Examensarbete
Design of Application Specific Microcontroller

Kristoffer Martinsson
LITH-ISY-EX-3283-2002
Linköping 2002-11-28
Design of Application Specific Microcontroller

Examensarbete utfört i datorteknik
vid Linköpings tekniska högskola
av
Kristoffer Martinsson

LiTH-ISY-EX-3283-2002

Handledare: Tomas Henriksson
Examinator: Dake Liu
Linköping den 28 november 2002
<table>
<thead>
<tr>
<th>Språk</th>
<th>Rapporttyp</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Svenska/Swedish</td>
<td>Licentiatavhandling</td>
<td>ISRN LITH-ISY-EX-3283-2002</td>
</tr>
<tr>
<td>X Engelska/English</td>
<td>Examensarbete</td>
<td></td>
</tr>
<tr>
<td>C-uppsats</td>
<td>D-uppsats</td>
<td></td>
</tr>
<tr>
<td>Övrig rapport</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Titel</th>
<th>Författare</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design av applikationsspecifik mikrocontroller</td>
<td>Kristoffer Martinsson</td>
</tr>
</tbody>
</table>

Sammanfattning

This master thesis describes the process of designing an application specific microcontroller. The microcontroller should be used in a demonstrator for a protocol processor. The demonstrator should show the possibilities to access high speed networks with small processor cores.

The demonstrator should be able to receive and playback an audio stream. Some of the tasks in the demonstrator should be performed by the microcontroller. It should handle ARP requests, buffer handling and sending audio samples to a stereo codec. Behavioral models for these applications were constructed and used to design the instruction set for the microcontroller.

An instruction set simulator was constructed. It was used to verify that the instruction set was sufficient to achieve functional coverage.

The micro architecture for the microcontroller was designed and implemented in VHDL. This implementation was verified by simulation. The test vectors used during simulation were mainly randomly generated.

Nyckelord

applikationsspecifik, mikrokontroller, application, specific, microcontroller
# Table of contents

**CHAPTER 1, INTRODUCTION**

1.1 BACKGROUND 1
1.2 PURPOSE 1
1.3 REQUIREMENTS 1
1.4 ORDER OF WORK 2

**CHAPTER 2, DEMONSTRATOR**

2.1 OVERVIEW 3
  2.1.1 MII -> XGMII CONVERTER 4
  2.1.2 ETHERNET MAC TRANSMISSION 4
  2.1.3 PLAYBACK CONTROLLER 4
  2.1.4 MEMORY ARBITER 4
2.2 MICROCONTROLLER INTERFACES 4
2.3 FUNCTIONAL DESCRIPTION 6

**CHAPTER 3, DESIGN OF INSTRUCTION SET**

3.1 GENERAL ASPECTS 7
  3.1.1 FUNCTIONAL COVERAGE 7
  3.1.2 PERFORMANCE DEMANDS 7
  3.1.3 ORTHOGONAL INSTRUCTION SET 8
  3.1.4 GENERAL INSTRUCTIONS 8
3.2 THIS MICROCONTROLLER 8
  3.2.1 APPLICATIONS 8
  3.2.2 REGISTERS 12
  3.2.3 CHOOSING THE INSTRUCTIONS 12
  3.2.4 INSTRUCTION WORD FORMAT 15
  3.2.5 ADDRESSING MODES 15
# Table Of Contents

### CHAPTER 4, INSTRUCTION SET SIMULATOR

- **4.1 Benefits of Using an ISS**
  
- **4.2 How It Works**
  
- **4.3 Implementation**
  
- **4.4 Verification of Functional Coverage**
  - 4.4.1 Test Case 1
  - 4.4.2 Test Case 2
  - 4.4.3 Test Case 3
  - 4.4.4 Test Case 4
  - 4.4.5 Test Case 5

### CHAPTER 5, DESIGN OF MICRO ARCHITECTURE

- **5.1 Top Level Design**
  - 5.1.1 Method
  - 5.1.2 Architecture Overview

- **5.2 Data Path**
  - 5.2.1 Arithmetic Unit
  - 5.2.2 Logic Unit
  - 5.2.3 Shift Unit
  - 5.2.4 Register File

- **5.3 Control Path**
  - 5.3.1 Instruction Decoder
  - 5.3.2 Program Counter
  - 5.3.3 Program Flow Controller

### CHAPTER 6, VERIFICATION OF MICRO ARCHITECTURE

- **6.1 General Aspects**
  - 6.1.1 Formal Verification
  - 6.1.2 Verification by Simulation

- **6.2 Verification of Microcontroller**
  - 6.2.1 Random Testing
  - 6.2.2 Corner Testing

### CHAPTER 7, RESULTS AND CONCLUSIONS

### REFERENCES
List of Figures

Figure 2.1, Overview of the parts in the demonstrator ........................................... 3
Figure 2.2, the interfaces of the microcontroller ....................................................... 5
Figure 2.3, Packet flow from the desktop computer to the demonstrator ................. 6
Figure 3.1, Description of the buffer, the arrows shows an example of how the pointers could be directed 9
Figure 3.2, Behavioral model of output empty subroutine ........................................ 9
Figure 3.3, Behavioral model of ARP handling routine ............................................. 10
Figure 3.4, Behavioral model of the sort buffer routine .......................................... 11
Figure 3.5, Available addressing modes in the microcontroller ............................... 16
Figure 4.1, Program flow of the instruction set simulator ........................................ 19
Figure 5.1, Overview of the microcontroller with all buses .................................... 26
Figure 5.2, The arithmetic unit ............................................................................... 27
Figure 5.3, The logic unit ...................................................................................... 28
Figure 5.4, The shift unit ....................................................................................... 29
Figure 5.5, The registers in the register file and its inputs ...................................... 31
Figure 5.6, The hardware for a status flag .............................................................. 31
Figure 5.7, The register file and its outputs ............................................................ 32
Figure 5.8, The program counter ......................................................................... 34
Figure 5.9, The program flow controller ............................................................... 34
List of Tables

Table 3.1, All instructions in the microcontroller ................................................................. 14
Table 3.2, Instruction formats in this instruction set ............................................................. 15
Table 4.1, Clock cycle when the packets arrive in test case 1.............................................. 21
Table 4.2, Clock cycle when the packets arrive in test case 2.............................................. 22
Table 4.3, Clock cycle when the packets arrive in test case 3.............................................. 22
Table 4.4, Clock cycle when the packets arrive in test case 5.............................................. 23
List of Tables
Chapter 1, Introduction

This Master Thesis project has been performed in the division of Computer Engineering at the department of Electrical Engineering at Linköping University. A master thesis project should be 20 weeks of fulltime work. It should show that the student could use the knowledge acquired during the education to solve a task related to the education.

1.1 Background

In a research project a demonstrator for a protocol processor should be developed. The demonstrator should show the possibility to use small processor cores to access high speed networks. This demonstrator should be able to receive an audio stream from the network and send the samples to a stereo codec. It should be implemented in a FPGA and work in a fast Ethernet environment. One part of this demonstrator is a microcontroller. The microcontroller should handle some tasks that the protocol processor cannot handle. Typical tasks are buffer handling and sending the audio samples to a stereo codec.

1.2 Purpose

The purpose with this project is to design and implement a special purpose microcontroller that should be used in the demonstrator. The instruction set for the microcontroller should be designed and proven. Benchmarking will not be used to optimize the instruction set since there are no demands on high performance. The interfaces between the microcontroller and the rest of the demonstrator should be designed. A micro architecture should also be designed and it should be implemented in the hardware description language VHDL. Finally, the VHDL description of the microcontroller should be verified.

1.3 Requirements

The requirements for the master thesis project were listed during the start-up of the project. They were divided into primary and secondary demands. The primary requirements must be completed during the project and the secondary if time allows.
**Primary Requirements**

1. Design behavioral models for typical applications (ARP request handling, buffer handling etc)
2. Design the instruction set architecture based on the behavioral models.
3. Construct an instruction set simulator and verify that the instruction set is sufficient to implement typical applications.
4. Design the micro architecture and implement it using VHDL.
5. Verify the implementation.

**Secondary Requirements**

1. Integrate the microcontroller with the demonstrator.
2. Synthesis of the demonstrator to the FPGA.
3. Write program for the microcontroller.

1.4 **Order of work**

The first task of the master thesis project was to find the tasks in the demonstrator that should be handled by the microcontroller. Another important thing that must be decided was how the microcontroller should communicate with the other parts of the demonstrator. These things are described in chapter 2.

The first step in the design process of the microcontroller was to design the instruction set. The starting point of this design step was the behavioral models of the applications that the microcontroller should be able to run. The design of the instruction set is described in chapter 3.

The next step in the design process was to prove the instruction set. To be able to do this an instruction set simulator was constructed. The applications written in assembly code were executed on the instruction set simulator. This is described in chapter 4.

After the instruction set simulator the micro architecture should be designed. The starting point when designing the micro architecture was the instruction set. The architecture must be able to execute all instructions. The architecture should also be implemented in VHDL. This design step is described in chapter 5.

The last step in the design of the microcontroller is to verify the implementation. This was done by simulation. The verification is described in chapter 6.

Finally, chapter 7 contains results and conclusions of the master thesis project.
Chapter 2, Demonstrator

This chapter gives an overview of the demonstrator. It also describes how the microcontroller communicates with the rest of the components in the demonstrator.

2.1 Overview

The demonstrator should be implemented in a Xilinx Virtex FPGA on a XSV-300 board. In addition to the FPGA, the XSV-300 board consists of two SRAM banks, the Ethernet PHY and the stereo codec. There are also other circuitries on the board but these are not used in the demonstrator. For more information, see the reference manual [3].

In addition to the microcontroller and the protocol processor there are some other hardware that must be implemented in the FPGA, namely the MII -> XGMII converter, an Ethernet MAC Transmission, a Playback Controller and a Memory Arbiter. The protocol processor is not described in this document, but information about it could be found in [7].

![Figure 2.1, Overview of the parts in the demonstrator](image_url)
2.1.1 MII -> XGMII converter
The protocol processor is designed to operate in a Gigabit or 10 Gigabit Ethernet. Since the demonstrator should operate in a Fast Ethernet, the interface must be converted to the correct format. This is done by the MII -> XGMII converter.

2.1.2 Ethernet MAC Transmission
The microcontroller should handle ARP requests. When an ARP request arrives, it should create an ARP packet and send it back to the sending host. When the microcontroller starts to send the packet it just writes to a 16-bit special purpose register and set a flag. The Ethernet MAC Transmission sends the packet to the Ethernet PHY and it creates some parts of the Ethernet header, for example the CRC.

2.1.3 Playback Controller
When the microcontroller sends a sample to the stereo codec it just writes to a 16-bit special purpose register and set a flag. The playback controller reads from this register, serializes the samples, and sends it to the stereo codec using a shift register. It also generates control signals to the stereo codec and the microcontroller.

2.1.4 Memory Arbiter
The data memory is used for synchronization between the protocol processor and the microcontroller. This means that they must both access the data memory. The memory arbiter is responsible for handling the access to the data memory. The memory arbiter use handshaking with both the microcontroller and the protocol processor. It is possible that the microcontroller and the protocol processor request a memory access at the same time and normally the protocol processor has higher priority. However, the memory arbiter could be configured by the microcontroller to prevent the protocol processor to use the memory. The configuration is done through a memory mapped register and it is used when the microcontroller cannot accept any delays.

2.2 Microcontroller Interfaces
The microcontroller has four different interfaces, to the program memory, the data memory, the stereo codec and the ARP interface, see figure 2.2. There are also five general control signals, which could be used by external hardware to control the microcontroller.

The interface to the program memory consists of the address bus and the data bus. The address bus is 19 bits wide and the data bus is 16 bits wide. The microcontroller could only read from the program memory, which means that it cannot be used by the microcontroller to store data.
The interface to the data memory goes through the memory arbiter. It consists of the address and data bus and three control signals. These control signals are read, write and halt. The address bus is 20 bits wide and the data bus is 16 bits wide. The data memory is only $2^{19}$ words, the extra bit is used so that memory mapped registers could be addressed. Handshaking is used between the microcontroller and the memory arbiter when the microcontroller would like to access the memory. When the microcontroller would like to access the memory it sets either read or write signal high. It also applies the address and data, if it is write instruction. If the memory arbiter sets the halt signal high, the microcontroller must wait until the halt signal goes low again.

