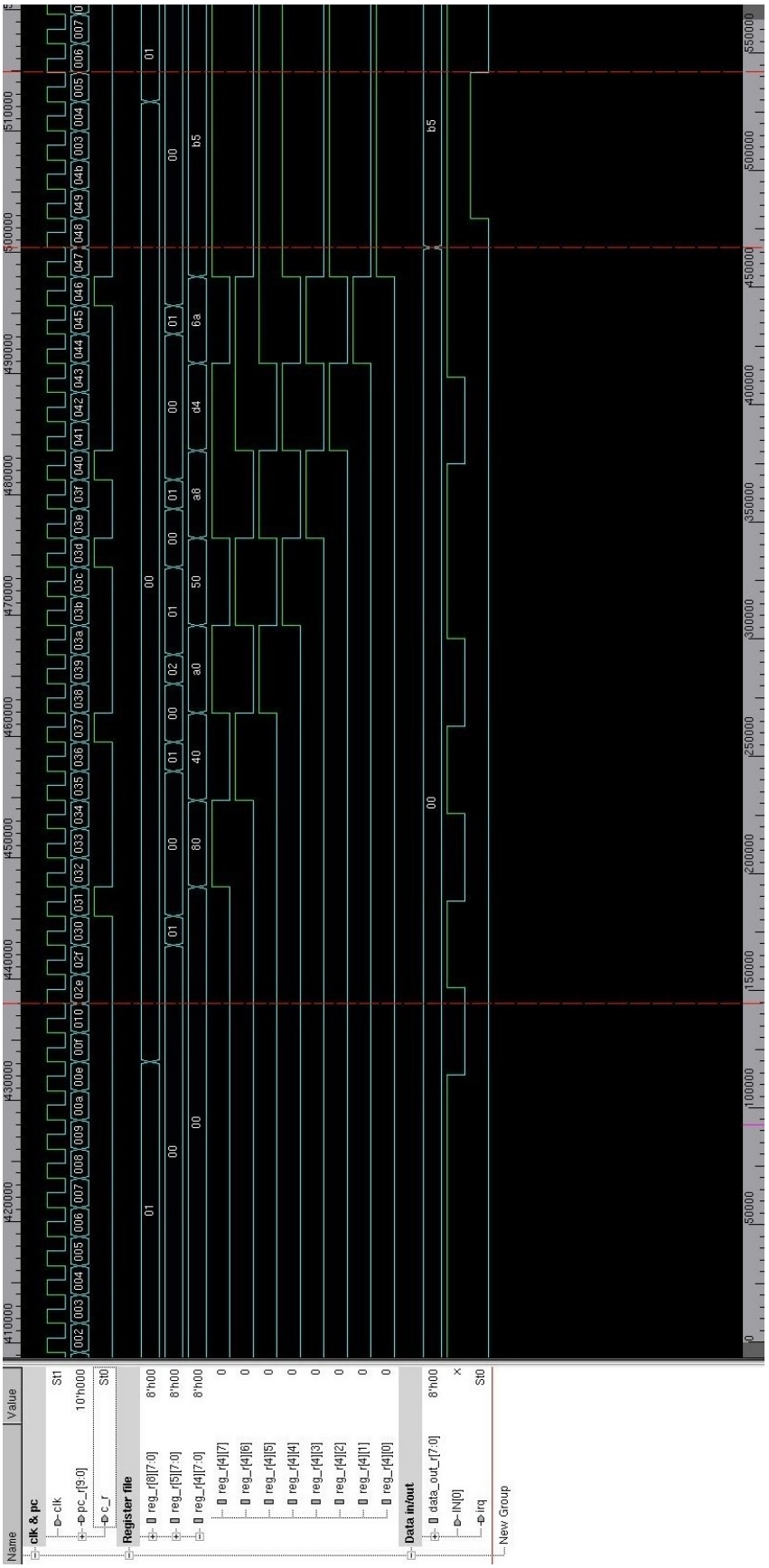

FIGURE A.8: Shifting data in

Appendix B

Simulation programs

B.1 Test of instructions

```
module instruction_test();
`include "cpu_params.sv"

  initial begin
    #10;
    tb.scan_test_mode = 0;
    tb.write_mem = 0;
    tb.instr_addr = 0;
    tb.instr_data = 0;
  end

  task run();
    begin
      $display("Start test");
wait(tb.clk==0);
tb.U_DUT.CS1.reg_r[0] = 8'h03;
tb.U_DUT.C1.R1.reg_r[0] = 8'h00;
tb.U_DUT.C1.R1.reg_r[1] = 8'h01;
tb.U_DUT.C1.R1.reg_r[2] = 8'h02;
tb.U_DUT.C1.R1.reg_r[3] = 8'h03;
tb.U_DUT.C1.R1.reg_r[4] = 8'h04;
tb.U_DUT.C1.R1.reg_r[5] = 8'hab;
tb.U_DUT.C1.R1.reg_r[6] = 8'hac;
tb.U_DUT.C1.R1.reg_r[7] = 8'had;
tb.U_DUT.C1.R1.reg_r[8] = 8'hae;
tb.U_DUT.C1.R1.reg_r[9] = 8'haf;
tb.U_DUT.C1.R1.reg_r[10] = 8'hfa;
tb.U_DUT.C1.R1.reg_r[11] = 8'hfb;
tb.U_DUT.C1.R1.reg_r[12] = 8'hfc;
tb.U_DUT.C1.R1.reg_r[13] = 8'hfd;
tb.U_DUT.C1.R1.reg_r[14] = 8'hfe;
tb.U_DUT.C1.R1.reg_r[15] = 8'hff;
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[0] = 32'h00000000;
// NOP, NOP
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[1] = 32'h1cf11cf1;
// ADC rf<-r1, ADC rf<-r0
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[2] = 32'h0cee0c21;
// ADD re<-re, ADD r2<-r1
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[3] = 32'h945a205c;
// DEC r5, AND r5<-rc
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[4] = 32'h186e1864;
// SUB r6<-rd, SUB r6<-r4
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[5] = 32'h08d108d0;
// SBC rd<-r1, SBC rd<-r0
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[6] = 32'h40324f3f;
```

```
// SBCI r3<-02, SBCI r3<-ff
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[7] = 32'h94379436;
// ROR r3, LSR r3
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[8] = 32'h24452865;
// XOR r4<-r5, OR r6<-r5
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[9] = 32'h14781477;
// CP r7-r8, CP r7-r7
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[10] = 32'h04870477;
// CPC r7-r8, CPC r7-r7
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[11] = 32'h3b60306a;
// CPI r6-b0, CPI r6-0a
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[12] = 32'h927f2c67;
// PUSH r7, MOV r6<-r7
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[13] = 32'h929f928f;
// PUSH r9, PUSH r8
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[14] = 32'h92bf92af;
// PUSH r11, PUSH r10
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[15] = 32'he281e070;
// LDI r8<-21, LDI r7<-00
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[16] = 32'h907fef9c;
// POP r7, LDI r9<-fc
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[17] = 32'h909f908f;
// POP r9, POP r8
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[18] = 32'haa70a972;
// STS io_reg<- r7, cs_r[2]<-r7
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[19] = 32'hab03ab22;
// STS timer_compare<-{r0,r3}
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[20] = 32'hab44e047;
// timer_c_r<-r4, r4<-07
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[21] = 32'h00000000;
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[22] = 32'h00000000;
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[23] = 32'h0000a3f5;
// r15<-timer_c_r
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[24] = 32'hf42a0ced;
// BRCS +5, ADD r14<-r13
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[25] = 32'h0000f442;
// NOP, BRPL +8
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[26] = 32'h00000000;
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[27] = 32'h0000f7e0;
// NOP, BRCC -4
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[28] = 32'h00000000;
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[29] = 32'h0000940c;
// NOP, JMP 01
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[30] = 32'h00000001;
tb.U_DUT.C1.PM.U_SRAM_MEM.u0.mem_core_array[31] = 32'h00000000;
#1500;

      $display("TEST OK");
    end
  endtask // run
endmodule // instruction_test
```