The interface to the stereo codec goes through the playback controller. It consists of a 16 bits wide data signal, which is directly connected to a special purpose register, and two control signals. When the microcontroller writes to the 16 bits special purpose register, it sets the enable codec signal high. The playback controller is also connected to one of the general control signals namely F3. When the playback controller read the data in this special purpose register, it sets F3 high and it remains high until the microcontroller writes to the register.

The ARP interface is similar to the stereo codec interface. It consists of a 16 bits wide data signal, which is directly connected to a special purpose register, and one control signal. When the microcontroller writes to the special purpose register it sets the enable ARP signal high.

The protocol processor is also directly connected to the microcontroller through two control signals namely F1 and F2. These two signals are used by the protocol processor to inform the microcontroller that a new packet has been written to the memory. F1 is high when an ARP request packet has been written to memory and F2 is high when a new audio packet has been written to memory.

The control signals F4 and F5 are not used in the demonstrator but could be used for other applications.

Demonstrator Overview
2.3 Functional Description

The first thing that happens when the demonstrator starts is that the protocol processor is configured by the microcontroller. The configuration data is stored in the program for the microcontroller. This configuration subroutine writes to memory mapped register in the protocol processor.

The demonstrator is used to decode and play back an audio stream received from a fast Ethernet. The audio stream is sent from a desktop computer. The first thing that happens is that the desktop computer sends an ARP request to find out the Ethernet address of the demonstrator. When the ARP packet arrives to the demonstrator, the protocol processor decodes it and saves the payload in the data memory. It also sets the corresponding control signal high that tells the microcontroller that an ARP packet has arrived. The microcontroller then creates an ARP packet with its Ethernet address and sends it back to the desktop computer through the special purpose register that is the ARP interface.

When the Ethernet address is known, the desktop computer starts to send an audio stream. When a packet arrives, it is decoded and stored in a circular buffer in the data memory by the protocol processor. The protocol processor also tells the microcontroller that a new audio packet has arrived by setting the corresponding control signal high. The microcontroller then starts sorting the buffer. When it is sorted, it waits for another packet to arrive. The entire packet flow between the desktop computer and the demonstrator is described in figure 2.3.

To be able to handle late packets etc, the play back of the audio stream does not start until a certain number of packets have arrived. When a sufficient number of packets have arrived, it starts to write audio samples to the special purpose register that is the interface to the stereo codec. It writes to this register every time the control signal is set high.

---

Figure 2.3, Packet flow from the desktop computer to the demonstrator
Chapter 3, Design of Instruction Set

This chapter describes the design process of the instruction set. It also mentions some general aspects of instruction set design. The instruction set defines the processor architecture and it is the programmer’s view of the system. You could say that the instruction set is the link between the hardware and software [2].

3.1 General Aspects

The main task, during the design of the instruction set, is to decide the instructions that should be available in the processor. Addressing modes and the format of the instruction word are also important tasks. The designer must also make some decisions about the register file. For example, how many registers there should be in the processor. To make these decisions, some aspects need to be considered.

3.1.1 Functional Coverage

The most important aspect when designing an instruction set is to achieve functional coverage, which means that the processor must be able to run all necessary applications. When designing an application specific processor this is rather obvious, but it is also important to consider this when designing a general purpose processor. Of course, it is much more difficult for general purpose because the amount of application is infinite.

3.1.2 Performance Demands

Another important aspect when designing an instruction set is demands on performance. It might be necessary to implement complex instructions to fulfill demands on performance, even though it will not affect the functional coverage. There could be other reasons to implement complex instructions. For example, it could reduce the amount of program memory needed for an application if one instruction could replace many simpler instructions. One drawback with complex instructions is that it requires extra hardware, which increase chip area and probably power consumption. Benchmarking and iterative methods is often used during the instruction set design, to improve the performance.
3.1.3 Orthogonal Instruction Set

An orthogonal instruction set, which means that all instructions have the same format and could use all registers [1], is beneficial to use in some situations. One of the advantages is that it makes it easier for the compiler. However, it is not always beneficial to have an orthogonal instruction set.

3.1.4 General Instructions

When designing an application specific instruction set, it is customized to be able to run some specific applications. Sometimes it could be a good idea to make it a little bit more general by implementing some common instructions. This could be an advantage when the applications are not that specific or if the probability that the applications will change after a period of time is large. It also increases the possibility to use the processor for other applications.

3.2 This microcontroller

The first step in the instruction set design was to determine what kind of demands there were on the microcontroller. One demand was that the program memory bus should be 16 bits wide and therefore it is natural to use an instruction word that is 16 bits wide. Another demand was the applications that the microcontroller must be able to run. These applications are described in the following section.

3.2.1 Applications

To be able to design the applications it is necessary to decide the structure of the buffer for the audio packets. This buffer is divided in two. The first is the circular data buffer where the protocol processor stores the audio packets. The first word in each packet contains the sequence number of the packet and then the audio samples follows. The second one is the sorting buffer used by the sorting algorithm. Each packet in the data buffer has a corresponding packet in the sorting buffer. Each packet in the sorting buffer contains three pointers and the sequence number of the packet. There is one pointer to the previous packet and one to the next packet. There is also a pointer to corresponding packet in the data buffer. The reason why the sequence number is stored in the sorting buffer is that it will decrease the number of memory accesses in the sorting algorithm. For a more detailed description of the pointers, see figure 3.1.

The first application that the microcontroller should be able to handle is to send audio samples to the stereo codec. The audio samples are stored in the data memory and the microcontroller should move them to a 16 bits register, which is the interface to the stereo codec. Figure 3.2 describes this application. There are some demands on performance for this application. After the register is empty, the microcontroller must write the next sample during the next 260 clock cycles.
Figure 3.1, Description of the buffer, the arrows shows an example of how the pointers could be directed

Output <= (smallest + offset)
Offset <= offset + 1

Offset = blocksize

Output <= 1

Smallest <= smallest.next

Figure 3.2, Behavioral model of output empty subroutine

Design Of Instruction Set
The microcontroller must also be able to handle ARP (Address Resolution Protocol) requests. When a request arrives, the microcontroller should create an ARP packet and send it to the Ethernet interface. This interface consists of a 16 bits register. This application is described in figure 3.3. This application has some real time demands. When transmission of the ARP packet starts, the microcontroller must write to the register every fourth clock cycle. To achieve this, the memory arbiter must be configured so that the microcontroller has exclusive rights to the memory during this period.

The last application that the microcontroller should be able to handle is the routine that sorts the audio packets. When the packets arrive out of order, this application should make sure that they are sorted before they are sent to the stereo codec. Other things that this application should handle are to remove duplicate packets and packets that arrive late. The time for this sorting algorithm to finish could be rather long, therefore the application must check if the output to the stereo codec is empty during the main loop. For a more detailed description of this application, see figure 3.4. The complete application written in assembly code could be found in appendix B.
Design Of Instruction Set
3.2.2 Registers

The decision about how many registers that should be implemented in the microcontroller were based on the applications described in the previous section. It turned out that 16 registers were sufficient. As mentioned in chapter 2 two special purpose registers is necessary as ARP and stereo codec interface. Since the address bus to the data memory is 20 bits wide general register 0 to 13 is 20 bits wide. Register 14 and 15 are the special purpose register mentioned above and should be 16 bits wide. It would be difficult to use more than 16 registers, because of the limitations in the instruction word length. To many bits in the instruction word would be used to point out the operand registers and it would be too few bits left for the operation code.

The status register contains three flags, N, O and Z.

N – Negative value.
O – Overflow has occurred.
Z – Value equals zero.

The only flags that are necessary for the applications in the demonstrator are N and Z, but O was implemented to make the microcontroller a little bit more general.

3.2.3 Choosing the Instructions

The next step in the design process was to decide the instructions that should be included in the microcontroller. To achieve functional coverage these decisions were based on the behavioral models of the applications described in the previous section.

The behavioral models show that branch instructions are very common in these applications, especially conditional branch. Normal conditional branch instructions based on the status register needs to be implemented. Instructions for three different conditions are implemented.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Status flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than</td>
<td>N=0, Z=0</td>
</tr>
<tr>
<td>Not equal</td>
<td>Z=0</td>
</tr>
<tr>
<td>Overflow</td>
<td>O=1</td>
</tr>
</tbody>
</table>

In addition to these some special conditional branch instructions are necessary, for example to be able to branch when the output register to the stereo codec is empty. To solve this problem, instructions that branch conditionally on an external signal were implemented. Five external signals could be used for this purpose. Three of them are used in the applications for the demonstrator. It could seem unnecessary to implement five when only three are used. This is because when the design of the instruction set started, the number of external control signal needed was not certain. For example, one signal was in the beginning used by the ARP interface, but later in the design process, this signal became redundant.

Subroutine branch instructions were also implemented, both unconditional and conditional. These instructions store the program counter and the status flags so that they could be restored when returning from the subroutine. Only one version could be saved, therefore nested subroutine calls are not allowed. A stack could have been used to allow nested subroutine calls, but to make the microcontroller simpler and to minimize the number of memory accesses this was not implemented. Another reason not to implement a stack was that it was not necessary for the application used in the demonstrator.
There are two versions of all branch and subroutine branch instructions. One is a relative jump where the length of the jump is given in the instruction word. The other is an absolute jump where jump address is given in a register. Since the instruction word is only 16 bits wide, the absolute jump with the address in a register is necessary to be able to reach all positions in the program memory. The absolute branch instructions are sufficient for all applications, but the relative versions were implemented to make it simpler to write programs to the microcontroller. The relative branches also reduce the amount of code. An example how the code could be reduced by using relative branch instructions is shown below. Both alternatives results in a jump to address 209. Three clock cycles and three words in memory are saved with relative branch. Load constant, LDLI, occupies two words in the memory.

<table>
<thead>
<tr>
<th>Code with absolute branch</th>
<th>Code with relative branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>197            LDHI GR0 #0</td>
<td>197           JMPR #12</td>
</tr>
<tr>
<td>198            LDLI GR0 #209</td>
<td>200            JMP GR0</td>
</tr>
</tbody>
</table>

Other instructions, which are necessary for functional coverage, are load to register and store in memory. General register 0 to 13 are 20 bits and the data memory is 16 bits, therefore there are two types of all load and store instructions. One operate on the 16 least significant and another operate on the four most significant bits.

The only arithmetic instructions needed are addition, subtraction and compare. There are two types for each of these instructions. One operates on the 16 least significant bits and the other on all 20 bits. The 20 bit version could only operate on register 0 to 13 since register 14 and 15 are only 16 bits.

Some logical instructions and shift instructions were also implemented to achieve a more general microcontroller. These are not necessary for the applications of the demonstrator. These instructions operate on 20 bits and therefore only on general register 0 to 13.

Normally benchmarking is used during the instruction set design. It is used to reduce the number of clock cycles that is needed to execute important applications. Since this microcontroller is designed for a special purpose, where performance is not that important, benchmarking was not used during the design process.

All instructions implemented in the microcontroller are listed in table 3.1. A more detailed description off all instruction could be found in appendix A.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD16</td>
<td>1</td>
<td>Addition of two 16 bits number</td>
</tr>
<tr>
<td>ADD20</td>
<td>1</td>
<td>Addition of two 20 bits number</td>
</tr>
<tr>
<td>CMP16</td>
<td>1</td>
<td>Compare two 16 bits number</td>
</tr>
<tr>
<td>CMP20</td>
<td>1</td>
<td>Compare two 20 bits number</td>
</tr>
<tr>
<td>SUB16</td>
<td>1</td>
<td>Subtraction of two 16 bits number</td>
</tr>
<tr>
<td>SUB20</td>
<td>1</td>
<td>Subtraction of two 20 bits number</td>
</tr>
<tr>
<td>BF1 - BF5</td>
<td>2</td>
<td>Conditional jump, external signal, absolute address</td>
</tr>
<tr>
<td>BO</td>
<td>2</td>
<td>Conditional jump, overflow, absolute address</td>
</tr>
<tr>
<td>BGT</td>
<td>2</td>
<td>Conditional jump, greater than, absolute address</td>
</tr>
<tr>
<td>BNE</td>
<td>2</td>
<td>Conditional jump, not equal, absolute address</td>
</tr>
<tr>
<td>JMP</td>
<td>2</td>
<td>Unconditional jump, absolute address</td>
</tr>
<tr>
<td>BF1R – BF5R</td>
<td>3</td>
<td>Conditional jump, external signal, relative address</td>
</tr>
<tr>
<td>BOR</td>
<td>3</td>
<td>Conditional jump, overflow, relative address</td>
</tr>
<tr>
<td>BGT R</td>
<td>3</td>
<td>Conditional jump, greater than, relative address</td>
</tr>
<tr>
<td>BNER</td>
<td>3</td>
<td>Conditional jump, not equal, relative address</td>
</tr>
<tr>
<td>JMP R</td>
<td>3</td>
<td>Unconditional jump, relative address</td>
</tr>
<tr>
<td>BSF1 – BSF5</td>
<td>2</td>
<td>Conditional subroutine call, external signal, absolute address</td>
</tr>
<tr>
<td>BSO</td>
<td>2</td>
<td>Conditional subroutine call, overflow, absolute address</td>
</tr>
<tr>
<td>BSGT</td>
<td>2</td>
<td>Conditional subroutine call, greater than, absolute address</td>
</tr>
<tr>
<td>BSNE</td>
<td>2</td>
<td>Conditional subroutine call, not equal, absolute address</td>
</tr>
<tr>
<td>BSR</td>
<td>2</td>
<td>Unconditional subroutine call, absolute address</td>
</tr>
<tr>
<td>BSF1R – BSF5R</td>
<td>3</td>
<td>Conditional subroutine call, external signal, relative address</td>
</tr>
<tr>
<td>BSOR</td>
<td>3</td>
<td>Conditional subroutine call, overflow, relative address</td>
</tr>
<tr>
<td>BSCTR</td>
<td>3</td>
<td>Conditional subroutine call, greater than, relative address</td>
</tr>
<tr>
<td>BSNER</td>
<td>3</td>
<td>Conditional subroutine call, not equal, relative address</td>
</tr>
<tr>
<td>BSRR</td>
<td>3</td>
<td>Unconditional subroutine call, relative address</td>
</tr>
<tr>
<td>RTS</td>
<td>7</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>LDL</td>
<td>1</td>
<td>Load data to the 16 lsb from memory</td>
</tr>
<tr>
<td>LDLO</td>
<td>1</td>
<td>Load data to the 16 lsb from memory, offset in register</td>
</tr>
<tr>
<td>LDLOD</td>
<td>4</td>
<td>Load data to the 16 lsb from memory, offset in instruction</td>
</tr>
<tr>
<td>LDLI</td>
<td>5</td>
<td>Load data to the 16 lsb with constant</td>
</tr>
<tr>
<td>LDH</td>
<td>1</td>
<td>Load data to the 4 msb from memory</td>
</tr>
<tr>
<td>LDHO</td>
<td>1</td>
<td>Load data to the 4 msb from memory, offset in register</td>
</tr>
<tr>
<td>LDHOD</td>
<td>4</td>
<td>Load data to the 4 msb from memory, offset in instruction</td>
</tr>
<tr>
<td>LDHI</td>
<td>6</td>
<td>Load data to the 4 msb with constant</td>
</tr>
<tr>
<td>MOVE</td>
<td>1</td>
<td>Move data between two registers</td>
</tr>
<tr>
<td>STL</td>
<td>1</td>
<td>Store data in memory from the 16 lsb</td>
</tr>
<tr>
<td>STLO</td>
<td>1</td>
<td>Store data in memory from the 16 lsb, offset in register</td>
</tr>
<tr>
<td>STLOD</td>
<td>4</td>
<td>Store data in memory from the 16 lsb, offset in instruction</td>
</tr>
<tr>
<td>STH</td>
<td>1</td>
<td>Store data in memory from the 4 msb</td>
</tr>
<tr>
<td>STHO</td>
<td>1</td>
<td>Store data in memory from the 4 msb, offset in register</td>
</tr>
<tr>
<td>STHOD</td>
<td>4</td>
<td>Store data in memory from the 4 msb, offset in instruction</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>Bitwise and between two registers</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>Bitwise or between two registers</td>
</tr>
<tr>
<td>XOR</td>
<td>1</td>
<td>Bitwise xor between two registers</td>
</tr>
<tr>
<td>INV</td>
<td>2</td>
<td>Bitwise inversion of register</td>
</tr>
<tr>
<td>LSL</td>
<td>1</td>
<td>Logical left shift</td>
</tr>
<tr>
<td>LSR</td>
<td>1</td>
<td>Logical right shift</td>
</tr>
<tr>
<td>ASR</td>
<td>1</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td>NOP</td>
<td>7</td>
<td>No instruction</td>
</tr>
</tbody>
</table>

Table 3.1. All instructions in the microcontroller
3.2.4 Instruction Word Format

After the decision about which instructions that should be included in the processor, the next step is to decide the instruction word format. As mentioned earlier the program memory bus is 16 bits wide. Therefore, it is natural to use an instruction word that is 16 bits wide. The width of 16 bits is enough for all instructions but one. To be able to load a register with data from program memory a 32 bit instruction is necessary. This is also the only instruction that needs more than one clock cycle to be decoded and executed. The seven different instruction word formats are described in table 3.2.

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>11</th>
<th>7</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format 1</td>
<td>Op-code</td>
<td>Reg1</td>
<td>Reg2</td>
<td></td>
</tr>
<tr>
<td>Format 2</td>
<td>Op-code</td>
<td>Reg1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format 3</td>
<td>Op-code</td>
<td>Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format 4</td>
<td>Op-code</td>
<td>Offset</td>
<td>Reg1</td>
<td>Reg2</td>
</tr>
<tr>
<td>Format 5</td>
<td>Op-code</td>
<td>Reg1</td>
<td>xxxx</td>
<td>Data</td>
</tr>
<tr>
<td>Format 6</td>
<td>Op-code</td>
<td>Reg1</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Format 7</td>
<td>Op-code</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2, Instruction formats in this instruction set

As shown in table 3.2 most instruction formats use operands in registers. This is because the instruction word is only 16 bits wide. It is not possible to include operands in the instruction word. There are some exceptions to this, relative branch instructions, load with offset and store with offset. In the case of relative branch, 8 bits are used to decide the length of the jump, this gives a range for the jump of -128 to 127. In the case of load and store with offset, four bits are used for the offset. This gives a range for the offset of 0 to 15. All instructions that generate a result use GR[Reg1] as destination register. Table 3.1 contains a list of all instruction and which format it uses.

3.2.5 Addressing modes

Store and load are the only two types of instructions that could access the memory. There are three different types of addressing modes for these instructions. These are register direct, register direct with offset in register and register direct with offset in instruction. There is also a fourth type of load instruction with immediate data but this does not operate on the data memory.

Instructions with register direct addressing use the instruction word format 1, see table 3.2. Reg2 in the table tells which general register that contain the memory address.

Instructions word format 1 is also used for instructions with register direct with offset in register addressing. The general register that contains the base address is pointed out by Reg2 in table 3.2. General register 12 is implicitly used as offset register. The absolute address is GR[Reg2] + GR12.
Instructions with register direct with offset in instruction uses instruction word format 4. GR[Reg2] is a pointer to the general register that contains the base address and the offset are four bits included in the instructions word. The offset is interpreted as positive numbers between 0 and 15. The absolute address is GR[Reg2] + offset. The different addressing modes are described in figure 3.5.

![Figure 3.5, Available addressing modes in the microcontroller](image_url)
Chapter 4, Instruction Set Simulator

This chapter describes the instruction set simulator (ISS). The instruction set simulator is a program that simulates the functionality of the instruction set for the microcontroller. It should be able to run programs as if it was the microcontroller itself and it should generate the same result. To be able to see the program flow the ISS could create a log of instructions and the internal state. The ISS could also contain functionality for debugging. The last part of this chapter describes how the ISS is used to verify that functional coverage has been achieved with the selected instruction set.

4.1 Benefits of using an ISS

The ISS is important in the design of a processor and it is useful in many steps of the design process.

The first thing the ISS does is to load a binary file created by the assembler and transform it back to assembly instructions. This makes it easy to verify the functionality of the assembler, by just compare the program from the ISS and original assembly program. If they are equal, it is a great possibility that the assembler works as intended.

The ISS is also used to test the behavior of the processor. It is a complete behavior model of the processor at instruction level. With the ISS, it is possible to verify functional coverage with the intended instruction set. For this, the ISS must be bit-true, which means that it must produce exactly the correct result. The ISS could also be cycle-true and then it is possible to test performance demands. Since it is very hard to make the ISS cycle-true this is normally not true.

Another step in the design process when the ISS is useful is during verification of the hardware implementation. The ISS could be used to generate reference data, which could be used to compare with data generated by the hardware.

Another advantage with ISS is that it could be used by software engineers, so that they could start develop the applications before the complete hardware exists. This means that software and hardware engineers could work concurrent. This is very important because of the demands on short development time.
4.2 How it works

The ISS for the microcontroller in this project works the following way. First of all the program and other inputs are loaded from files. Examples of other inputs are the content of the data memory and stimuli to the external flags. Then the instruction counter is reset to zero and then the loop with the execution of instructions starts. The first thing that happens in the loop is that stimuli are set. If no stimuli for the present value of the program counter exist, then the flags are not updated. Then the instruction is executed, if the log option is set then the program also writes to the log. The log consists of the assembly code and the content of all registers and memory locations that the instruction uses. The status register is also written to the log. The loop is controlled by the number of instruction that should be executed. Figure 4.1 describes program flow in the ISS.

Each line in the stimuli file used in the ISS consists of the following things, a time stamp, the external flags and the content of the memory mapped register with the address to last incoming packet. The time stamp is an ordinary integer that tells the simulator during which clock cycle the stimuli should be applied. The external flags are two digits coded in hexadecimal and tell the simulator the state of the five external flags. The memory mapped register is coded with five hexadecimal digits.

In addition to the log the ISS also creates four output files. The first two output files are the stereo codec interface and the ARP interface. These interfaces are the registers GR14 and GR15. Every time the simulator loads data from memory to these registers, the data is also written to the corresponding output file. There are also two output files that consist of the clock cycle when the data was written to the stereo codec and the ARP interface. These are used to verify that data has been written to the corresponding registers fast enough.

To be able to generate reference data for the hardware implementation another log file is created. Every clock cycle the simulator writes the contents of all registers and status register to this file.

4.3 Implementation

The ISS was implemented in C++. The structure of the program is object oriented. It consists of two mayor classes, one describing the hardware and one base class for the instructions. There is also one class for each instruction. The base class for all instructions and class definition for one instruction could be seen below.

Base class for all instructions:

```cpp
class Line
{
  public:
    char bin[17];
    virtual void Execute(bool log, bool log2) = 0;
    virtual void PrintText() = 0;
    int Bin2Dec(const char *str, int pos, int nob, bool sign);
    Line() {};
    virtual ~Line() {};
};
```

Class for the instruction ADD16:

```cpp
class ADD16 : public Line
{
  public:
    ADD16(char code[17]);
    void Execute(bool log, bool log2);
    void PrintText();
  private:
    int GRx;
    int GRy;
};
```

Instruction Set Simulator
When the program is loaded into the program memory, an object is created for each instruction in the program. The program memory in the simulator is modeled as an array of pointers to objects of the class `Line`, the base class for all instructions. The instruction `LDLI` should occupy two lines in the memory but in the simulator, this is solved by inserting a NOP instruction after the `LDLI`.