B.2 UART program

```
module toptest();
`include "cpu_params.sv"

  initial begin
    #10;
    tb.write_mem = 0;
    tb.instr_addr = 0;
    tb.instr_data = 0;
    tb.scan_test_mode = 0;

  end
```

```
  task run();
    begin
$display("Start test");
wait(tb.clk==0);
#24;
tb.paddr = 32'h00000000; //reset cpu and hold PC so prog_mem can be programmed
tb.pwdata = 8'h00;
tb.pwrite = 1'b1;
tb.psel = 1'b1;
tb.penable_cs = 1'b1;

tb.instr_addr = 9'h000;    // start programming of prog_mem
tb.instr_data = 32'h00000000;     // NOP
tb.write_mem = 1'b1;
#24;
tb.penable_cs = 1'b0;
tb.instr_addr = 9'h001;
tb.instr_data = 32'he011e031;   // LDI r1<-1, r3<-1
#24;
tb.instr_addr = 9'h002;
tb.instr_data = 32'he0e1a580;   // LDI r14<-1, LDS r8<-io_port
#24;
tb.instr_addr = 9'h003;
tb.instr_data = 32'ha127ade3;   // LDS r2<-cs_r[7], FORCE OUT r14 (set stop bit)
#24;
tb.instr_addr = 9'h004;
tb.instr_data = 32'h14f12012;   // CP r15 r1, AND r1<- r2
#24;
tb.instr_addr = 9'h005;
tb.instr_data = 32'h940cf420;   // JMP addr 52, BRCC +4
#24;
tb.instr_addr = 9'h006;
tb.instr_data = 32'h00000052;   // Jump to START PARITY
#24;
tb.instr_addr = 9'h007;
tb.instr_data = 32'h2038a580;   // AND r3<-r8, LDS r8<-io_port
#24;
tb.instr_addr = 9'h008;
tb.instr_data = 32'h940cf0f1;   // JMP addr3, BREQ to shift in (addr +30)
#24;
tb.instr_addr = 9'h009;
tb.instr_data = 32'h00000002;
#24;
//START SHIFT OUT
tb.instr_addr = 9'h00a;
tb.instr_data = 32'ha9f7a200;   // STS cs_reg[7]<-r15, LDS r0<-io_reg
#24
tb.instr_addr = 9'h00b;
tb.instr_data = 32'hadf3adf3;   // FORCE OUT<-r15, FORCE OUT<-r15 (set start bit)
#24;
tb.instr_addr = 9'h00c;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
tb.instr_addr = 9'h00d;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
tb.instr_addr = 9'h00e;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
tb.instr_addr = 9'h00f;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
tb.instr_addr = 9'h010;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
tb.instr_addr = 9'h011;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
tb.instr_addr = 9'h012;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
```

```
tb.instr_addr = 9'h013;
tb.instr_data = 32'h9406ad03;   // LSR r0, FORCE OUT<-r0
#24;
tb.instr_addr = 9'h014;
tb.instr_data = 32'had63ad63;   // FORCE OUT<-r6, FORCE OUT<-r6(parity bit)
#24;
tb.instr_addr = 9'h015;
tb.instr_data = 32'h940cade3;   // JMP, FORCE OUT<-r14(stop bit)
#24;
tb.instr_addr = 9'h016;
tb.instr_data = 32'h00000003;   // Jump to start
#24;
//START SHIFT IN
tb.instr_addr = 9'h017;
tb.instr_data = 32'ha550e040;   // LDS r5<-io_port, LDI r4<-0
#24;
tb.instr_addr = 9'h018;
tb.instr_data = 32'h94479456;   // ROR r4, LSR r5
#24;
tb.instr_addr = 9'h019;
tb.instr_data = 32'h9456a550;   // LSR r5, LDS r5<-io_port
#24;
tb.instr_addr = 9'h01a;
tb.instr_data = 32'ha5509447;   // LDS r5<-io_port, ROR r4
#24;
tb.instr_addr = 9'h01b;
tb.instr_data = 32'h94479456;   // ROR r4, LSR r5
#24;
tb.instr_addr = 9'h01c;
tb.instr_data = 32'h9456a550;   // LSR r5, LDS r5<-io_port
#24;
tb.instr_addr = 9'h01d;
tb.instr_data = 32'ha5509447;   // LDS r5<-io_port, ROR r4
#24;
tb.instr_addr = 9'h01e;
tb.instr_data = 32'h94479456;   // ROR r4, LSR r5
#24;
tb.instr_addr = 9'h01f;
tb.instr_data = 32'h9456a550;   // LSR r5, LDS r5<-io_port
#24;
tb.instr_addr = 9'h020;
tb.instr_data = 32'ha5509447;   // LDS r5<-io_port, ROR r4
#24;
tb.instr_addr = 9'h021;
tb.instr_data = 32'h94479456;   // ROR r4, LSR r5
#24;
tb.instr_addr = 9'h022;
tb.instr_data = 32'h9456a550;   // LSR r5, LDS r5<-io_port
#24;
tb.instr_addr = 9'h023;
tb.instr_data = 32'haa409447;   // STS io_reg<-r4, ROR r4
#24;
tb.instr_addr = 9'h024;
tb.instr_data = 32'h940cace0;   // JMP addr03, STS irq <-r14
#24;
tb.instr_addr = 9'h025;
tb.instr_data = 32'h00000003;
#24;
//START CALCULATE PARITY
tb.instr_addr = 9'h026;
tb.instr_data = 32'he0e1a270;   // LDI r14<-1, LDS r7<-io_reg
#24;
tb.instr_addr = 9'h027;
tb.instr_data = 32'h2cdee060;   // MOV r13<-r14, LDI r6<-0
#24;
tb.instr_addr = 9'h028;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h029;
tb.instr_data = 32'h2cde9476;   // MOV r13<-r14, LSR r7
#24;
```

```
tb.instr_addr = 9'h02a;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h02b;
tb.instr_data = 32'h2cde9476;   // MOV r13<-r14, LSR r7
#24;
tb.instr_addr = 9'h02c;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h02d;
tb.instr_data = 32'h2cde9476;   // MOV r13<-r14, LSR r7
#24;
tb.instr_addr = 9'h02e;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h02f;
tb.instr_data = 32'h2cde9476;   // MOV r13<-r14, LSR r7
#24;
tb.instr_addr = 9'h030;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h031;
tb.instr_data = 32'h2cde9476;   // MOV r13<-r14, LSR r7
#24;
tb.instr_addr = 9'h032;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h033;
tb.instr_data = 32'h2cde9476;   // MOV r13<-r14, LSR r7
#24;
tb.instr_addr = 9'h034;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h035;
tb.instr_data = 32'h2cde9476;   // MOV r13<-r14, LSR r7
#24;
tb.instr_addr = 9'h036;
tb.instr_data = 32'h0c6d20d7;   // ADD r6<-r13, AND r13<-r7
#24;
tb.instr_addr = 9'h037;
tb.instr_data = 32'h940c9463;   // JUMP to 14, INC r6
#24;
tb.instr_addr = 9'h038;
tb.instr_data = 32'h00000014;   // Jump to START SHIFT OUT

#24;
tb.IN = 8'h01;
tb.write_mem = 1'b0;            // stop writing to U_SRAM_MEM
tb.paddr = 32'h00000007;        // writing to cs_reg[7]
tb.pwdata = 8'hd0;              // writing data 8'h00
tb.penable_cs = 1'b1;           // enable cs_reg to write data
#24;
tb.paddr = 32'h00000000;        // writing to cs_reg[0]
tb.pwdata = 8'h03;              // writing data 8'h03 (enable cpu)
#48;
tb.penable_cs = 1'b0;
#240;
tb.pwdata = 8'h6d;              // writing 'h6d to io_reg
tb.penable_io = 1'b1;           // writing 'h6d to io_reg
#24;
tb.paddr = 32'h00000007;        // writing to cs_reg
tb.pwdata = 8'h01;              // set cs_reg[7][0] to 1 for cpu to shift out
tb.penable_io = 1'b0;
tb.penable_cs = 1'b1;
#24;
tb.penable_cs = 1'b0;
#2544;
tb.IN = 8'h0;                                   // Start data to be shifted in
#72;
tb.penable_cs = 1'b0;
tb.IN = 8'h1;
```

```
#72;
tb.IN = 8'h0;
#72;
tb.IN = 8'h1;
#72;
tb.IN = 8'h2;
#72;
tb.IN = 8'h1;
#72;
tb.IN = 8'h1;
#72;
tb.IN = 8'h0;
#72;
tb.IN = 8'h1;
#240;
tb.pwdata = 8'h0;
tb.pwrite = 1'b1;
tb.psel = 1'b1;
tb.penable_irq = 1'b1;
#24;
tb.penable_irq = 1'b0;          // Clear interrupt
#480;


      $display("TEST OK");
    end
  endtask // run
endmodule // toptest
```

Appendix C

Verilog Code

C.1 Register File

```
module register (/*AUTOARG*/
  // Outputs
  data_a, data_b,
  // Inputs
  write, clk, rst_n, addr_a, addr_b, addr_d, data_d
  );

  `include "cpu_params.sv"

  input write, clk, rst_n;
  input [ADDR_MSB:0] addr_a, addr_b, addr_d;
  input [DATA_MSB:0]  data_d;
  output [DATA_MSB:0]  data_a, data_b;

  reg [DATA_MSB:0] data_a, data_b;
  reg [DATA_MSB:0]  reg_r[REG_NUMBERS:0];
  int i;


  always@(posedge clk or  negedge rst_n) begin
    if (!rst_n) begin
      for (i = 0; i <= REG_NUMBERS; i=i+1) begin
        reg_r[i] = 'b00000000;
      end
    end
    else if (write) begin
      case (addr_d)
        0 : begin reg_r[0] <= data_d; end
        1 : begin reg_r[1] <= data_d; end
        2 : begin reg_r[2] <= data_d; end
        3 : begin reg_r[3] <= data_d; end
        4 : begin reg_r[4] <= data_d; end
        5 : begin reg_r[5] <= data_d; end
        6 : begin reg_r[6] <= data_d; end
        7 : begin reg_r[7] <= data_d; end
        8 : begin reg_r[8] <= data_d; end
        9 : begin reg_r[9] <= data_d; end
        10 : begin reg_r[10] <= data_d; end
        11 : begin reg_r[11] <= data_d; end
        12 : begin reg_r[12] <= data_d; end
        13 : begin reg_r[13] <= data_d; end
        14 : begin reg_r[14] <= data_d; end
        15 : begin reg_r[15] <= data_d; end
      endcase // case (addrd)
    end // if (write)
```

```
  end // always@ (posedge clk)

  always@(*) begin
    data_a = reg_r[addr_a];
    data_b = reg_r[addr_b];
  end

endmodule // register
```