![Flowchart](image-url)

**Figure 4.1, Program flow of the instruction set simulator**
There are two functions in every instruction class, Execute and PrintText. The PrintText function is used for logging purposes. The Execute function contains the information about what should happen when the instruction is executed. This function calls different functions in the hardware class so that the content in all registers and memory is updated correctly. The classes for each instruction also contain different variables for the operands of the instruction. These variables are private and could only be used by the object itself.

The code example below shows the execute function for the ADD16 instruction. It first reads the operands from the registers and stores them in the variables x and y. Then it writes the sum of these variables to the correct register. A mask is used to get the correct number of bits since this instruction only operates on 16 bits. After the register has been updated, it calls the function that updates the status register. Then the program counter is updated and the counter for the number of instruction executed is increased by one. At last, a log is created if this option is true.

Example of the execute function:

```cpp
void ADD16::Execute(bool log, bool log2)
{
    const int mask=0xFFFF;
    int x, y;
    x = (processor.ReadGR(GRx, 'l') & mask);
    y = (processor.ReadGR(GRy, 'l') & mask);
    processor.SetGR((x+y) & mask, GRx, 'l');
    processor.UpdateFlags((x+y), 16, x, y, '+');
    processor.UpdatePC();
    processor.Tick();
    if (log)
    {
        PrintText();
        cout << "PC: " << hex << setfill('0') << setw(5) << processor.ReadPC() << dec << endl;
        cout << "Clk: " << processor.clk << endl;
        cout << "GR" << GRx <<": " << hex << setfill('0') << setw(5) << processor.ReadGR(GRx,'a') << dec << endl;
        cout << "GR" << GRy <<": " << hex << setfill('0') << setw(5) << processor.ReadGR(GRy,'a') << dec << endl;
    }
}
```

The main loop in the simulator consists mainly of two function calls. First it calls the SetStimuli function that sets the flags and memory mapped registers. Then it calls the execute function for the current instruction. It also creates the log that contains the content in all registers.

Main loop of the simulator:

```cpp
for(int i=0; i < no_clk; i++)
{
    SetStimuli();
    processor.ProgMem[processor.ReadPC()]->Execute(log, log2);
    if (log2)
    {
        Here is the content in all registers written to a log file.
    }
}
```
The code for the program is divided into different files. The class that describes the hardware of the microcontroller is implemented in two files called “hardware.c” and “hardware.h”. The classes that describe all instructions are implemented in “instruction.c” and “instruction.h”. The base class for all instructions could be found in “line.h”. The main program of the simulator is located in the file “simulator.c”.

4.4 Verification of functional coverage

To verify that the instruction set was sufficient to implement the applications in section 3.2.1 and that all demands where fulfilled five different test cases where performed. Because of difficulties in simulating the behavior of the protocol processor, no new packets arrived during runtime. All packets were already in the memory but the addresses to the packets did arrive during runtime. The audio packets consists of 244 audio samples, which correspond to 5 ms. In all test cases, 12 500 000 clock cycles were simulated, which corresponds to 500 ms of audio. The output files, which represent the stereo codec and ARP interface, were used to check the correctness of the application.

When all test cases produced the correct result the instruction set were considered sufficient for the desired application.

4.4.1 Test Case 1

The first test was performed just to see that main functionality in the application worked as it should. It was also used to verify that the ISS worked as expected. All the audio packets in the memory were placed in order, which mean that the sorting algorithm was not used that much. The ARP packet was placed first in the memory and it arrived after 8 clock cycles. Then after another 250 clock cycles the audio packets started to arrive. The first audio packet had sequence number zero. The application for the microcontroller was programmed to buffer 10 audio packets before it starts to send samples to the stereo codec. Therefore, the first 10 packets arrived with much smaller interval than in reality, to reduce the simulation time. They arrived with 1 000 clock cycles interval compared to 125 000 clock cycles in reality. During this time, the playback controller signals that it wants a new sample but the application should not send any until it has received 10 packets. After the arrival of the first 10 packets, the microcontroller starts to send samples to the stereo codec and during this time, the packets arrived with the correct interval. The clock cycle when the packets arrive is described in table 4.1.

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>ARP</td>
</tr>
<tr>
<td>258</td>
<td>Audio 0</td>
</tr>
<tr>
<td>1258</td>
<td>Audio 1</td>
</tr>
<tr>
<td>9258</td>
<td>Audio 9</td>
</tr>
<tr>
<td>135186</td>
<td>Audio 10</td>
</tr>
<tr>
<td>260114</td>
<td>Audio 11</td>
</tr>
<tr>
<td>385042</td>
<td>Audio 12</td>
</tr>
</tbody>
</table>

Table 4.1, Clock cycle when the packets arrive in test case 1
4.4.2 Test Case 2

The goal with the second test was to test the sorting algorithm. The packets arrived out of order or more precisely, they were placed out of order in the memory. During simulation, the buffer size was ten packets and this gives the limit of how late a packet could arrive without being lost. All packets arrived in time so that no packets should be lost. As in test case 1, the first 10 packets arrived with much smaller interval than in reality.

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>ARP</td>
</tr>
<tr>
<td>258</td>
<td>Audio 3</td>
</tr>
<tr>
<td>1258</td>
<td>Audio 0</td>
</tr>
<tr>
<td>2258</td>
<td>Audio 2</td>
</tr>
<tr>
<td>3258</td>
<td>Audio 1</td>
</tr>
<tr>
<td>4258</td>
<td>Audio 4</td>
</tr>
<tr>
<td>5258</td>
<td>Audio 5</td>
</tr>
<tr>
<td>6258</td>
<td>Audio 6</td>
</tr>
<tr>
<td>7258</td>
<td>Audio 7</td>
</tr>
<tr>
<td>8258</td>
<td>Audio 8</td>
</tr>
<tr>
<td>9258</td>
<td>Audio 32</td>
</tr>
<tr>
<td>135186</td>
<td>Audio 9</td>
</tr>
<tr>
<td>260114</td>
<td>Audio 10</td>
</tr>
<tr>
<td>385042</td>
<td>Audio 11</td>
</tr>
<tr>
<td>509970</td>
<td>Audio 12</td>
</tr>
<tr>
<td>634898</td>
<td>Audio 13</td>
</tr>
<tr>
<td>759826</td>
<td>Audio 15</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Table 4.2, Clock cycle when the packets arrive in test case 2

4.4.3 Test Case 3

In the third test, the audio packets arrived in pairs with minimum spacing between them. The time between the packets was 2 048 clock cycles or 82 µs, this limit is because fast Ethernet cannot transmit faster. The packets did also arrive out of order. The purpose with this test is to check if the application could handle packets that arrives before the sorting of the previous packet has finished. The ARP packet and the 10 first audio packets arrive as in the previous test cases. Then they arrive in pairs with minimum spacing.

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>ARP</td>
</tr>
<tr>
<td>258</td>
<td>Audio 0</td>
</tr>
<tr>
<td>1258</td>
<td>Audio 2</td>
</tr>
<tr>
<td>2258</td>
<td>Audio 3</td>
</tr>
<tr>
<td>3258</td>
<td>Audio 4</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>9258</td>
<td>Audio 11</td>
</tr>
<tr>
<td>12306</td>
<td>Audio 5</td>
</tr>
<tr>
<td>260114</td>
<td>Audio 23</td>
</tr>
<tr>
<td>262162</td>
<td>Audio 6</td>
</tr>
<tr>
<td>509970</td>
<td>Audio 12</td>
</tr>
<tr>
<td>312018</td>
<td>Audio 13</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Table 4.3, Clock cycle when the packets arrive in test case 3
**4.4.4 Test Case 4**

The goal with the fourth test was to check if the application managed to handle missing packets, duplicate packets and packets that arrive too late. When a packet is missing, the preceding packets should be sent to the stereo codec once more. When a duplicate packet arrives, it should be discarded and this is also the case when a packet arrives too late.

In this test, packet with sequence number four is missing. This means that packet with sequence number three should be retransmitted to the playback controller.

A duplicate of packet with sequence number five arrives in this test. This packet should be discarded of the application.

The packet with sequence number nine arrives too late in this test. This means that when it arrives it should be discarded. It also means that packet with sequence number eight should be retransmitted to the playback controller.

**4.4.5 Test Case 5**

The audio packets that are stored in the data memory consist of a sequence number followed by audio samples. The sequence number is used by the sorting algorithm. The sequence number is represented by a 16 bit number this gives a limitation in its size. To be able to handle this, the packet with sequence number 0 must be sent to the stereo codec after the packet with sequence number 65 535. This test was performed to verify this.

The packet with sequence number 65 530 should be sent first to the stereo codec. After packet 65 535, the packet with sequence number 0 should be transmitted. The packets did arrive out of order.

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>ARP</td>
</tr>
<tr>
<td>258</td>
<td>Audio 65530</td>
</tr>
<tr>
<td>1258</td>
<td>Audio 65532</td>
</tr>
<tr>
<td>2258</td>
<td>Audio 65533</td>
</tr>
<tr>
<td>3258</td>
<td>Audio 65531</td>
</tr>
<tr>
<td>4258</td>
<td>Audio 5</td>
</tr>
<tr>
<td>5258</td>
<td>Audio 65534</td>
</tr>
<tr>
<td>6258</td>
<td>Audio 0</td>
</tr>
<tr>
<td>7258</td>
<td>Audio 1</td>
</tr>
<tr>
<td>8258</td>
<td>Audio 65535</td>
</tr>
<tr>
<td>9258</td>
<td>Audio 2</td>
</tr>
<tr>
<td>135186</td>
<td>Audio 3</td>
</tr>
<tr>
<td>260114</td>
<td>Audio 6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.4, Clock cycle when the packets arrive in test case 5
Chapter 5, Design of Micro Architecture

This chapter describes the design of the architecture for the microcontroller. Some design choices are motivated and the RTL implementation is described.

5.1 Top Level Design

The top-level design describes the microcontroller at a high level. The microcontroller is described as large blocks and buses. The main decision that must be made is how the blocks are interconnected. The method for making the top-level design should be mapping one instruction at the time into hardware. Much of the top-level design is done during the design of the instruction set, so called concurrent design.

5.1.1 Method

One example of mapping of an instruction to hardware could be the ADD20 instruction. This instruction adds two operands and therefore an adder is necessary. This instruction uses two operands from the register file. Therefore we need two buses from the register file to the arithmetic unit were the operation should be performed. The result from the operation should be stored in the register file and for this, another bus from the arithmetic unit to the register file is necessary. Another example of this mapping could be for the instruction LDL. This instruction load data from the memory to a register in the register file. The address is also stored in the register file and therefore we need one address bus from the register file to the memory. A data bus from the memory to the register file is also necessary.

The method used above continues until all instructions could be executed. To reduce the size of the design it is important to multiplex the buses. This means that one bus could be used for many things. For example, there are only one bus between the arithmetic and logic unit and the register file. The result from the adder, shifter and the logic block use the same bus.


**5.1.2 Architecture Overview**

Figure 5.1 describes the block and the buses of the microcontroller. Some parts are not included, such as the instruction decoder and the program flow controller. These are not included since they only produce control signals and no internal buses are connected to them.

The ALU, arithmetic and logic unit, is only connected to the RF, register file, since all arithmetic, logical and shift instructions use operands from the register file and always store the result in the register file. The adder in the ALU is also used for the address calculation when using offset addressing. To do this, the result bus from the ALU to RF is connected to the address bus in the register file. Since the ALU should perform 20-bit operations, all buses between RF and ALU are 20 bits wide.

For all absolute branch instructions, the jump address is stored in the RF. Therefore, it must be connected to the PC, program counter. One of the operand buses to the ALU is used for this purpose and is connected to the PC. The relative branch instruction uses data from the instructions word to calculate the jump address. Therefore, the PC must also be connected to the data bus from the program memory.

Load and store are the only instructions that could access the data memory. This means that the interface to the data memory only needs to be connected to the RF. The ARP and stereo codec interfaces are also connected to the RF.

![Figure 5.1, Overview of the microcontroller with all buses](image_url)
5.2 Data Path

The data path in the microcontroller consists of the RF and the ALU. The ALU could be divided in three parts, the arithmetic unit, the logic unit and the shift unit. The design methodology of the data path is similar to the one used for the top-level design. Instructions should be mapped into hardware one at the time.

5.2.1 Arithmetic Unit

The arithmetic unit should be able to perform addition and subtraction. It should be able to perform these operations on both 16 bits and 20 bits operands. It should also generate three status flags, negative, overflow and zero. The structure of the arithmetic unit is described in figure 5.2.

In figure 5.2 there are two blocks called “create operand”, these creates the correct operands and are controlled by whether it is a 16 bit or 20 bit operation. The operands to the adder are 21 bits, 20 bits and one guard bit. If it is a 20-bit operation, the sign bit is copied to the guard bit. If it is a 16-bit operation, the sign bit is copied to bit 16 to 20.

The arithmetic unit should be able to perform subtraction. Therefore, the second operand could be inverted and the carry in signal to the adder could be controlled.

In figure 5.2 there is a block called “generate flags”, it should update all status flags. This block is controlled by whether it is a 16-bit or a 20-bit operation. One example is when the negative flag should be updated, it should look at the sign bit and this bit is of course in different positions for 16-bit and 20-bit operations.

![Figure 5.2, The arithmetic unit](image-url)
5.2.2 Logic Unit

Another part of the ALU is the logic unit, which should perform all logic operations. The operations that it should be able to perform are and, or, xor and it should also be able to invert all bits in an operand. These operations only operate on the full data width, 20 bits. The INV operation only needs one operand and it is taken from rf_alu_opa_bus. The structure of the logic unit is described in figure 5.3.

The logic unit should also generate flags for the operations. The flags that should be updated are negative and zero. The overflow flag are not updated since it is not possible to get overflow in these operations. All different operations use the same hardware for updating the flags.

5.2.3 Shift Unit

The shift unit should be able to perform three different shift operations, arithmetic right shift, logical right shift and logical left shift. The shift unit contains three barrel shifters, one for each type of shift operation. The structure of the shift unit is described in figure 5.4.

The shift unit needs two operands. The operand that should be shifted comes from rf_alu_opa_bus and the operand that determines the number of steps comes from rf_alu_opb_bus. The five least significant bits of this bus are used to control the multiplexers in the shift unit.

The shift unit must also generate flags for the operations that it performs. All operations should generate a negative and a zero flag. The logical left shift should also generate an overflow flag, but for the other shift operations, it is not relevant. To be able to generate the overflow flag for the logical left shift operation all out shifted bits must be saved. In all levels of the shifter, the out shifted bits must be saved. If a level does not perform a shift the out shifted bits instead contains multiple copies of the sign bit. To be able to determine if an overflow has occurred, all these out shifted bits are compared with the shifted numbers sign bit. If they are not all equal overflow has occurred.

![Figure 5.3, The logic unit](image-url)
Figure 5.4, The shift unit
5.2.4 Register File

The register file contains 14 20-bit general purpose registers and 2 16-bit special purpose registers. It also contains the status register and a saved version of the status register, which is used by subroutine branch instructions.

Figure 5.5 describes the possibilities to load data into the registers. All load instructions use either data from the data memory or from the instruction word. This means that all registers must be connected to dm_data_bus and pm_data_bus. All computational instructions save the data in the register file. This means that all registers also must be connected to alu_rf_bus.

The registers that contain 20 bits are divided into one 4-bit part and one 16-bit part. This means that it is possible to write data into the different parts independently of the other. The memory data bus is only 16 bits wide and the division of the register into two parts makes it possible to store and load data from and to the four most significant bits.

The register file contains the status flags. Figure 5.6 shows the hardware for updating a flag. A new state for a flag could come from either a saved flag or the ALU. The status flag are also connected to the program counter. The saved flag are updated when subroutine branch instructions are executed. It should only be updated when the conditional branch is taken and therefore all different branch conditions must be used as inputs to the leftmost multiplexer. Figure 5.6 only shows zero flag but for the other flags, it is the same.

There are four output buses from the register file. Figure 5.7 shows how these are connected to the different registers. There are two buses to the ALU and both are 20 bits wide. All registers are connected to these buses. Since G14 and GR15 are 16-bit registers the four most significant bits are zero in these cases. Load and store instructions with offset in the instruction word uses the ALU to calculate the address and therefore one rf_alu_opb_bus must also be connected to pm_data_bus. Since GR14 and GR15 are only 16-bit registers, these are not connected to dm_addr_bus. As mentioned above the ALU is used to calculate the address for offset addressing. This means that the alu_rf_bus must be connected to dm_addr_bus. It should be possible load and store data to and from all bits in the register. This means that the dm_data_bus is connected to both the 16-bit part and the 4-bit part of all registers.
Figure 5.5, The registers in the register file and its inputs

Figure 5.6, The hardware for a status flag
Figure 5.7, The register file and its outputs
5.3 Control Path

In this microcontroller, the control path consists of the instruction decoder, ID, the program counter, PC, and the program flow controller, PFC. The performance on the microcontroller does not need to be very high. Therefore, to make the microcontroller simpler, no pipeline was implemented.

5.3.1 Instruction Decoder

The instruction decoder reads the instruction word and creates control signals. Most of the control signals go to the data path and the control path, but some external control signals are also created.

The instruction LDLI uses two instruction words. The first contain the op-code and destination register and the second contain the data. This means that the instruction decoder must remember the destination register one clock cycle. At first, this was solved by saving all control signals one clock cycle. The problem with this is that it requires some extra hardware, one D-flop for each control signal. To reduce the hardware cost only four bits for the destination register was saved.

5.3.2 Program Counter

The program counter is responsible to calculate the address to the following instruction. Normally, the PC is just increased by one each clock cycle, but for program flow instructions the new address could be different. The program counter is 19 bits wide, which means that it could address 512 kWords of program memory. The program counter is described in figure 5.8.

For all conditional branch instructions, the new address depends on an internal or external condition. The jump address could come from a register in the register file. Therefore, the program counter is connected to the rf_alu_opa_bus. To be able to execute a relative branch instruction the program counter must be connected to the pm_data_bus.

The program counter also contains a PC_saved register. This register is used to save the return address when using subroutine branch instructions. When a program returns from a subroutine, the value in PC_saved is added with one and stored in PC.

In figure 5.8, there are three large multiplexers used to control other multiplexers. These are used to determine if a conditional branch should occur or not. The upper left multiplexer is used for relative branch instructions. The lower left one is used for absolute branch instructions and the right one is used for subroutine branch instructions.
5.3.3 Program Flow Controller

Since there is no pipeline in this microcontroller the only thing that it must take care of is the LDLI instruction. Since it consists of two instruction words, the second one must not be interpreted as an instruction. When the first word of the instruction arrives at the instruction decoder, it sets the signal id_pfc_ldli high. This will trigger the state machine in the program flow controller to change state. In this state, the disable_id signal is high and this will make sure that the instruction decoder does not interpret the following instruction word as an instruction. The next clock cycle the state machine returns to the ground state. Figure 5.9 describe the state machine in the program flow controller.
This chapter describes the verification process of the microcontroller. It also discusses some general aspects of verification.

6.1 General Aspects

During the verification process, the designer must make sure that the design behaves as intended. The implementation should be consistent with the specification.

Verification of a system is a large part of the design and it could be the most difficult part. The problem with verification is that the designs are often very large, which means that the verification time could be very long. There are two different methods for performing the verification, formal verification and verification by simulation.

6.1.1 Formal Verification

Formal verification is a method of mathematically checking that the behaviour of a system satisfies a given property. This property must hold for all possible inputs or at least a big class of inputs. To be able to use formal verification the system must be described in a restricted way, for example a finite state machine. Of course it is not always possible to describe an entire system in this restricted way, but some parts could often be described this way. One part of a processor that could be described like this is the program flow controller, which normally is a finite state machine. This means that it is very hard to use formal methods to verify an entire system but it could be used to verify some parts of the system. There are many different methods for formal verification and one example is model checking. More information about formal verification could be found in [4].

One area where formal methods are a good choice, is safety critical systems. A dead look of a finite state machine in this area could cause a catastrophe. Since verification of finite state machines is possible with formal methods the dead look would probably be found during verification.
6.1.2 Verification by Simulation

Verification by simulation is the dominating method for verification of digital systems. The main problem with simulation is that it is very time consuming. It is not possible to test all combinations of inputs since it takes such a long time. Therefore, it is necessary to use some kind of metric to measure how much of the system that was tested. There are some different metrics for this purpose.

Line coverage is a metric that measures how many percent of the code that has been executed during the simulation. Branch coverage measures how many branches that have been taken and path coverage measures the percent of all possible paths that have been executed. All these metrics come from software testing [5].

There are also other coverage metrics that could be used for some special hardware. For example, state coverage and transition coverage. These could be used to measure how much of a finite state machine that has been verified.

Another important aspect that needs to be considered is how the test vectors should be generated. One common method is random testing, which means that the test vectors are generated at random. This could easily be done with a small software program. Another method is corner testing. The generation of test vectors for this method is harder to do with software. It is probably easier to do it by hand. There are other algorithms that could be used to automatically generate test vectors. One example of these could be found in [6].

6.2 Verification of Microcontroller

The microcontroller in this master thesis was verified by simulation. The tool used to simulate the microcontroller was Model Sim, from Mentor Graphics. This simulator only supported line coverage metric. This is not a very good metric since it is very easy to achieve 100 percent line coverage. Therefore, the microcontroller was considered verified after 3 weeks of verification work.

The test vectors for the simulation of the microcontroller were mainly the program loaded into the program memory. They were generated randomly, but some corner tests were also performed.

6.2.1 Random Testing

During the random testing a C++ program was written, that generated random program code for the microcontroller. This program works the following way. Since the microcontroller has 67 instructions, a number between 0 and 66 were randomly generated. Each number represented a specific instruction. This number was translated to the operation code for the specific instruction. If the instruction needs an operand, these were generated randomly as well and added to the operation code.

Since all registers in the microcontroller are zero when the simulation starts, it is very likely that a branch instruction with a register as destination address, will produce an infinite loop already in the beginning of the program. A couple of tricks were used to minimize the risk for this problem.
First of all, the program always starts with 50 LDLI instructions. The operands to these instructions were random. This result in that most of the registers were assigned a value differing from zero. Secondly, RTS instructions were not allowed in the beginning of the program. At last, the probability for branch instructions were made less compared to other instructions. In the code example below eight successive branch instructions must be generated to produce one branch instruction.

```c
for(long int i=0;(i < (prog_size - 1)) && (line_counter < (524288 - 1)); i++)
{
    if (i < 100)
        op_code_rand = 46; //LDLI
    else
        while((op_code_rand == 42) && (line_counter < 25000)) //Don't allow RTS in the beginning of the program
            op_code_rand = rand() % 66;
    for (int j=0; j < 8; j++)
    {
        if (((op_code_rand > 5) && (op_code_rand < 43))) //Less probability for branch instructions and RTS
            op_code_rand = rand() % 66;
    }
    strcpy(line, gen_op_code(op_code_rand)); //Generate operands for the specific instruction
    strncpy(line, "1100100110000000", 17); //Always end with JMPR # -128
    out_file << line << endl;
    line_counter++;
}
```

Nine test programs were generated and simulated on the microcontroller. The probability for branch instructions varied between the programs. One of the test programs did not contain any branch instruction at all. All programs were simulated more than 10 000 clock cycles.

The test bench for the microcontroller was constructed so that every clock cycle the content of all register and status register were written to an output file. This file was then compared with the output file produced by the instruction set simulator.

### 6.2.2 Corner Testing

For a complete verification of the microcontroller, it is necessary to generate test vectors for the external control signals. Since the time for verification was limited, it was not tested during the random testing. Therefore, a test program and a test vector for the external control signals were created by hand. The test program verified all branch instructions that use F1 to F5 as conditions. In addition to all branch instructions it also contained load, arithmetic and shift instructions so that the content of the registers would differ if the branch was taken or not. The test vector for the external control signals was constructed so that the conditions were both true and not true. The instruction set simulator was used to create the reference files for the simulation.