C.2 ALU

```
module alu (/*AUTOARG*/
  // Outputs
  data_d, v_r, c_r, n_r, z_r, do_cpu,
  // Inputs
  clk, rst_n, ms, data_a, data_b, di_cpu, constant, fs
  );

`include "cpu_params.sv"

  input clk, rst_n;
  input [1:0] ms;
  input [DATA_MSB:0] data_a, data_b, di_cpu, constant;
  input [MSB_FS:0] fs;
  output [DATA_MSB:0] data_d;
  output v_r, c_r, n_r, z_r;
  output [DATA_MSB:0] do_cpu;

  reg [DATA_MSB:0]  data_d;
  reg v_r, c_r, n_r, z_r;
  reg v, c, n, z;
  reg [DATA_SIZE:0] temp;
  reg [DATA_MSB:0]  b_in;

  assign do_cpu = b_in;

  always@(*) begin
    case (ms)
      0 : begin b_in = data_b; end
      1 : begin b_in = di_cpu; end
      2 : begin b_in = constant; end
      default : begin b_in = data_b; end
    endcase // case (ms)
  end // always@(*)

  always@(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      v_r <= 1'b0;
      c_r <= 1'b0;
      n_r <= 1'b0;
      z_r <= 1'b0;
    end
    else begin
      v_r <= v;
      c_r <= c;
      n_r <= n;
      z_r <= z;
    end // else: !if(rst_n == 0)
  end // always@ (negedge clk or negedge rst_n)

  always@(*) begin
    /*AUTORESET*/
    // Beginning of autoreset for uninitialized flops
    c = 1'h0;
    data_d = {(1+(DATA_MSB)){1'b0}};
    n = 1'h0;
```

```
temp = {(1+(DATA_SIZE)){1'b0}};
v = 1'h0;
z = 1'h0;
// End of automatics
case (fs)
  NOP : begin                       // NOP
    data_d = 'b0;
  end
  ADD : begin                         // ADD
    temp = data_a + b_in;
    {c,data_d} = data_a + b_in;
   // c = temp[DATA_SIZE];
    z = (temp == 0) ? 1 : 0;
  end
  ADC : begin                       // ADC
    temp = data_a + b_in + c_r;
    {c,data_d} = data_a + b_in +c_r;
   // c = temp[DATA_SIZE];
    z = (temp == 0) ? 1:0;
  end
  SUB : begin                       // SUB, SUBI
    data_d = data_a - b_in;
    c = (b_in > data_a) ? 1:0;
    z = (data_a==b_in) ?  1:0;
  end
  SBC: begin                        // SBC, SBCI
    data_d =  data_a - b_in - c_r;
    c = (b_in+c_r > data_a) ? 1:0;
    z = ((data_a-b_in-c_r)==0) ? 1:0;
  end
  LDI : begin                       // LDI
    data_d = b_in;
  end
  LSL : begin                       // LSL
    data_d = (data_a << 1);
    c = data_a[DATA_MSB];
    z = (data_a << 1) ? 0:1;
  end
  LSR : begin                       // LSR
    data_d = (data_a >> 1);
    c = data_a[0];
    z = (data_a >> 1) ? 0:1;
  end
  ROL : begin                       // ROL
    temp = (data_a << 1);
    temp[0] = c_r;
    data_d = temp[DATA_MSB:0];
    c = data_a[DATA_MSB];
    z = (temp[DATA_MSB:0]) ? 0:1;
  end
  ROR : begin                       // ROR
    temp = (data_a << 1);
    temp = (temp >> 1);
    temp[DATA_SIZE] = c_r;
    data_d = temp[DATA_SIZE:1];
    c = temp[0];
    z = (temp[DATA_SIZE:1]) ? 0:1;
  end
  AND : begin                       // AND
    data_d = data_a & b_in;
    z = (data_a & b_in) ? 0:1;
  end
  OR : begin                        // OR, ORI
    data_d = data_a | b_in;
    z = (data_a | b_in) ? 0:1;
  end
  XOR : begin                       // XOR, CLR
    data_d = data_a ^ b_in;
    z = (data_a ^ b_in) ? 0:1;
  end
  CP : begin                        // CP, CPI
```

```
        c = (b_in > data_a) ? 1:0;
        z = (data_a==b_in) ?  1:0;
      end
      CPC : begin                       // CPC
        c = (b_in+c_r > data_a) ? 1:0;
        z = ((data_a-b_in-c_r)==0) ? 1:0;
      end
      DEC : begin                       // DEC
        data_d = data_a - 1;
        z = (data_a-1)==0 ? 1:0;
      end
      default : begin
        temp = 0;
        data_d = 0;
        v = 1'b0;
        c = 1'b0;
        n = 1'b0;
        z = 1'b0;
      end
    endcase // case (fs)
  end // always@ (*)
endmodule // alu
```

C.3 Instruction Decoder

```
module decode (/*AUTOARG*/
  // Outputs
  addr_a, addr_b, addr_d, constant, write, pl, jb, bset, hold_pc, bc, fs, ms,
  databus_addr, write_bus, read_bus, offset,
  // Inputs
  instr_pntr, done_hold, di_ireg, cpu_reset_n, cpu_disable_n, msd
  );

`include "cpu_params.sv"

  input instr_pntr;
  input done_hold;              // from brctrl to tell that the pc have been holded
  input [31:0] di_ireg;
  input cpu_reset_n, cpu_disable_n;
  input msd;                            // Mux Select Decode
  output [ADDR_MSB:0] addr_a, addr_b, addr_d;
  output [DATA_MSB:0] constant;
  output write, pl, jb, bset;
  output hold_pc;               // to brctrl to hold the pc for one clock cycle
  output [2:0] bc;
  output [MSB_FS:0] fs;
  output [MUXA_MSB:0] ms;
  output [BUSADDR_MSB:0] databus_addr;
  output write_bus, read_bus;
  output [6:0] offset;

  reg [ADDR_MSB:0] addr_a, addr_b, addr_d;
  reg [DATA_MSB:0] constant;
  reg write, pl, jb, bset, write_bus, read_bus;
  reg hold_pc;
  reg [2:0] bc;
  reg [MSB_FS:0] fs;
  reg [MUXA_MSB:0] ms;
  reg [BUSADDR_MSB:0] databus_addr;
  reg [OPCODE_MSB:0] opcode;
  reg [6:0] offset;
  wire [31:0] ireg;

  /*AUTOINPUT*/
  /*AUTOREG*/
```

```
/*AUTOWIRE*/

assign ireg = msd ? 32'h00000000:di_ireg;

always@(*) begin
  opcode = ireg[15:0];
  offset = ireg[9:3];
  if (!instr_pntr) begin
    opcode = ireg[15:0];
    offset = ireg[9:3];
  end
  else begin
    opcode = ireg[31:16];
    offset = ireg[25:19];
  end
end

always@(*) begin
  /*AUTORESET*/
  // Beginning of autoreset for uninitialized flops
  addr_a = {(1+(ADDR_MSB)){1'b0}};
  addr_b = {(1+(ADDR_MSB)){1'b0}};
  addr_d = {(1+(ADDR_MSB)){1'b0}};
  bc = 3'h0;
  bset = 1'h0;
  constant = {(1+(DATA_MSB)){1'b0}};
  databus_addr = {(1+(BUSADDR_MSB)){1'b0}};
  fs = {(1+(MSB_FS)){1'b0}};
  hold_pc = 1'h0;
  jb = 1'h0;
  ms = {(1+(MUXA_MSB)){1'b0}};
  pl = 1'h0;
  read_bus = 1'h0;
  write = 1'h0;
  write_bus = 1'h0;
  // End of automatics
  if (!cpu_disable_n || !cpu_reset_n) begin         // CPU disabeled
    hold_pc = 1;
    addr_a = 'b0000;
    addr_b = 'b0000;
    addr_d = 'b0000;
    write = 'b0;
    write_bus = 'b0;
    read_bus = 'b0;
    pl = 'b0;
    jb = 'b0;
    bc = 'b000;
    fs = NOP;
    ms = 'b0;
  end // if (!cpu_disable_n)
  else if (opcode == OPC_NOP) begin    // NOP
    hold_pc = 0;
    addr_a = 'b0000;
    addr_b = 'b0000;
    addr_d = 'b0000;
    write = 'b0;
    write_bus = 'b0;
    read_bus = 'b0;
    pl = 'b0;
    jb = 'b0;
    bc = 'b000;
    fs = NOP;
    ms = 'b0;
  end // if (opcode == OPC_NOP)
  else if (opcode[OPCODE_MSB:10] == OPC_ADC) begin //ADC and  ROL
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
```

```
    read_bus = 0;
    ms = 0;
    fs = ADC;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_ADC)
  else if (opcode[OPCODE_MSB:10] == OPC_ADD) begin // ADD, LSL
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = ADD;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_ADD)
  else if (opcode[OPCODE_MSB:10] == OPC_AND) begin // AND
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = AND;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_AND)
  else if (opcode[OPCODE_MSB:9] == OPC_DEC[10:4]
  && opcode[3:0] == OPC_DEC[3:0]) begin // DEC
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    fs = DEC;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:9] == OPC_DEC[10:4] && opcode[3:0] == OPC_DEC[3:0])
  else if (opcode[OPCODE_MSB:10] == OPC_SUB) begin // SUB
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = SUB;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_SUB)
  else if (opcode[OPCODE_MSB:12] == OPC_SUBI) begin // SUBI
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 2;
    fs = SUB;
```

```
    constant = {opcode[11:8],opcode[3:0]};
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:12] == OPC_SUBI)
  else if (opcode[OPCODE_MSB:10] == OPC_SBC) begin // SBC
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = SBC;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_SBC)
  else if (opcode[OPCODE_MSB:12] == OPC_SBCI) begin // SBCI
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 2;
    fs = SBC;
    constant = {opcode[11:8],opcode[3:0]};
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:12] == OPC_SBCI)
  else if (opcode[OPCODE_MSB:12] == OPC_LDI) begin // LDI
    hold_pc = 0;
    addr_d = opcode[7:4];
    constant = {opcode[11:8], opcode[3:0]};
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 2;
    fs = LDI;
    pl = 0;
    jb = 0;
    bc = 0;
  end
  else if (opcode[OPCODE_MSB:9] == OPC_LSR[10:4]
  && opcode[3:0] == OPC_LSR[3:0]) begin // LSR
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = LSR;
    pl = 0;
    jb = 0;
    bc = 0;
  end
  else if (opcode[OPCODE_MSB:9] == OPC_ROR[10:4]
  && opcode[3:0] == OPC_ROR[3:0]) begin // ROR
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = ROR;
    pl = 0;
```

```
    jb = 0;
    bc = 0;
  end
  else if (opcode[OPCODE_MSB:10] == OPC_OR) begin // OR
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = OR;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_OR)
  else if (opcode[OPCODE_MSB:10] == OPC_XOR) begin // XOR
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    addr_d = opcode[7:4];
    write = 1;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = XOR;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_XOR)
  else if (opcode[OPCODE_MSB:10] == OPC_CP) begin // CP
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    write = 0;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = CP;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_CP)
  else if (opcode[OPCODE_MSB:10] == OPC_CPC) begin // CPC
    hold_pc = 0;
    addr_a = opcode[7:4];
    addr_b = opcode[3:0];
    write = 0;
    write_bus = 0;
    read_bus = 0;
    ms = 0;
    fs = CPC;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_CPC)
  else if (opcode[OPCODE_MSB:12] == OPC_CPI) begin // CPI
    hold_pc = 0;
    addr_a = opcode[7:4];
    constant = {opcode[11:8], opcode[3:0]};
    write = 0;
    write_bus = 0;
    read_bus = 0;
    ms = 2;
    fs = CP;
    pl = 0;
    jb = 0;
    bc = 0;
  end // if (opcode[OPCODE_MSB:10] == OPC_CPI)
  else if (opcode[OPCODE_MSB:10] == OPC_MOV) begin // MOV
```

```
  hold_pc = 0;
  addr_b = opcode[3:0];
  addr_d = opcode[7:4];
  write = 1;
  write_bus = 0;
  read_bus = 0;
  ms = 0;
  fs = LDI;
  pl = 0;
  jb = 0;
  bc = 0;
end // if (opcode[OPCODE_MSB:10] = OPC_MOV)
else if (opcode[OPCODE_MSB:10] == OPC_BRVS[8:3]
&& opcode[2:0] == OPC_BRVS[2:0]) begin // BRVS
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  read_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 1;
  bc = 'b00;
end
else if (opcode[OPCODE_MSB:10] == OPC_BRVC[8:3]
&& opcode[2:0] == OPC_BRVC[2:0]) begin // BRVC
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  read_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 0;
  bc = 'b00;
end
else if (opcode[OPCODE_MSB:10] == OPC_BRCS[8:3]
&& opcode[2:0] == OPC_BRCS[2:0]) begin // BRCS
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  read_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 1;
  bc = 'b01;
end
else if (opcode[OPCODE_MSB:10] == OPC_BRCC[8:3]
&& opcode[2:0] == OPC_BRCC[2:0]) begin // BRCC
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  read_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 0;
  bc = 'b01;
end
else if (opcode[OPCODE_MSB:10] == OPC_BRMI[8:3]
&& opcode[2:0] == OPC_BRMI[2:0]) begin // BRMI
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  read_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 1;
  bc = 'b10;
```

```
end
else if (opcode[OPCODE_MSB:10] == OPC_BRPL[8:3]
&& opcode[2:0] == OPC_BRPL[2:0]) begin // BRPL
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  read_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 0;
  bc = 'b10;
end
else if (opcode[OPCODE_MSB:10] == OPC_BREQ[8:3]
&& opcode[2:0] == OPC_BREQ[2:0]) begin // BREQ
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 1;
  bc = 'b11;
end
else if (opcode[OPCODE_MSB:10] == OPC_BRNE[8:3]
&& opcode[2:0] == OPC_BRNE[2:0]) begin // BRNE
  hold_pc = 0;
  write = 0;
  write_bus = 0;
  read_bus = 0;
  fs = NOP;
  pl = 1;
  jb = 1;
  bset = 0;
  bc = 'b11;
end
else if (opcode[OPCODE_MSB:9] == OPC_PUSH[10:4]
&& opcode[3:0] == OPC_PUSH[3:0]) begin // PUSH
  if (!done_hold) begin
    hold_pc = 1;
    addr_b = opcode[7:4];
    ms = 0;
    write = 0;
    fs = NOP;
    pl = 0;
    jb = 0;
    bc = 0;
    databus_addr = 'b000000;
    write_bus = 0;
    read_bus = 0;
  end
  else begin
    hold_pc = 0;
    addr_b = opcode[7:4];
    ms = 0;
    write = 0;
    fs = NOP;
    pl = 0;
    jb = 0;
    bc = 0;
    databus_addr = 'b000000;
    write_bus = 1;
    read_bus = 0;
  end // else: !if(!done_hold)
end // if (opcode[OPCODE_MSB:9] == OPC_PUSH[10:4]
&& opcode[3:0] == OPC_PUSH[3:0])
else if (opcode[OPCODE_MSB:9] == OPC_POP[10:4]
&& opcode[3:0] == OPC_POP[3:0]) begin // POP
  if (!done_hold) begin
    hold_pc = 1;
    addr_b = opcode[7:4];
```

```
        ms = 0;
        write = 0;
        fs = NOP;
        pl = 0;
        jb = 0;
        bc = 0;
        databus_addr = 'b000000;
        write_bus = 0;
        read_bus = 0;
      end
      else begin
        hold_pc = 0;
        ms = 1;
        write = 1;
        write_bus = 0;
        read_bus = 1;
        addr_d = opcode[7:4];
        fs = LDI;
        pl = 0;
        jb = 0;
        bc = 0;
        databus_addr = 'b000000;
      end // else: !if(!done_hold)
    end // if (opcode[OPCODE_MSB:9] == OPC_POP[10:4] && opcode[3:0] == OPC_POP[3:0])

    else if (opcode[OPCODE_MSB:11] == OPC_STS) begin // STS
      hold_pc = 0;
      ms = 0;
      write = 0;
      write_bus = 1;
      read_bus = 0;
      addr_b = opcode[7:4];
      fs = NOP;
      pl = 0;
      jb = 0;
      bc = 0;
      databus_addr = {opcode[10:8],opcode[3:0]};
    end // if (opcode[OPCODE_MSB:11] == OPC_STS)
    else if (opcode[OPCODE_MSB:11] == OPC_LDS) begin // LDS
      hold_pc = 0;
      ms = 1;
      write = 1;
      write_bus = 0;
      read_bus = 1;
      addr_d = opcode[7:4];
      fs = LDI;
      pl = 0;
      jb = 0;
      bc = 0;
      databus_addr = {opcode[10:8],opcode[3:0]};
    end // if (opcode[OPCODE_MSB:11] == OPC_LDS)
    else if (opcode[OPCODE_MSB:0] == OPC_JMP) begin // JMP
      hold_pc = 0;
      ms = 0;
      write = 0;
      write_bus = 0;
      read_bus = 0;
      fs = NOP;
      pl = 1;
      jb = 0;
      bc = 0;
    end // if (opcode[OPCODE_MSB:0] == OPC_JMP)
  end // always@ (posedge clk or negedge rst_n)
endmodule // decode
```