The application that should run on the microcontroller was not used to verify the microcontroller itself. Since the application was tested on the instruction set simulator and the implementation behaved as the instruction set simulator this did not seem necessary.
The final verification of the microcontroller was conducted together with the entire demonstrator. For this simulation, the correct application was used. The microcontroller did work as intended during this simulation. This part of the verification was performed by Tomas Henriksson since he designed the protocol processor and other parts of the demonstrator.
Chapter 7, Results and Conclusions

This chapter summarizes results and conclusions from the master thesis project. It also mentions some things that could be improved in future versions of the microcontroller.

The goal with this master thesis project was to design a microcontroller and implement it in VHDL. The microcontroller should be able to co-operate with a protocol processor and some other hardware and carry out certain tasks in the demonstrator. The simulation of the demonstrator shows that the microcontroller fulfills these requirements.

The following things were produced during the project:

1. Instruction set architecture
2. Assembler implemented in C++
3. Instruction set simulator implemented in C++
4. Assembly program for the microcontroller
5. Micro architecture for the microcontroller implemented in VHDL

During startup of the project, the requirements were listed. All primary requirements were fulfilled during the project. The assembly program for the microcontroller was constructed, which means that one of the secondary requirements was fulfilled. The secondary requirements also state that the microcontroller should be integrated with the demonstrator and that the demonstrator should be synthesized to the FPGA. These requirements were not fulfilled. The work with integration and synthesis was done by Tomas Henriksson since he designed the protocol processor and other parts of the demonstrator.

Since this microcontroller is designed for a special purpose, where the performance is not that important, optimization of the instruction set has not been considered. This means that the required number of clock cycles to run certain applications could probably be reduced by using benchmarking and iterative methods during the design of the instruction set.

The instruction set is not full, there is space for additional instructions. This means that new instructions could be added to increase the functionality of future versions of the microcontroller.

A problem occurred during synthesis to the FPGA. It showed that the microcontroller was not fast enough. The microcontroller should run in 25 MHz. To achieve this it must be pipelined. A pipeline could rather easily be introduced in the microcontroller. When this master thesis project started a pipeline did not seem necessary and to keep the microcontroller as simple as possible it was not implemented.
Another improvement that possibly will increase the performance in the FPGA is to use multiplexers with not more than four inputs. In the design, there are multiplexers with up to 32 inputs and describing these as combinations of multiplexers with four inputs could possibly increase the performance. This improvement is related to the limitations in the FPGA and the synthesis tool.

Results and Conclusions


Appendix A, Instruction Set

This appendix contains a description of the instruction set of the microcontroller. It describes the syntax, possible operands, op code and a description of the execution. All instructions are 16 bits wide except for LDLI, which is 32 bits wide.

All instructions that use offset in register implicitly use GR12 as offset register. The registers are divided in two parts, therefore the instruction must state which part it should operate on. This is implicitly decided by the instruction. For example, ADD16 always operate on the 16 least significant bits and ADD20 use all 20 bits in the register.

When using subroutine instructions the program counter and the status flags, N, O and Z, is saved and could be restored when returning from the subroutine. Only one version of the program counter and the status flags could be saved, therefore no nested subroutines calls could be used.

All values in an instruction such as offset, relative jump and data could be written in several ways. All numbers should start with “#”. The number could be expressed as hexadecimal, binary or decimal. When using hexadecimal the “#” should be followed by an “h”. When using binary numbers the “#” should be followed by a “b”.

Appendix A
**ADD16**

**Type of instruction**
Arithmetic instruction, signed addition of two 16 bits number.

**Syntax**
ADD16 GRx GRy

**Operands**
GRx: GR0 – GR15
GRy: GR0 – GR15

**Execution**
GRx + GRy → GRx
if (GRx+GRy) < 0
  N=1, Z=0
else if (GRx+GRy) = 0
  N=0, Z=1
else
  N=0, Z=0
if (GRx+GRy) > h7FFF or (GRx+GRy) < h8000
  O=1
else
  O=0

**Op code**
```
1000 0000 | rrrr | RRRR
```
“RRRR” is GRy and “rrrr” is GRx.

**Description**
The 16 least significant bits in register GRx and GRy are added and the result is stored in GRx. The status flags N, O and Z are updated.

**Example**
ADD16 GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>h700a3</td>
<td>h701a4</td>
</tr>
<tr>
<td>GR2</td>
<td>h10101</td>
<td>h10101</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b10000100</td>
<td>b10000000</td>
</tr>
</tbody>
</table>

Appendix A
**ADD20**

**Type of instruction**
Arithmetic instruction, signed addition of two 20 bits number.

**Syntax**
ADD20 GRx GRy

**Operands**
GRx: GR0 – GR13  
GRy: GR0 – GR13

**Execution**

\[ GRx + GRy \rightarrow GRx \]

- if \((GRx+GRy) < 0\)
  - \(N=1, Z=0\)
- else if \((GRx+GRy) = 0\)
  - \(N=0, Z=1\)
- else
  - \(N=0, Z=0\)
  - if \((GRx+GRy) > h7FFFF\) or \((GRx+GRy) < h80000\)
    - \(O=1\)
  - else
    - \(O=0\)

**Op code**

\[
\begin{array}{ccc}
1000 & 0001 & \text{rrrr} \\
\end{array}
\]

“RRRR” is GRy and “rrrr” is GRx.

**Description**
The values in register GRx and GRy are added and the result is stored in GRx. The status flags N, O and Z are updated.

**Example**
ADD20 GR3 GR4

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR3</td>
<td>h600a3</td>
<td>h701a4</td>
</tr>
<tr>
<td>GR4</td>
<td>h10101</td>
<td>h10101</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b01000000</td>
<td>b01000000</td>
</tr>
</tbody>
</table>

Appendix A
**AND**

**Type of instruction**
Logic instruction, bitwise and between two registers.

**Syntax**

\[ \text{AND GRx GRy} \]

**Operands**

GRx: GR0 – GR13  
GRy: GR0 – GR13

**Execution**

\[ \text{GRx} \& \text{GRy} \rightarrow \text{GRx} \]

\[
\begin{align*}
\text{if } (\text{GRx} \& \text{GRy}) < 0 & \quad N=1, Z=0 \\
\text{else if } (\text{GRx} \& \text{GRy}) = 0 & \quad N=0, Z=1 \\
\text{else} & \quad N=0, Z=0
\end{align*}
\]

**Op code**

\[
\begin{array}{c|cc}
1111 & 0000 & \text{rrrr} \\
& \text{RRRR}
\end{array}
\]

“RRRR” is GRy and “rrrr” is GRx.

**Description**

Bitwise and between the values in GRx and GRy. Operates on all 20 bit, therefore it is not possible to operate on GR14 or GR15. The result is stored in GRx. The flags N and Z are updated.

**Example**

\[ \text{AND GR0 GR1} \]

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR0</td>
<td>h00fff</td>
<td>h00320</td>
</tr>
<tr>
<td>GR1</td>
<td>h1a320</td>
<td>h1a320</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b01000000</td>
<td>b01000000</td>
</tr>
</tbody>
</table>

Appendix A
**ASR**

**Type of instruction**
Shift instruction, arithmethic right shift.

**Syntax**
ASR GRx GRy

**Operands**
GRx: GR0 – GR13  
GRy: GR0 – GR13

**Execution**
\[
GRx >> (GRy) \rightarrow GRx
\]

**Op code**

<table>
<thead>
<tr>
<th></th>
<th>1111 0110</th>
<th>rrrr</th>
<th>RRRR</th>
</tr>
</thead>
</table>

“RRRR” is GRy and “rrrr” is GRx.

**Description**
The value in GRx is shifted to the right. The value in GRy decides the number of steps that GRx should be shifted. The signbit is shifted in to the most significant bit. The flags N,O and Z are updated. The instruction operates on 20 bits, therefore GR14 and GR15 can’t be used.

**Example**
ASR GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>h80ff0</td>
<td>hf80ff</td>
</tr>
<tr>
<td>GR2</td>
<td>h00004</td>
<td>h00004</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00000000</td>
<td>b00000000</td>
</tr>
</tbody>
</table>

Appendix A
**BFx, BFxR**

**Type of instruction**
Program flow instruction, conditional jump.

**Syntax**
1) BF1 GRx
   Register direct addressing
2) BF1R #data
   Relative addressing

**Operands**
GRx: GR0 – GR13
h0 ≤ data ≤ hFF

**Execution**
if Fx=1 then
1) GRx \( \rightarrow \) PC
2) PC + data \( \rightarrow \) PC
else
PC + 1 \( \rightarrow \) PC

**Op code**

<table>
<thead>
<tr>
<th></th>
<th>1100 0000 0001</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0000 0010</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0000 0011</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0000 0100</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0000 0101</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0001</th>
<th>dddd dddd</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF1R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0010</th>
<th>dddd dddd</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF2R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0011</th>
<th>dddd dddd</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF3R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0100</th>
<th>dddd dddd</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF4R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100 0101</th>
<th>dddd dddd</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF5R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description
The program counter is set to the value in GRx or increased by data if the flag Fx is one, if not PC is increased by one. F1 to F5 are the status flags that can be used. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.

Example
BF5R #-28

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h00ff3</td>
<td>h00fd7</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00101110</td>
<td>b00101110</td>
</tr>
</tbody>
</table>
**BGT, BGTR**

**Type of instruction**
Program flow instruction, conditional jump.

**Syntax**
1) BGT GRx  
   Register addressing
2) BGTR #data  
   Relative addressing

**Operands**
GRx: GR0 – GR13  
h0 ≤ data ≤ hFF

**Execution**
*If N=0 and Z=0 then*
1) GRx → PC
2) PC + data → PC
*else*
PC + 1 → PC

**Op code**

<table>
<thead>
<tr>
<th></th>
<th>BGT</th>
<th>BGTR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1100 0000 0111</td>
<td>rrrr</td>
</tr>
<tr>
<td>BGTR</td>
<td>1100 0111</td>
<td>ddd dddd</td>
</tr>
</tbody>
</table>

**Description**
The program counter is set to the value in GRx or increased by data if the flags N and Z are zero, if not PC is increased by one. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.

**Example**
BGT GR13

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h000a3</td>
<td>h00f01</td>
</tr>
<tr>
<td>GR13</td>
<td>h80f01</td>
<td>h80f01</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b10000000</td>
<td>b10000000</td>
</tr>
</tbody>
</table>

Appendix A
**BNE, BNER**

**Type of instruction**
Program flow instruction, conditional jump.

**Syntax**
1) BNE GRx  
   Register direct addressing
2) BNER #data  
   Relative addressing

**Operands**
GRx: GR0 – GR13
h0 ≤ data ≤ hFF

**Execution**

\[
\begin{align*}
&\text{if } Z=0 \text{ then} \\
&\text{1) } GRx \rightarrow PC \\
&\text{2) } PC + data \rightarrow PC \\
&\text{else} \\
&\text{PC + 1} \rightarrow PC
\end{align*}
\]

**Op code**

<table>
<thead>
<tr>
<th>BNE</th>
<th>1100 0000 1000</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNER</td>
<td>1100 1000</td>
<td>dddd dddd</td>
</tr>
</tbody>
</table>

**Description**
The program counter is set to the value in GRx or increased by data if the flag Z are zero, if not PC is increased by one. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.

**Example**

BNER #hA

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h00ff3</td>
<td>h00ff4</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00100001</td>
<td>b00100001</td>
</tr>
</tbody>
</table>

Appendix A
**BO, BOR**

**Type of instruction**
Program flow instruction, conditional jump.

**Syntax**
1) BO GRx  
   Register direct addressing
2) BOR #data  
   Relative addressing

**Operands**
GRx: GR0 – GR13  
h0 ≤ data ≤ hFF

**Execution**
if O=1 then
1) GRx → PC  
2) PC + data → PC
else
PC + 1 → PC

**Op code**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BO</td>
<td>1100 0000 0110</td>
<td>rrrr</td>
</tr>
<tr>
<td>BOR</td>
<td>1100 0110</td>
<td>dddd dddd</td>
</tr>
</tbody>
</table>

**Description**
The program counter is set to the value in GRx or increased by data if the flag O are one, if not PC is increased by one. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.

**Example**
BOR #hA

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h00ff3</td>
<td>h00ffd</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00100110</td>
<td>b00100110</td>
</tr>
</tbody>
</table>

Appendix A
**BSFx, BSFxR**

**Type of instruction**
Program flow instruction, conditional subroutine jump.

**Syntax**
1) BF5 GRx  Register direct addressing
2) BF5R #data Relative addressing

**Operands**
GRx: GR0 – GR13
h0 ≤ data ≤ hFF

**Execution**
if Fx=1 then
PC → PC saved
Flags → Flags saved
1) GRx → PC
2) PC + data → PC
else
PC + 1 → PC

**Op code**

<table>
<thead>
<tr>
<th>BSF1</th>
<th>1110 0000 0001</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSF2</td>
<td>1110 0000 0010</td>
<td>rrrr</td>
</tr>
<tr>
<td>BSF3</td>
<td>1110 0000 0011</td>
<td>rrrr</td>
</tr>
<tr>
<td>BSF4</td>
<td>1110 0000 0100</td>
<td>rrrr</td>
</tr>
<tr>
<td>BSF5</td>
<td>1110 0000 0101</td>
<td>rrrr</td>
</tr>
<tr>
<td>BSF1R</td>
<td>1110 0001</td>
<td>dddd dddd</td>
</tr>
<tr>
<td>BSF2R</td>
<td>1110 0010</td>
<td>dddd dddd</td>
</tr>
<tr>
<td>BSF3R</td>
<td>1110 0011</td>
<td>dddd dddd</td>
</tr>
<tr>
<td>BSF4R</td>
<td>1110 0100</td>
<td>dddd dddd</td>
</tr>
<tr>
<td>BSF5R</td>
<td>1110 0101</td>
<td>dddd dddd</td>
</tr>
</tbody>
</table>

Appendix A
Description
If the flag Fx is one then PC is saved and PC is set to GRx or PC+data, depending on the instruction. The status flags, N, O and Z, are also saved. If Fx is zero PC is increased by one. Only one return address could be saved, which means that no nested subroutine calls could be used. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.

Example
BSF1R #5

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h00ff3</td>
<td>h00ff8</td>
</tr>
<tr>
<td>PCsaved</td>
<td>h00012</td>
<td>h00ff3</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b10000110</td>
<td>b10000110</td>
</tr>
<tr>
<td>Flagssaved(N,O,Z)</td>
<td>b000</td>
<td>b110</td>
</tr>
</tbody>
</table>
**BSGT, BSGTR**

**Type of instruction**
Program flow instruction, conditional subroutine jump.

**Syntax**
1) BSGT Grx
2) BSGTR #data

**Operands**
GRx: GR0 – GR13
h0 ≤ data ≤ hFF

**Execution**

\[
\text{if } N=0 \text{ and } Z=0 \text{ then}
\]

\[
\begin{align*}
\text{PC} &\rightarrow \text{PCsaved} \\
\text{Flags} &\rightarrow \text{Flagssaved} \\
1) \text{GR}x &\rightarrow \text{PC} \\
2) \text{PC} + \text{data} &\rightarrow \text{PC}
\end{align*}
\]

**else**

\[
\begin{align*}
\text{PC} + 1 &\rightarrow \text{PC}
\end{align*}
\]

**OP Code**

<table>
<thead>
<tr>
<th>BSGT</th>
<th>1110 0000 0111 rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSGTR</td>
<td>1110 0111 dddd dddd</td>
</tr>
</tbody>
</table>

**Description**
If the result of the last operation is larger than zero, then the value in PC is saved and set to GRx or PC+data. The status flags, N and Z, are also saved. If the result is smaller or equal to zero, then PC is increased by one. Only one return address could be saved, which means that no nested subroutine calls could be used. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.
**Example**

BSGTR #hF

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h00ff3</td>
<td>h00ff4</td>
</tr>
<tr>
<td>PCsaved</td>
<td>h00012</td>
<td>h00012</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b10000100</td>
<td>b10000100</td>
</tr>
<tr>
<td>Flagssaved(N,O,Z)</td>
<td>b000</td>
<td>b000</td>
</tr>
</tbody>
</table>
**BSNE, BSNER**

**Type of instruction**
Program flow instruction, conditional subroutine jump.

**Syntax**
1) BSNE Grx
2) BSNER #data

**Operands**
Grx: GR0 – GR13
h0 ≤ data ≤ hFF

**Execution**

\[ \text{if } Z = 0 \text{ then} \]
\[
\begin{align*}
\text{PC} &\rightarrow \text{PC}\text{saved} \\
\text{Flags} &\rightarrow \text{Flagssaved} \\
1) \ GRx &\rightarrow PC \\
2) \ PC + \text{data} &\rightarrow PC \\
\text{else} \\
PC + 1 &\rightarrow PC
\end{align*}
\]

**OP Code**

\[
\begin{array}{c|c}
\text{BSNE} & 1110 0000 1000 \text{ rrrr} \\
\hline
\text{BSNER} & 1110 1000 \text{ dddd dddd}
\end{array}
\]

**Description**
If the result of the last operation is not equal to zero, then the value in PC is saved and set to Grx or PC+data. The status flags, N and Z, are also saved. If the result is smaller or equal to zero, then PC is increased by one. Only one return address could be saved, which means that no nested subroutine calls could be used. The program counter is only 19 bits wide, which means that the most significant bit in Grx is discarded when using register direct addressing.
### Example

BSNE GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR2</td>
<td>h01f00</td>
<td>h01f00</td>
</tr>
<tr>
<td>PC</td>
<td>h00ff3</td>
<td>h01f00</td>
</tr>
<tr>
<td>PCsaved</td>
<td>h00012</td>
<td>h00ff3</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b10000110</td>
<td>b10000110</td>
</tr>
<tr>
<td>Flagssaved(N,O,Z)</td>
<td>b000</td>
<td>b110</td>
</tr>
</tbody>
</table>
**BSO, BSOR**

**Type of instruction**
Program flow instruction, conditional subroutine jump.

**Syntax**
1) BSO Grx
2) BSOR #data

**Operands**
GRx: GR0 – GR13
h0 ≤ data ≤ hFF

**Execution**
if O=1 then
PC → PCsaved
Flags → Flagssaved
1) GRx → PC
2) PC + data → PC
else
PC + 1 → PC

**OP Code**

<table>
<thead>
<tr>
<th>BSO</th>
<th>1110 0000 0110</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSOR</td>
<td>1110 0110</td>
<td>dddd dddd</td>
</tr>
</tbody>
</table>

**Description**
If the result of the last operation generates an overflow, then the value in PC is saved and set to GRx or PC+data. The status flags, N, O and Z, are also saved. If the result didn’t generate an overflow, then PC is increased by one. Only one return address could be saved, which means that no nested subroutine calls could be used. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.
### Example

**BSO GR2**

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR2</td>
<td>h01f00</td>
<td>h01f00</td>
</tr>
<tr>
<td>PC</td>
<td>h00ff3</td>
<td>h01f00</td>
</tr>
<tr>
<td>PCsaved</td>
<td>h00012</td>
<td>h00ff3</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b10000110</td>
<td>b10000110</td>
</tr>
<tr>
<td>Flagssaved(N,O,Z)</td>
<td>b000</td>
<td>b110</td>
</tr>
</tbody>
</table>
**BSR, BSRR**

**Type of instruction**
Program flow instruction, subroutine jump.

**Syntax**
1) BSR GRx Register direct addressing
2) BSRR #data Relative addressing

**Operands**
GRx: GR0 – GR13
h0 ≤ data ≤ hFF

**Execution**

\[ PC \rightarrow PC_{\text{saved}} \]

Flags → Flagssaved
1) \( GRx \rightarrow PC \)
2) \( PC + \text{data} \rightarrow PC \)

**Op code**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR</td>
<td>1110 0000 1001</td>
<td>rrrr</td>
</tr>
<tr>
<td>BSRR</td>
<td>1110 1001</td>
<td>dddd dddd</td>
</tr>
</tbody>
</table>

**Description**
This is a subroutine jump. The old value in PC is saved in \( PC_{\text{saved}} \) and then PC is updated with a new value. The status flags, N and Z, are also saved. Only one return address could be saved, which means that no nested subroutine calls could be used. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.

**Example**

BSRR #–64

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h00ff0</td>
<td>h00fb0</td>
</tr>
<tr>
<td>PC_{\text{saved}}</td>
<td>h00012</td>
<td>h00ff0</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b10001001</td>
<td>b10001001</td>
</tr>
<tr>
<td>Flagssaved(N,O,Z)</td>
<td>b000</td>
<td>b001</td>
</tr>
</tbody>
</table>

Appendix A
**CMP16**

**Type of instruction**
Arithmetic instruction, compare two 16 bits numbers.

**Syntax**
CMP16 GRx GRy

**Operands**
GRx: GR0 – GR15
GRy: GR0 – GR15

**Execution**

\[ GRx - Gry \rightarrow \text{none} \]

if \((GRx - GRy) < 0\)
\[ N=1, Z=0 \]
else if \((GRx - GRy) = 0\)
\[ N=0, Z=1 \]
else
\[ N=0, Z=0 \]

if \((GRx - GRy) > h7FFF \text{ or } (GRx - GRy) < h8000\)
\[ O=1 \]
else
\[ O=0 \]

**Op code**

| 1000 0010 | rrrr | RRRR |

“RRRR” is GRy and “rrrr” is GRx.

**Description**
GRy is substracted from GRx but the result is not saved. The flag Z is set to one if GRx and GRy are equal. The flag N is set to one if GRy > GRx. If overflow occurs O is set to one.

**Example**

CMP16 GR5 GR6

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR5</td>
<td>h10ff3</td>
<td>h10ff3</td>
</tr>
<tr>
<td>GR6</td>
<td>h020fa</td>
<td>h020fa</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b000000101</td>
<td>b00000100</td>
</tr>
</tbody>
</table>

Appendix A
**CMP20**

**Type of instruction**
Arithmetic instruction, compare two 20 bits numbers.

**Syntax**
CMP20 GRx GRy

**Operands**
GRx: GR0 – GR13
GRy: GR0 – GR13

**Execution**
GRx – Gry → none
if (GRx - GRy) < 0
N=1, Z=0
else if (GRx - GRy) = 0
N=0, Z=1
else
N=0, Z=0
if (GRx - GRy) > h7FFFF or (GRx - GRy) < h80000
O=1
else
O=0

**Op code**
```
1000 0011  rrrr  RRRR
```
“RRRR” is GRy and “rrrr” is GRx.

**Description**
GRy is substracted from GRx but the result is not saved. The flag Z is set to one if GRx and GRy are equal. The flag N is set to one if GRy > GRx. If overflow occurs O is set to one.

**Example**
CMP20 GR5 GR6

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR5</td>
<td>h10ff3</td>
<td>h10ff3</td>
</tr>
<tr>
<td>GR6</td>
<td>h020fa</td>
<td>h020fa</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00000001</td>
<td>b00000000</td>
</tr>
</tbody>
</table>

Appendix A
**INV**

**Type of instruction**
Logic instruction, invert register.

**Syntax**
INV GRx

**Operands**
GRx: GR0 – GR13

**Execution**

\[
\begin{align*}
\text{Inv(} \text{GRx} \text{)} & \rightarrow \text{GRx} \\
\text{if } (\text{inv(} \text{GRx} \text{)}) < 0 & \text{ } N=1, Z=0 \\
\text{else if } (\text{inv(} \text{GRx} \text{)}) = 0 & \text{ } N=0, Z=1 \\
\text{else } & \text{ } N=0, Z=0
\end{align*}
\]

**Op code**

1111 0011 0000  rrrr

**Description**
Invert GRx and the result is stored in GRx. The flags N and Z are updated.

**Example**
INV GR6

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR6</td>
<td>h00fff</td>
<td>hff00</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b01000000</td>
<td>b01000100</td>
</tr>
</tbody>
</table>

Appendix A
**JMP, JMPR**

Type of instruction
Program flow instruction, unconditional jump.