C.4 Branch control

```
module brctrl (/*AUTOARG*/
  // Outputs
  pc_r, done_hold, msd,
  // Inputs
  clk, rst_n, v_r, c_r, n_r, z_r, pl, jb, bset, bc, hold_pc, cpu_reset_n,
  offset, jmp_addr
  );

`include "cpu_params.sv"

  input clk, rst_n, v_r, c_r, n_r, z_r, pl, jb, bset;
  input [2:0] bc;
  input hold_pc, cpu_reset_n;
  input signed [6:0] offset;
  input [PC_MSB:0] jmp_addr;
  output [PC_MSB:0] pc_r;
  output done_hold;
  output msd;                          // Mux Select Decode

  reg [PC_SIZE:0] pc;
  reg [PC_MSB:0] pc_r;
  reg done_hold, jmp_ctrl, msd;
  reg signed [(PC_SIZE+2):0] temp;
  wire signed  [(PC_MSB+2):0] s_pc, s_offset;

  always@(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      pc_r <= 'h0;
      done_hold <= 'b0;
      jmp_ctrl <= 0;                   // To controll jump procedure
      msd <= 0;
    end
    else if (!cpu_reset_n) begin
      pc_r <= 'h0;
      done_hold <= 'b0;
      msd <= 0;
    end
    else if (hold_pc) begin            // if cpu is disabeled
      pc_r <= pc[PC_MSB:0]-1;
      done_hold <= 1;
      msd <= 0;
    end
    else if (jmp_ctrl) begin           // if a jump is to be done
      jmp_ctrl <= 0;
      pc_r <= jmp_addr;
      msd <= 0;
    end
    else if (pl && !jb) begin          // if jump is called
      pc_r <= pc_r+2;
      jmp_ctrl <= 1;
      msd <= 1;
    end
    else begin
      pc_r <= pc[PC_MSB:0];
      done_hold <= 0;
      msd <= 0;
    end
  end

  assign s_offset = {5'b11111, offset}; // signed offset
  assign s_pc = {2'b00,pc_r};              // signed pc

  always@(*) begin
    pc = 0;
    temp = 0;
    if (pc_r == 0) begin
      pc = 1;
      temp = 0;
```

```
end
else if (pc[PC_SIZE] == 1) begin       // pc owerflow
  pc = 0;
end
else  if (!pl) begin              // if not jump/branch
  pc = pc_r+1;
end
else if (jb && bc==’b00) begin    // if branch and trigging on v_r
  if (v_r && bset) begin          // trigging on v_r set
    if (offset[6] == ’b1) begin   // if offset is negative
      temp = s_pc+s_offset;
      pc = {1’b0,temp[PC_MSB:0]};
    end
    else begin                    // offset not negative
      pc = pc_r+offset;
    end
  end
  else if (!v_r && !bset) begin   // trigging on v_r not set
    if (offset[6] == ’b1) begin   // if offset negative
      temp = s_pc+s_offset;
      pc = {1’b0,temp[PC_MSB:0]};
    end
    else begin                    // offset not negative
      pc = pc_r+offset;
    end
  end
  else begin
    pc = pc_r+1;
  end // else: !if(!v_r && !bset)
end // if (jb && bc==’b00)
else if (jb && bc==’b01) begin
  if (c_r && bset) begin
    if (offset[6] == ’b1) begin
      temp = s_pc + s_offset;
      pc = {1’b0,temp[PC_MSB:0]};
    end
    else begin
      pc = pc_r+offset;
    end
  end
  else if (!c_r && !bset) begin
    if (offset[6] == ’b1) begin
      temp = s_pc+s_offset;
      pc = {1’b0,temp[PC_MSB:0]};
    end
    else begin
      pc = pc_r+offset;
    end
  end
  else begin
    pc = pc_r+1;
  end // else: !if(c_r)
end // if (jb && bc==’b01)
else if (jb && bc==’b10) begin
  if (n_r && bset) begin
    if (offset[6] == ’b1) begin
      temp = s_pc + s_offset;
      pc = {1’b0,temp [PC_MSB:0]};
    end
    else begin
      pc = pc_r+offset;
    end
  end
  else if (!n_r && !bset) begin
    if (offset[6] == ’b1) begin
      temp = s_pc+s_offset;
      pc = {1’b0,temp[PC_MSB:0]};
    end
    else begin
      pc = pc_r+offset;
    end
```

```
      end
      else begin
        pc = pc_r+1;
      end // else: !if(!n_r && !bset)
    end // if (jb && bc==’b10)
    else if (jb && bc==’b11) begin
      if (z_r && bset) begin
        if (offset[6] == ’b1) begin
          temp = s_pc + s_offset;
          pc = {1’b0,temp [PC_MSB:0]};
        end
        else begin
          pc = pc_r+offset;
        end
      end
      else if (!z_r && !bset) begin
        if (offset[6] == ’b1) begin
          temp = s_pc+s_offset;
          pc = {1’b0,temp[PC_MSB:0]};
        end
        else begin
          pc = pc_r+offset;
        end
      end
      else begin
        pc = pc_r+1;
      end // else: !if(!z_r && !bset)
    end // if (jb && bc==’b11)
  end // always@ (*)
endmodule // brctrl
```

C.5 Program Memory

```
module prog_mem (/*AUTOARG*/
  // Outputs
  di_ireg, instr_pntr,
  // Inputs
  clk, write_mem, pc_r, instr_addr, instr_data, scan_test_mode
  );

  `include "cpu_params.sv"

  input clk, write_mem;
  input [PC_MSB:0] pc_r;
  input [8:0] instr_addr;
  input [31:0] instr_data;
  output [31:0] di_ireg;
  output instr_pntr;    // output to point to part of di_ireg to be decoded

  input scan_test_mode;

  reg [31:0] di_ireg;
  reg [8:0] sram_addr;
  reg [31:0] sram_di;
  reg [3:0] sram_wem;
  reg sram_we;
  wire [31:0] tmp_do;
  wire instr_pntr;      // points to part of di_ireg to be decoded
  wire [8:0] mem_pntr;  // points to the memory location to be read
  wire clk_n;

  assign clk_n = ~clk;
  assign mem_pntr = (pc_r >> 1);
  assign instr_pntr = pc_r[0];

  always@(*) begin
```

```
    sram_we = 0;
    sram_addr = 0;
    sram_di = 0;
    di_ireg = 0;
    sram_wem = 4'b0000;
    if (!write_mem) begin
      sram_we = 0;
      sram_addr = mem_pntr;
      di_ireg = scan_test_mode ? instr_data : tmp_do ;
      sram_wem = 4'b0000;
    end
    else  begin
      sram_we = 1;
      sram_addr = instr_addr;
      sram_di = instr_data;
      di_ireg = 0;
      sram_wem = 4'b1111;
    end // else: !if(!write_mem)
  end // always@ (*)

  //SRAM block
  sram_512x32cm4sw8 U_SRAM_MEM(  .\do (tmp_do),
                               .addr (sram_addr),
                               .di (sram_di),
                               .wem (sram_wem),
                               .we (sram_we),
                               .oe (1'b1),
                               .me (1'b1),
                               .clk (clk_n),
                               .awt (1'b0),
                               .taddr (9'h0),
                               .tdi (32'h0),
                               .twem (4'h0),
                               .twe (1'b0),
                               .toe (1'b0),
                               .tme (1'b0),
                               .biste (1'b0),
                               .test1 (1'b0),
                               .rm (4'hF));

endmodule // prog_mem
```

C.6 Internal Data Bus

```
module databus (/*AUTOARG*/
  // Outputs
  write_stack, read_stack, w_cs_reg, w_io_reg, w_irq, cs_reg_addr, di_cs_reg,
  di_io_reg, di_stack, di_cpu, di_irq, di_timer, di_io_port, w_io_port,
  io_port_addr, timer_addr, r_timer, w_timer,
  // Inputs
  write_bus, read_bus, do_cpu, do_stack, databus_addr, do_cs_reg, do_io_reg,
  do_io_port, do_timer
  );

  `include "cpu_params.sv"

  input write_bus, read_bus;
  input [DATA_MSB:0] do_cpu, do_stack;
  input [BUSADDR_MSB:0] databus_addr;
  input [DATA_MSB:0] do_cs_reg, do_io_reg;
  input [DATA_MSB:0] do_io_port, do_timer;
  output write_stack, read_stack;
  output w_cs_reg, w_io_reg;
  output w_irq;
  output [EXREG_ADDR_MSB:0] cs_reg_addr;
```

```
output [DATA_MSB:0] di_cs_reg, di_io_reg;
output [DATA_MSB:0] di_stack, di_cpu;
output [DATA_MSB:0] di_irq, di_timer, di_io_port;
output w_io_port;
output [EXREG_ADDR_MSB:0] io_port_addr, timer_addr;
output r_timer, w_timer;

reg write_stack, read_stack;
reg [DATA_MSB:0]  di_stack, di_cpu;
reg [EXREG_ADDR_MSB:0] cs_reg_addr;
reg [DATA_MSB:0] di_cs_reg, di_io_reg;
reg w_cs_reg, w_io_reg;
reg [DATA_MSB:0] di_irq;
reg w_irq;
reg [DATA_MSB:0] di_io_port;
reg [EXREG_ADDR_MSB:0] io_port_addr;
reg w_io_port;
reg [DATA_MSB:0] di_timer;
reg [EXREG_ADDR_MSB:0] timer_addr;
reg w_timer, r_timer;

always@(*) begin
  case(databus_addr[BUSADDR_MSB:EXREG_ADDR_SIZE])
    0 : begin                       // Stack
      di_stack = do_cpu;
      di_cpu = do_stack;
      write_stack = write_bus;
      read_stack = read_bus;

      di_cs_reg = 0;
      w_cs_reg = 'b0;
      cs_reg_addr = 0;
      di_io_reg = 0;
      w_io_reg = 0;
      di_irq = 0;
      w_irq = 'b0;
      di_io_port = 0;
      w_io_port = 0;
      io_port_addr = 0;
      di_timer = 0;
      w_timer = 0;
      r_timer = 0;
      timer_addr = 0;
    end // case: 0
    1 : begin                       // cs register
      di_cs_reg = do_cpu;
      di_cpu = do_cs_reg;
      w_cs_reg = write_bus;
      cs_reg_addr = databus_addr[EXREG_ADDR_MSB:0];

      di_stack = 0;
      write_stack = 'b0;
      read_stack = 'b0;
      di_io_reg = 0;
      w_io_reg = 0;
      di_irq = 0;
      w_irq = 'b0;
      di_io_port = 0;
      w_io_port = 0;
      io_port_addr = 0;
      di_timer = 0;
      w_timer = 0;
      r_timer = 0;
      timer_addr = 0;
    end // case: 1
    2 : begin                       // data in/out
      di_io_reg = do_cpu;
      di_cpu = do_io_reg;
      w_io_reg = write_bus;

      di_stack = 0;
```

```
  write_stack = 'b0;
  read_stack = 'b0;
  di_cs_reg = 0;
  w_cs_reg = 'b0;
  cs_reg_addr = 0;
  di_irq = 0;
  w_irq = 'b0;
  di_io_port = 0;
  w_io_port = 0;
  io_port_addr = 0;
  di_timer = 0;
  w_timer = 0;
  r_timer = 0;
  timer_addr = 0;
end // case: 2
3 : begin                          // timer
  di_timer = do_cpu;
  di_cpu = do_timer;
  w_timer = write_bus;
  r_timer = read_bus;
  timer_addr = databus_addr[EXREG_ADDR_MSB:0];

  di_stack = 0;
  write_stack = 'b0;
  read_stack = 'b0;
  di_cs_reg = 0;
  w_cs_reg = 'b0;
  cs_reg_addr = 0;
  di_io_reg = 0;
  w_io_reg = 0;
  di_irq = 0;
  w_irq = 'b0;
  di_io_port = 0;
  w_io_port = 0;
  io_port_addr = 0;
end // case: 3
4 : begin                          // irq_ine
  di_irq = do_cpu;
  w_irq = write_bus;

  di_stack = 0;
  di_cpu = 0;
  write_stack = 'b0;
  read_stack = 'b0;
  di_cs_reg = 0;
  w_cs_reg = 'b0;
  cs_reg_addr = 0;
  di_io_reg = 0;
  w_io_reg = 0;
  di_io_port = 0;
  w_io_port = 0;
  io_port_addr = 0;
  di_timer = 0;
  w_timer = 0;
  r_timer = 0;
  timer_addr = 0;
end // case: 4
5 : begin                          // io_port
  di_io_port = do_cpu;
  w_io_port = write_bus;
  io_port_addr = databus_addr[EXREG_ADDR_MSB:0];
  di_cpu = do_io_port;

  di_stack = 0;
  write_stack = 'b0;
  read_stack = 'b0;
  di_cs_reg = 0;
  w_cs_reg = 'b0;
  cs_reg_addr = 0;
  di_io_reg = 0;
  w_io_reg = 0;
```

```
        di_irq = 0;
        w_irq = 'b0;
        di_timer = 0;
        w_timer = 0;
        r_timer = 0;
        timer_addr = 0;
      end // case: 5
      default : begin
        di_stack = 0;
        di_cpu = 0;
        write_stack = 'b0;
        read_stack = 'b0;
        di_cs_reg = 0;
        w_cs_reg = 'b0;
        cs_reg_addr = 0;
        di_io_reg = 0;
        w_io_reg = 0;
        di_irq = 0;
        w_irq = 'b0;
        di_io_port = 0;
        w_io_port = 0;
        io_port_addr = 0;
        di_timer = 0;
        w_timer = 0;
        r_timer = 0;
        timer_addr = 0;
      end // case: default
    endcase // case (addr_in[5:3])
  end // always@ (*)
endmodule // databus
```

C.7 Data Stack

```
module stack (/*AUTOARG*/
  // Outputs
  do_stack,
  // Inputs
  clk, rst_n, write_stack, read_stack, di_stack, scan_test_mode
  );

  `include "cpu_params.sv"

  input clk, rst_n, write_stack, read_stack;
  input [DATA_MSB:0] di_stack;
  input scan_test_mode;
  output [DATA_MSB:0]  do_stack;

  reg [DATA_MSB:0] do_stack;
  reg [8:0] sram_addr;
  reg [31:0] sram_di;
  reg [3:0] sram_wem;
  reg sram_we;
  wire [31:0] tmp_do;
  reg [31:0] tmp_di;
  reg [8:0] p1;          // pointer size given by 1/4 stack size 2^9=512=STACK_SIZE
  reg [1:0] p2;          // pointer counting bytes in each word
  wire clk_n;

  assign clk_n = ~clk;

  always@(posedge clk or  negedge rst_n) begin // pointer counters
    if (!rst_n) begin
      p1 <= 0;
      p2 <= 0;
    end
    else if (write_stack == 1) begin
```

```
    if (p2 < 'd3) begin
      p2 <= p2+1;
    end
    else if (p1 < 'd511) begin
      p1 <= p1+1;
      p2 <= 0;
    end
    else begin
      p1 <= 0;
      p2 <= 0;
    end
  end // if (write_stack == 1)
  else if (read_stack == 1) begin
    if (p2 > 'd0) begin
      p2 <= p2-1;
    end
    else if (p1 > 'd0) begin
      p1 <= p1-1;
      p2 <= 'd3;
    end
    else begin
      p1 <= 'd511;
      p2 <= 'd3;
    end
  end // if (read_stack == 1)
  else begin
    p1 <= p1;
    p2 <= p2;
  end // else: !if(read_stack == 1)
end // always@ (posedge clk or  negedge rst_n)

always@(*) begin
  sram_addr = 0;
  sram_di = 0;
  sram_wem = 0;
  sram_we = 0;
  do_stack = 0;
  if (write_stack) begin            // write to sram
    sram_we = 1;
    tmp_di = 0;
    sram_wem = 0;
    sram_wem = 1 << p2;
    sram_addr = p1;
    tmp_di[p2*8 +: 8] = di_stack;
    sram_di = tmp_di;
    sram_we = write_stack;
  end
  else if (read_stack) begin        // read sram
    sram_we = 0;
    sram_wem = 0;
    if (p2 > 0) begin
      sram_addr = p1;
      do_stack = scan_test_mode ? di_stack : tmp_do[(p2-1)*8 +: 8];
    end
    else  begin
      sram_addr = p1-1;
      do_stack = scan_test_mode ? di_stack : tmp_do[(p2+3)*8 +: 8];
    end
  end // if (read_stack)
  else begin                        // Give 32-bit  instruction
    sram_addr = 0;
    sram_di = 0;
    sram_wem = 0;
    sram_we = 0;
    do_stack = 0;
  end // else: !if(read_stack)
end // always@ (*)

//SRAM block
sram_512x32cm4sw8 U_SRAM_STACK(  .\do (tmp_do),
                          .addr (sram_addr),
```

```
                              .di (sram_di),
                              .wem (sram_wem),
                              .we (sram_we),
                              .oe (1'b1),
                              .me (1'b1),
                              .clk (clk_n),
                              .awt (1'b0),
                              .taddr (9'h0),
                              .tdi (32'h0),
                              .twem (4'h0),
                              .twe (1'b0),
                              .toe (1'b0),
                              .tme (1'b0),
                              .biste (1'b0),
                              .test1 (1'b0),
                              .rm (4'hF));

endmodule // stack
```

C.8 Control and Status Registers

```
module cs_reg (/*AUTOARG*/
  // Outputs
  do_cs_reg, prdata_cs, cpu_reset_n, cpu_disable_n,
  // Inputs
  clk, rst_n, w_cs_reg, cs_reg_addr, di_cs_reg, paddr, pwrite, psel,
  penable_cs, pwdata
  );

  `include "cpu_params.sv"

  input clk, rst_n, w_cs_reg;
  input [EXREG_ADDR_MSB:0] cs_reg_addr;
  input [DATA_MSB:0] di_cs_reg;
  input [31:0] paddr;
  input pwrite, psel, penable_cs;
  input [DATA_MSB:0] pwdata;

  output [DATA_MSB:0]  do_cs_reg;
  output [31:0] prdata_cs;
  output cpu_reset_n, cpu_disable_n;

  wire [DATA_MSB:0] do_cs_reg;
  reg [DATA_MSB:0]  reg_r[CSREG_MSB:0];
  integer i;

  always@(posedge clk or  negedge rst_n) begin
    if (!rst_n) begin
      for (i = 0; i < (CSREG_MSB+1); i=i+1) begin
        reg_r[i] <= 'h00;
      end
    end
    else begin  // Write to different register at same time is possibe
      if (paddr == 'd0 && pwrite && psel && penable_cs) begin // AMBA first priority
        reg_r[0] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h0) begin
        reg_r[0] <= di_cs_reg;
      end
      if (paddr == 'h1 && pwrite && psel && penable_cs) begin
        reg_r[1] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h1) begin
        reg_r[1] <= di_cs_reg;
      end
```

```
      if (paddr == 'h2 && pwrite && psel && penable_cs) begin
        reg_r[2] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h2) begin
        reg_r[2] <= di_cs_reg;
      end
      if (paddr == 'h3 && pwrite && psel && penable_cs) begin
        reg_r[3] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h3) begin
        reg_r[3] <= di_cs_reg;
      end
      if (paddr == 'h4 && pwrite && psel && penable_cs) begin
        reg_r[4] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h4) begin
        reg_r[4] <= di_cs_reg;
      end
      if (paddr == 'h5 && pwrite && psel && penable_cs) begin
        reg_r[5] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h5) begin
        reg_r[5] <= di_cs_reg;
      end
      if (paddr == 'h6 && pwrite && psel && penable_cs) begin
        reg_r[6] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h6) begin
        reg_r[6] <= di_cs_reg;
      end
      if (paddr == 'h7 && pwrite && psel && penable_cs) begin
        reg_r[7] <= pwdata;
      end
      else if (w_cs_reg && cs_reg_addr == 'h7) begin
        reg_r[7] <= di_cs_reg;
      end
    end // else: !if(!rst_n)
  end // always@ (posedge clk)

  assign do_cs_reg = reg_r[cs_reg_addr];
  assign prdata_cs[31:DATA_SIZE] = 0;
  assign prdata_cs[DATA_MSB:0] = reg_r[paddr];
  assign cpu_reset_n = reg_r[0][0];     // first bit of reg_r[0] will reset cpu
  assign cpu_disable_n = reg_r[0][1];  // second bit of reg_r[0] will hold cpu
endmodule // cs_reg
```

C.9 Data In/Out Registers

```
module io_reg (/*AUTOARG*/
  // Outputs
  do_io_reg, prdata_io,
  // Inputs
  clk, rst_n, w_io_reg, di_io_reg, pwrite, psel, penable_io, pwdata
  );

  `include "cpu_params.sv"

  input clk, rst_n, w_io_reg;
  input [DATA_MSB:0]  di_io_reg;
  input pwrite, psel, penable_io;
  input [DATA_MSB:0] pwdata;
  output [DATA_MSB:0]  do_io_reg;
  output [31:0] prdata_io;

  wire [DATA_MSB:0] do_io_reg;
  reg [DATA_MSB:0]  data_in_r, data_out_r;
```

```
  always@(posedge clk or  negedge rst_n) begin
    if (!rst_n) begin
      data_in_r <= 0;
      data_out_r <= 0;
    end
    else begin
      if (pwrite && psel & penable_io) begin
        data_in_r <= pwdata;
      end
      if (w_io_reg) begin
        data_out_r <= di_io_reg;
      end
    end // else: !if(!rst_n)
  end // always@ (posedge clk or  negedge rst_n)

  assign do_io_reg = data_in_r;
  assign prdata_io[31:DATA_SIZE] = 0;
  assign prdata_io[DATA_MSB:0] = data_out_r;
endmodule // io_reg
```

C.10 IRQ Line

```
module irq_line (/*AUTOARG*/
  // Outputs
  irq,
  // Inputs
  clk, rst_n, w_irq, di_irq, paddr, pwrite, psel, penable_irq, pwdata
  );

`include "cpu_params.sv"

  input clk, rst_n, w_irq;
  input [DATA_MSB:0] di_irq;
  input [31:0] paddr;
  input pwrite, psel, penable_irq;
  input [DATA_MSB:0] pwdata;
  output irq;

  reg irq;

  always@(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      irq <= 0;
    end
    else if (paddr == 'h0 && pwrite && psel && penable_irq && w_irq) begin
      irq <= (di_irq[0] || pwdata[0]);
    end
    else if (paddr == 'h0 && pwrite && psel && penable_irq) begin
      irq <= pwdata[0];
    end
    else if (w_irq) begin
      irq <= di_irq[0];
    end
    else begin
      irq <= irq;
    end
  end // always@ (posedge clk or negedge rst_n)
endmodule // irq_line
```

C.11 Timer/Counter

```
module timer_counter (/*AUTOARG*/
  // Outputs
  do_timer,
  // Inputs
  clk, rst_n, w_timer, r_timer, timer_addr, di_timer
  );

`include "cpu_params.sv"

  input clk, rst_n, w_timer, r_timer;
  input [EXREG_ADDR_MSB:0] timer_addr;
  input [DATA_MSB:0] di_timer;
  output [DATA_MSB:0] do_timer;

  reg [15:0] counter_r, compare_r;
  reg [DATA_MSB:0] c_r;         // control and status reg, c_r[7] is compare flag
  reg set_flag, clear_flag;
  reg [DATA_MSB:0] do_timer;

  always@(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      counter_r <= 0;
      set_flag <= 0;
    end
    else if (c_r[0]) begin            // if counter is on
      if (!c_r[1]) begin              // if reset counter
        counter_r <= 0;
        set_flag <= 0;
      end
      else if (!c_r[2]) begin         // if not reset counter and compare off
        set_flag <= 0;
        if (counter_r < 'hffff) begin
          counter_r <= counter_r+1;
        end
        else begin
          counter_r <= 0;
        end
      end
      else begin
        if (counter_r < compare_r-1) begin
          counter_r <= counter_r+1;
          set_flag <= 0;
        end
        else if (counter_r == compare_r-1) begin
          counter_r <= counter_r+1;
          if (c_r[2]) begin
            set_flag <= 1;
          end
        end
        else if (counter_r == compare_r) begin
          counter_r <= 0;
        end
        else begin
          counter_r <= 0;
          set_flag <= 0;
        end
      end // else: !if(!c_r[1])
    end // if (c_r[0])
    else begin
      counter_r <= counter_r;
    end // else: !if(c_r[0])
  end // always@ (posedge clk or negedge rst_n)

  always@ (posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      c_r[6:0] <= 0;
      compare_r <= 0;
    end
```

```
    else if (w_timer) begin
      if (timer_addr == 'h2) begin
        compare_r[DATA_MSB:0] <= di_timer;
      end
      else if (timer_addr == 'h3) begin
        compare_r[15:DATA_SIZE] <= di_timer;
      end
      else if (timer_addr == 'h4) begin
        c_r[6:0] <= di_timer[6:0];
      end
      else begin
        compare_r <= compare_r;
        c_r[6:0] <= c_r[6:0];
      end
    end // if (w_timer)
    else begin
      if (!c_r[1]) begin
        c_r[0] <= c_r[0];
        c_r[1] <= 1;                  // to remove the rset automatically
        c_r[2] <= c_r[2];
        compare_r <= compare_r;
      end
      else begin
        c_r[6:0] <= c_r[6:0];
        compare_r <= compare_r;
      end
    end // else: !if(w_timer)
  end // always@ (posedge clk or negedge rst_n)

  always@(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      c_r[7] <= 0;
    end
    else if (clear_flag) begin
      c_r[7] <= 0;
    end
    else if(set_flag) begin
      c_r[7] <= 1;
    end
    else begin
      c_r[7] <= c_r[7];
    end
  end // always@ (posedge clk or negedge rst_n)

  always@(*) begin
    clear_flag = 0;
    if (timer_addr == 'h0) begin
      do_timer = counter_r[DATA_MSB:0];
      clear_flag = 0;
    end
    else if (timer_addr == 'h1) begin
      do_timer = counter_r[15:DATA_SIZE];
      clear_flag = 0;
    end
    else if (timer_addr == 'h2) begin
      do_timer = compare_r[DATA_MSB:0];
      clear_flag = 0;
    end
    else if (timer_addr == 'h3) begin
      do_timer = compare_r[15:DATA_SIZE];
      clear_flag = 0;
    end
    else if (timer_addr == 'h4) begin
      do_timer = c_r;
      clear_flag = 0;
    end
    else if (timer_addr == 'h5 && r_timer) begin
      do_timer = c_r;
      clear_flag = 1;
    end
    else begin
```

```
      clear_flag = 0;
      do_timer = 0;
    end
  end // always@ (*)
endmodule // timer_counter
```

C.12 Input/Output Port

```
module io_port (/*AUTOARG*/
  // Outputs
  do_io_port, OUT, OE, PD, PU,
  // Inputs
  clk, rst_n, w_io_port, io_port_addr, di_io_port, IN
  );

`include "cpu_params.sv"

  input clk, rst_n, w_io_port;
  input [EXREG_ADDR_MSB:0] io_port_addr; // extend to 4 bit
  input [DATA_MSB:0] di_io_port, IN;
  output [DATA_MSB:0] do_io_port;
  output [DATA_MSB:0] OUT, OE, PD, PU;

  reg [DATA_MSB:0]      OE;
  reg [DATA_MSB:0]      OUT;
  reg [DATA_MSB:0]      PD;
  reg [DATA_MSB:0]      PU;
  wire [DATA_MSB:0]      do_io_port;

  assign do_io_port = IN;

  always@(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      //do_io_port <= 0;
      OUT <= 0;
      OE <= 0;
      PU <= 0;
      PD <= 0;
    end
    else begin
    case(io_port_addr[1:0])
      CLEAR : begin
        //do_io_port <= IN;
        if (w_io_port) begin
          if (io_port_addr[3:2] == 00) begin
            OUT <= OUT & di_io_port;
          end
          else if (io_port_addr[3:2] == 01) begin
            OE <= OE & di_io_port;
          end
          else if (io_port_addr[3:2] == 10) begin
            PU <= PU & di_io_port;
          end
          else if (io_port_addr[3:2] == 11) begin
            PD <= PD & di_io_port;
          end
        end // if (w_oi_port)
        else begin
          OUT <= OUT;
          OE <= OE;
          PU <= PU;
          PD <= PD;
        end // else: !if(w_io_port)
      end // case: CLEAR
      SET : begin
        //do_io_port <= IN;
```

```
    if (w_io_port) begin
      if (io_port_addr[3:2] == 00) begin
        OUT <= OUT | di_io_port;
      end
      else if (io_port_addr[3:2] == 01) begin
        OE <= OE | di_io_port;
      end
      else if (io_port_addr[3:2] == 10) begin
        PU <= PU | di_io_port;
      end
      else if (io_port_addr[3:2] == 11) begin
        PD <= PD | di_io_port;
      end
    end // if (w_io_port)
    else begin
      OUT <= OUT;
      OE <= OE;
      PU <= PU;
      PD <= PD;
    end // else: !if(w_io_port)
  end // case: SET
  TOGGLE : begin
    //do_io_port <= IN;
    if (w_io_port) begin
      if (io_port_addr[3:2] == 00) begin
        OUT <= OUT ^ di_io_port;
      end
      else if (io_port_addr[3:2] == 01) begin
        OE <= OE ^ di_io_port;
      end
      else if (io_port_addr[3:2] == 10) begin
        PU <= PU ^ di_io_port;
      end
      else if (io_port_addr[3:2] == 11) begin
        PD <= PD ^ di_io_port;
      end
    end // if (w_io_port)
    else begin
      OUT <= OUT;
      OE <= OE;
      PU <= PU;
      PD <= PD;
    end // else: !if(w_io_port)
  end // case: TOGGLE
  FORCE : begin
    if (w_io_port) begin
      if (io_port_addr[3:2] == 00) begin
        OUT <= di_io_port;
      end
      else if (io_port_addr[3:2] == 01) begin
        OE <= di_io_port;
      end
      else if (io_port_addr[3:2] == 10) begin
      PU <= di_io_port;
      end
      else if (io_port_addr[3:2] == 11) begin
        PD <= di_io_port;
      end
    end // if (w_io_port)
    else begin
      OUT <= OUT;
      OE <= OE;
      PU <= PU;
      PD <= PD;
    end // else: !if(w_io_port)
  end // case: READ_OUT
  default : begin
    //do_io_port <= 0;
    OUT <= 0;
    OE <= 0;
    PU <= 0;
```

```
        PD <= 0;
      end
    endcase // case (io_port_addr[1:0]
    end // else: !if(!rst_n)
  end // always@ (*)
endmodule // io_port
```

C.13 Topmodule Core

```
module cpu(/*AUTOARG*/
  // Outputs
  do_cpu, databus_addr, write_bus, read_bus,
  // Inputs
  clk, rst_n, di_cpu, instr_addr, instr_data, write_mem, cpu_disable_n,
  cpu_reset_n, scan_test_mode
  );

`include "cpu_params.sv"

  input clk, rst_n;
  input [DATA_MSB:0] di_cpu;
  input [8:0] instr_addr;
  input [31:0] instr_data;
  input write_mem, cpu_disable_n, cpu_reset_n;
  input scan_test_mode;
  output [DATA_MSB:0] do_cpu;
  output [BUSADDR_MSB:0] databus_addr;
  output write_bus, read_bus;

  wire [DATA_MSB:0] data_a, data_b, data_d, constant;
  wire [ADDR_MSB:0] addr_a, addr_b, addr_d;
  wire write;
  wire [1:0] ms;
  wire [31:0] di_ireg;
  wire [MSB_FS:0] fs;
  wire v_r, c_r, n_r, z_r, pl, jb, bset;
  wire [2:0] bc;
  wire [PC_MSB:0] pc_r;
  wire [DATA_MSB:0] do_cpu;
  wire [BUSADDR_MSB:0] databus_addr;
  wire write_bus, read_bus;
  wire [6:0] offset;
  wire instr_pntr;
  wire done_hold, hold_pc;
  wire msd;

  register R1(/*AUTOINST*/
              // Outputs
              .data_a                   (data_a[DATA_MSB:0]),
              .data_b                   (data_b[DATA_MSB:0]),
              // Inputs
              .write                    (write),
              .clk                      (clk),
              .rst_n                    (rst_n),
              .addr_a                   (addr_a[ADDR_MSB:0]),
              .addr_b                   (addr_b[ADDR_MSB:0]),
              .addr_d                   (addr_d[ADDR_MSB:0]),
              .data_d                   (data_d[DATA_MSB:0]));

  alu A1(/*AUTOINST*/
         // Outputs
         .data_d                        (data_d[DATA_MSB:0]),
         .v_r                           (v_r),
         .c_r                           (c_r),
         .n_r                           (n_r),
         .z_r                           (z_r),
```

```
        .do_cpu                         (do_cpu[DATA_MSB:0]),
        // Inputs
        .clk                            (clk),
        .rst_n                          (rst_n),
        .ms                             (ms[1:0]),
        .data_a                         (data_a[DATA_MSB:0]),
        .data_b                         (data_b[DATA_MSB:0]),
        .di_cpu                         (di_cpu[DATA_MSB:0]),
        .constant                       (constant[DATA_MSB:0]),
        .fs                             (fs[MSB_FS:0]));

decode D1(/*AUTOINST*/
          // Outputs
          .addr_a                       (addr_a[ADDR_MSB:0]),
          .addr_b                       (addr_b[ADDR_MSB:0]),
          .addr_d                       (addr_d[ADDR_MSB:0]),
          .constant                     (constant[DATA_MSB:0]),
          .write                        (write),
          .pl                           (pl),
          .jb                           (jb),
          .bset                         (bset),
          .hold_pc                      (hold_pc),
          .bc                           (bc[2:0]),
          .fs                           (fs[MSB_FS:0]),
          .ms                           (ms[MUXA_MSB:0]),
          .databus_addr                 (databus_addr[BUSADDR_MSB:0]),
          .write_bus                    (write_bus),
          .read_bus                     (read_bus),
          .offset                       (offset[6:0]),
          // Inputs
          .instr_pntr                   (instr_pntr),
          .done_hold                    (done_hold),
          .di_ireg                      (di_ireg[31:0]),
          .cpu_reset_n                  (cpu_reset_n),
          .cpu_disable_n                (cpu_disable_n),
          .msd                          (msd));

prog_mem PM(/*AUTOINST*/
            // Outputs
            .di_ireg                    (di_ireg[31:0]),
            .instr_pntr                 (instr_pntr),
            // Inputs
            .clk                        (clk),
            .write_mem                  (write_mem),
            .pc_r                       (pc_r[PC_MSB:0]),
            .instr_addr                 (instr_addr[8:0]),
            .instr_data                 (instr_data[31:0]),
            .scan_test_mode             (scan_test_mode));

/* brctrl AUTO_TEMPLATE (
                        .jmp_addr (di_ireg[PC_MSB:0]),
 );
 */
brctrl B1(/*AUTOINST*/
          // Outputs
          .pc_r                         (pc_r[PC_MSB:0]),
          .done_hold                    (done_hold),
          .msd                          (msd),
          // Inputs
          .clk                          (clk),
          .rst_n                        (rst_n),
          .v_r                          (v_r),
          .c_r                          (c_r),
          .n_r                          (n_r),
          .z_r                          (z_r),
          .pl                           (pl),
          .jb                           (jb),
          .bset                         (bset),
          .bc                           (bc[2:0]),
          .hold_pc                      (hold_pc),
          .cpu_reset_n                  (cpu_reset_n),
```

```
            .offset                     (offset[6:0]),
            .jmp_addr                   (di_ireg[PC_MSB:0]));     // Templated
endmodule // cpu
```

C.14 Topmodule Peripheral Processor

```
module top (/*AUTOARG*/
  // Outputs
  prdata_io,
  prdata_cs,
  irq, OE, OUT,
  PU, PD,
  // Inputs
  clk, rst_n,
  write_mem,
  instr_addr,
  instr_data,
  paddr,
  penable_io,
  penable_cs,
  penable_irq,
  psel, pwrite,
  pwdata, IN
  );

`include "cpu_params.sv"

  input clk, rst_n, write_mem;
  input [8:0] instr_addr;
  input [31:0] instr_data;
  input [31:0] paddr;
  input penable_io, penable_cs, penable_irq, psel, pwrite;
  input [DATA_MSB:0] pwdata, IN;
  output [31:0] prdata_io, prdata_cs;
  output irq;
  output [DATA_MSB:0] OE, OUT, PU, PD;

  wire [DATA_MSB:0] do_cpu, di_cpu;
  wire [DATA_MSB:0] do_stack, di_stack;
  wire [BUSADDR_MSB:0] databus_addr;
  wire write_bus, read_bus;
  wire write_stack, read_stack;
  wire w_cs_reg, w_io_reg;
  wire [EXREG_ADDR_MSB:0] cs_reg_addr;
  wire [DATA_MSB:0] di_cs_reg, do_cs_reg, di_io_reg, do_io_reg;
  wire cpu_disable_n, cpu_reset_n;
  wire [DATA_MSB:0] di_irq;
  wire w_irq;
  wire [DATA_MSB:0]     di_io_port;             // From BUS1 of databus.v
  wire [DATA_MSB:0]     di_timer;               // From BUS1 of databus.v
  wire [DATA_MSB:0]     do_io_port;             // From IO_PORT of io_port.v
  wire [DATA_MSB:0]     do_timer;               // From TIMER of timer_counter.v
  wire [EXREG_ADDR_MSB:0] io_port_addr;         // From BUS1 of databus.v
  wire                  r_timer;                // From BUS1 of databus.v
  wire [EXREG_ADDR_MSB:0] timer_addr;           // From BUS1 of databus.v
  wire                  w_io_port;              // From BUS1 of databus.v
  wire                  w_timer;                // From BUS1 of databus.v

  cpu C1(/*AUTOINST*/
         // Outputs
         .do_cpu                        (do_cpu[DATA_MSB:0]),
         .databus_addr                  (databus_addr[BUSADDR_MSB:0]),
         .write_bus                     (write_bus),
         .read_bus                      (read_bus),
         // Inputs
         .clk                           (clk),
```

```
        .rst_n                          (rst_n),
        .di_cpu                         (di_cpu[DATA_MSB:0]),
        .instr_addr                     (instr_addr[8:0]),
        .instr_data                     (instr_data[31:0]),
        .write_mem                      (write_mem),
        .cpu_disable_n                  (cpu_disable_n),
        .cpu_reset_n                    (cpu_reset_n));

databus BUS1(/*AUTOINST*/
             // Outputs
             .write_stack               (write_stack),
             .read_stack                (read_stack),
             .w_cs_reg                  (w_cs_reg),
             .w_io_reg                  (w_io_reg),
             .w_irq                     (w_irq),
             .cs_reg_addr               (cs_reg_addr[EXREG_ADDR_MSB:0]),
             .di_cs_reg                 (di_cs_reg[DATA_MSB:0]),
             .di_io_reg                 (di_io_reg[DATA_MSB:0]),
             .di_stack                  (di_stack[DATA_MSB:0]),
             .di_cpu                    (di_cpu[DATA_MSB:0]),
             .di_irq                    (di_irq[DATA_MSB:0]),
             .di_timer                  (di_timer[DATA_MSB:0]),
             .di_io_port                (di_io_port[DATA_MSB:0]),
             .w_io_port                 (w_io_port),
             .io_port_addr              (io_port_addr[EXREG_ADDR_MSB:0]),
             .timer_addr                (timer_addr[EXREG_ADDR_MSB:0]),
             .r_timer                   (r_timer),
             .w_timer                   (w_timer),
             // Inputs
             .write_bus                 (write_bus),
             .read_bus                  (read_bus),
             .do_cpu                    (do_cpu[DATA_MSB:0]),
             .do_stack                  (do_stack[DATA_MSB:0]),
             .databus_addr              (databus_addr[BUSADDR_MSB:0]),
             .do_cs_reg                 (do_cs_reg[DATA_MSB:0]),
             .do_io_reg                 (do_io_reg[DATA_MSB:0]),
             .do_io_port                (do_io_port[DATA_MSB:0]),
             .do_timer                  (do_timer[DATA_MSB:0]));

stack STACK1(/*AUTOINST*/
             // Outputs
             .do_stack                  (do_stack[DATA_MSB:0]),
             // Inputs
             .clk                       (clk),
             .rst_n                     (rst_n),
             .write_stack               (write_stack),
             .read_stack                (read_stack),
             .di_stack                  (di_stack[DATA_MSB:0]));

cs_reg CS1(/*AUTOINST*/
           // Outputs
           .do_cs_reg                   (do_cs_reg[DATA_MSB:0]),
           .prdata_cs                   (prdata_cs[31:0]),
           .cpu_reset_n                 (cpu_reset_n),
           .cpu_disable_n               (cpu_disable_n),
           // Inputs
           .clk                         (clk),
           .rst_n                       (rst_n),
           .w_cs_reg                    (w_cs_reg),
           .cs_reg_addr                 (cs_reg_addr[EXREG_ADDR_MSB:0]),
           .di_cs_reg                   (di_cs_reg[DATA_MSB:0]),
           .paddr                       (paddr[31:0]),
           .pwrite                      (pwrite),
           .psel                        (psel),
           .penable_cs                  (penable_cs),
           .pwdata                      (pwdata[DATA_MSB:0]));

io_reg IO1(/*AUTOINST*/
           // Outputs
           .do_io_reg                   (do_io_reg[DATA_MSB:0]),
           .prdata_io                   (prdata_io[31:0]),
```

```
            // Inputs
            .clk                        (clk),
            .rst_n                      (rst_n),
            .w_io_reg                   (w_io_reg),
            .di_io_reg                  (di_io_reg[DATA_MSB:0]),
            .pwrite                     (pwrite),
            .psel                       (psel),
            .penable_io                 (penable_io),
            .pwdata                     (pwdata[DATA_MSB:0]));

  irq_line IRQ1(/*AUTOINST*/
                // Outputs
                .irq                    (irq),
                // Inputs
                .clk                    (clk),
                .rst_n                  (rst_n),
                .w_irq                  (w_irq),
                .di_irq                 (di_irq[DATA_MSB:0]),
                .paddr                  (paddr[31:0]),
                .pwrite                 (pwrite),
                .psel                   (psel),
                .penable_irq            (penable_irq),
                .pwdata                 (pwdata[DATA_MSB:0]));

  timer_counter TIMER(/*AUTOINST*/
                      // Outputs
                      .do_timer         (do_timer[DATA_MSB:0]),
                      // Inputs
                      .clk              (clk),
                      .rst_n            (rst_n),
                      .w_timer          (w_timer),
                      .r_timer          (r_timer),
                      .timer_addr       (timer_addr[EXREG_ADDR_MSB:0]),
                      .di_timer         (di_timer[DATA_MSB:0]));

  io_port IO_PORT(/*AUTOINST*/
                  // Outputs
                  .do_io_port           (do_io_port[DATA_MSB:0]),
                  .OUT                  (OUT[DATA_MSB:0]),
                  .OE                   (OE[DATA_MSB:0]),
                  .PD                   (PD[DATA_MSB:0]),
                  .PU                   (PU[DATA_MSB:0]),
                  // Inputs
                  .clk                  (clk),
                  .rst_n                (rst_n),
                  .w_io_port            (w_io_port),
                  .io_port_addr         (io_port_addr[EXREG_ADDR_MSB:0]),
                  .di_io_port           (di_io_port[DATA_MSB:0]),
                  .IN                   (IN[DATA_MSB:0]));
endmodule // top
```

C.15 Parameters

```
//register and alu parameters
parameter ADDR_BITS = 4;    // Number of bits in data address to register file
parameter ADDR_MSB  = ADDR_BITS-1;
parameter DATA_SIZE = 8;              // Size of the registers
parameter DATA_MSB = DATA_SIZE-1;
parameter REG_NUMBERS = 15;           // REG_NUMBERS-1
parameter FS_BITS = 6;                // Number of bits in Function Select
parameter MSB_FS = FS_BITS-1;
parameter MUXA_MSB = 1;

//ALU function parameters
parameter NOP=6'b000000, ADD=6'b000001, ADC=6'b000010, SUB=6'b000011, SBC=6'b000100,
  LDI=6'b000101,
  LSL=6'b000110, LSR=6'b000111, ROL=6'b001000, ROR=6'b001001, AND=6'b001010,
  OR=6'b001011, XOR=6'b001100, CP=6'b001101, CPC=6'b001110, MOV=6'b001111,
  DEC=6'b010000, INC=6'b010001;

//Instruction decoder
parameter OPCODE_SIZE = 16;
parameter OPCODE_MSB = OPCODE_SIZE-1;
parameter OPC_NOP='b0000000000000000, OPC_ADC='b000111, OPC_ADD='b000011,
  OPC_AND='b001000, OPC_SUB='b000110, OPC_SUBI='b0101, OPC_SBC='b000010,
  OPC_SBCI='b0100, OPC_LDI='b1110, OPC_LSR='b10010100110, OPC_ROR='b10010100111,
  OPC_OR='b001010, OPC_XOR='b001001, OPC_CP='b000101, OPC_CPC='b000001,
  OPC_CPI='b0011, OPC_MOV='b001011, OPC_DEC='b10010101010, OPC_INC='b10010100011,
  OPC_BRCS='b111100000, OPC_BRCC='b111101000, OPC_BRVS='b111100011,
  OPC_BRVC='b111101011, OPC_BRMI='b111100010, OPC_BRPL='b111101010,
  OPC_BREQ='b111100001, OPC_BRNE='b111101001, OPC_PUSH='b10010011111,
  OPC_POP='b10010001111, OPC_STS='b10101, OPC_LDS='b10100,
  OPC_JMP='b1001010000001100;

parameter PC_SIZE = 10;
parameter PC_MSB = PC_SIZE-1;
parameter MEM_SIZE = 31;
parameter BUSADDR_SIZE = 7;
parameter BUSADDR_MSB = BUSADDR_SIZE-1;
parameter EXREG_ADDR_SIZE = 4;
parameter EXREG_ADDR_MSB = EXREG_ADDR_SIZE-1;
parameter STACK_SIZE = 15;
parameter BUFFER_SIZE = 4;
parameter BUFFER_MSB = BUFFER_SIZE-1;
parameter CSREG_SIZE = 8;
parameter CSREG_MSB = CSREG_SIZE-1;

//io_port
parameter CLEAR=2'b00, SET=2'b01, TOGGLE=2'b10, FORCE=2'b11;
```

Bibliography

[1] J. E. Thornton. Design of a computer the control data 6600, 1970.

[2] Atmel. 8-bit atmel microcontoller with 128kbytes in-system programmable flash. 2011.

[3] ARM. Amba spesifications. 1999.

[4] Anders Ruden. Programmable microcontroller peripherals, 2012.

[5] Stellaris lm4f252h5qd microcontroller. 2012.

[6] Micro Energy. Efm32g refrence manual. 2012.

[7] Mary Bellis. Seymour cray - cray supercomputers, 2012.

[8] QuickLogic. Universal asynchronous receiver/transmitter (uart) data sheet. 2008.

[9] Atmel. Avr instruction set. 2010.