Syntax
1) JMP GRx Register direct addressing
2) JMPR #data Relative addressing

Operands
GRx: GR0 – GR13
h0 ≤ data ≤ hFF

Execution
1) GRx → PC
2) PC + data → PC

Op code

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>1100 0000 1001</td>
<td>rrrr</td>
</tr>
<tr>
<td>JMPR</td>
<td>1100 1001</td>
<td>dddd dddd</td>
</tr>
</tbody>
</table>

Description
This is an unconditional jump. PC is updated with a new value. The program counter is only 19 bits wide, which means that the most significant bit in GRx is discarded when using register direct addressing.

Example
JMPR #6

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h10ff3</td>
<td>h10ff9</td>
</tr>
</tbody>
</table>
**LDL, LDLO, LDLOD, LDLI**

**Type of instruction**
Data move instruction, load data to register with immediate data or from memory.

**Syntax**
1) LDL GRx GRy  
Register direct
2) LDLO GRx GRY  
Register direct with offset in register
3) LDLOD #offset GRx GRY  
Register direct with offset
4) LDLI GRx #data  
Immediate data

**Operands**
GRx: GR0 – GR15  
GRy: GR0 – GR13  
h0 ≤ offset ≤ hF  
h0000 ≤ data ≤ hFFFF

**Execution**
1) DM(GRy) → GRx  
2) DM(GRy + GR12) → GRx  
3) DM(GRy + offset) → GRx  
4) data → GRx

**Op code**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>RRRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDL</td>
<td>0000 0010</td>
<td>rr</td>
</tr>
<tr>
<td>LDLO</td>
<td>0000 0100</td>
<td>rr</td>
</tr>
<tr>
<td>LDLOD</td>
<td>0010 0000</td>
<td>rrrr</td>
</tr>
<tr>
<td>LDLI</td>
<td>0000 0110</td>
<td>rrr</td>
</tr>
</tbody>
</table>

“RRRR” is GRy and “rrrr” is GRx.

**Description**
Load data to the 16 least significant bits in register GRx. Register GR12 is implicitly used as offset register. In the first three addressing modes the data is taken from the memory and in the fourth it is taken from the instruction word.
Example

LDLI GR14 #hA001

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR14</td>
<td>h0ff3</td>
<td>ha001</td>
</tr>
</tbody>
</table>
**LDH, LDHO, LDHOD, LDHI**

**Type of instruction**
Data move instruction, load data to register with immediate data or from memory.

**Syntax**
1) LDH GRx GRy  
   Register direct
2) LDHO GRx GRY  
   Register direct with offset in register
3) LDHOD #offset GRx GRY  
   Register direct with offset
4) LDHI GRx #data  
   Immediate data

**Operands**
- GRx: GR0 – GR13
- GRy: GR0 – GR13
- h0 ≤ offset ≤ hF
- h0 ≤ data ≤ hF

**Execution**
1) DM(GRy) → GRx
2) DM(GRy + GR12) → GRx
3) DM(GRy + offset) → GRx
4) data → GRx

**Op code**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH</td>
<td>0000 1000 rrrr RRRR</td>
</tr>
<tr>
<td>LDHO</td>
<td>0000 1010 rrrr RRRR</td>
</tr>
<tr>
<td>LDHOD</td>
<td>0100 oooo rrrr RRRR</td>
</tr>
<tr>
<td>LDHI</td>
<td>0000 1100 rrrr dddd</td>
</tr>
</tbody>
</table>

“RRRR” is GRy and “rrrr” is GRx.

**Description**
Load data to the 4 most significant bits in register GRx. Register GR12 is implicitly used as offset register. In the first three addressing modes the data is taken from the memory and in the fourth it is taken from the instruction word. When data is taken from memory it is the 4 least significant bits in the memory word that is loaded into GRx.
**Example**

LDHO GR13 GR0

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR0</td>
<td>h000f0</td>
<td>h000f0</td>
</tr>
<tr>
<td>GR12</td>
<td>h00002</td>
<td>h00002</td>
</tr>
<tr>
<td>GR13</td>
<td>h01100</td>
<td>h21100</td>
</tr>
<tr>
<td>DM[h000f2]</td>
<td>h1112</td>
<td>h1112</td>
</tr>
</tbody>
</table>
**LSL**

**Type of instruction**
Shift instruction, logical left shift.

**Syntax**

LSL GRx GRy

**Operands**

GRx: GR0 – GR13  
GRy: GR0 – GR13

**Execution**

GRx << (GRy) → GRx

**Op code**

```
1111 0100  rrrr  RRRR
```

“RRRR” is GRy and “rrrr” is GRx.

**Description**

The value in GRx is shifted to the left. The value in GRy decides the number of steps that GRx should be shifted. Zeros are shifted in to the least significant bit. The flags N, O and Z are updated. The instruction operates on 20 bits, therefore GR14 and GR15 can’t be used.

**Example**

LSL GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>h00ff0</td>
<td>h0ff00</td>
</tr>
<tr>
<td>GR2</td>
<td>h00004</td>
<td>h00004</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00000000</td>
<td>b00000000</td>
</tr>
</tbody>
</table>
**LSR**

**Type of instruction**
Shift instruction, logical right shift.

**Syntax**
LSR GRx GRy

**Operands**
GRx: GR0 – GR13
GRy: GR0 – GR13

**Execution**
GRx >> (Gry) → GRx

**Op code**

<table>
<thead>
<tr>
<th>1111 0101</th>
<th>rrrr</th>
<th>RRRR</th>
</tr>
</thead>
</table>

“RRRR” is GRy and “rrrr” is GRx.

**Description**
The value in GRx is shifted to the right. The value in GRy decides the number of steps that GRx should be shifted. Zeros are shifted in to the most significant bit. The flags N,O and Z are updated. The instruction operates on 20 bits, therefore GR14 and GR15 can’t be used.

**Example**

LSR GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>h000ff</td>
<td>h0000f</td>
</tr>
<tr>
<td>GR2</td>
<td>h00004</td>
<td>h00004</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00000000</td>
<td>b00000000</td>
</tr>
</tbody>
</table>
**MOVE**

**Type of instruction**
Data move instruction, move between register.

**Syntax**
MOVE GRx GRy

**Operands**
GRx: GR0 – GR13
GRy: GR0 – GR13

**Execution**
GRy → GRx

**Op code**

```
0000 1110 rrrr RRRR
```

“RRRR” is GRy and “rrrr” is GRx.

**Description**
The value in GRy is copied to GRx.

**Example**

MOVE GR1 GR6

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>h10ff3</td>
<td>hfffff</td>
</tr>
<tr>
<td>GR6</td>
<td>hfffff</td>
<td>hfffff</td>
</tr>
</tbody>
</table>
**OR**

**Type of instruction**
Logic instruction, bitwise or between two registers.

**Syntax**
OR GRx GRy

**Operands**
GRx: GR0 – GR13
GRy: GR0 – GR13

**Execution**

\[
GRx \mid GRy \rightarrow GRx
\]

if \((GRx \mid GRy) < 0\)
\[N=1, Z=0\]
else if \((GRx \mid GRy) = 0\)
\[N=0, Z=1\]
else
\[N=0, Z=0\]

**Op code**

```
1111 0001  rrrr  RRRR
```

“RRRR” is GRy and “rrrr” is GRx.

**Description**
Bitwise or between the values in GRx and GRy. The result is stored in GRx. Operates on all 20 bits, therefore it is not possible to operate on GR14 or GR15. The flags N and Z are updated.

**Example**
OR GR2 GR3

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR2</td>
<td>h00fff</td>
<td>h1afff</td>
</tr>
<tr>
<td>GR3</td>
<td>h1a320</td>
<td>h1a320</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b01000001</td>
<td>b01000000</td>
</tr>
</tbody>
</table>

Appendix A
**RTS**

**Type of instruction**
Program flow instruction, return from subroutine.

**Syntax**
RTS

**Operands**
No operands.

**Execution**

\[ PC_{saved} + 1 \rightarrow PC \]

\[ Flagssaved \rightarrow Flags \]

**Op code**

\[
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

**Description**

Jumps back from the subroutine and restore the value in PC and the flags.

**Example**

RTS

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h0fa3</td>
<td>h000a2</td>
</tr>
<tr>
<td>PC_{saved}</td>
<td>h000a2</td>
<td>h000a2</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00000001</td>
<td>b00000010</td>
</tr>
<tr>
<td>Flagssaved(N,O,Z)</td>
<td>b010</td>
<td>b010</td>
</tr>
</tbody>
</table>
**STL, STLO, STLOD**

**Type of instruction**
Memory instruction, store data in memory.

**Syntax**
1) STL GRx GRy Register direct
2) STLO GRx GRy Register direct with offset in register
3) STLOD #offset GRx GRy Register direct with constant offset

**Operands**
GRx: GR0 – GR15
GRy: GR0 – GR13
h0 ≤ offset ≤ hF

**Execution**
1) GRx → DM(GRy)
2) GRx → DM(GRy + G12)
3) GRx → DM(GRy + offset)

**Op code**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STL</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td>STLO</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>STLOD</td>
<td>0110</td>
<td>0000</td>
</tr>
</tbody>
</table>

“RRRR” is GRy and “rrrr” is GRx.

**Description**
The 16 least significant bits in GRx are stored in memory.
### Example

STL GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>ha0ff3</td>
<td>ha0ff3</td>
</tr>
<tr>
<td>GR2</td>
<td>h00fff</td>
<td>h00fff</td>
</tr>
<tr>
<td>DM[h00fff]</td>
<td>h0111</td>
<td>h0ff3</td>
</tr>
</tbody>
</table>

Appendix A
**STH, STHO, STHOD**

**Type of instruction**
Memory instruction, store data in memory.

**Syntax**
1) STH GRx GRy  Register direct
2) STHO GRx GRy  Register direct with offset in register
3) STHOD #offset GRx GRy  Register direct with constant offset

**Operands**
GRx: GR0 – GR15
GRy: GR0 – GR13
h0 ≤ offset ≤ hF

**Execution**
1) GRx → DM(GRy)
2) GRx → DM(GRy + G12)
3) GRx → DM(GRy + offset)

**Op code**

<table>
<thead>
<tr>
<th></th>
<th>STH</th>
<th>STHO</th>
<th>STHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>0001 0001</td>
<td>0001 0011</td>
<td>0111 0000</td>
</tr>
<tr>
<td>rrrr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRRR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“RRRR” is GRy and “rrrr” is GRx.

**Description**
The 4 most significant bits in GRx are stored in memory.
Example

STH GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>ha0ff3</td>
<td>ha0ff3</td>
</tr>
<tr>
<td>GR2</td>
<td>h00ff</td>
<td>h00ff</td>
</tr>
<tr>
<td>DM[h00ff]</td>
<td>h0000</td>
<td>h000a</td>
</tr>
</tbody>
</table>
**SUB16**

**Type of instruction**
Arithmetic instruction, subtraction of two 16 bits number.

**Syntax**
SUB16 GRx GRy

**Operands**
GRx: GR0 – GR15
GRy: GR0 – GR15

**Execution**

\[ GRx - GRy \rightarrow GRx \]

- if \((GRx - GRy) < 0\)
  - \(N=1, Z=0\)
- else if \((GRx - GRy) = 0\)
  - \(N=0, Z=1\)
- else
  - \(N=0, Z=0\)
  - if \((GRx - GRy) > h7FFF or (GRx - GRy) < h8000\)
    - \(O=1\)
  - else
    - \(O=0\)

**Op code**

```
1000 0100 rrrr RRRR
```

“RRRR” is GRy and “rrrr” is GRx.

**Description**
GRy is subtracted from GRx and the result is stored in GRx. The flag Z is set to one if GRx and GRy are equal. The flag N is set to one if GRy > GRx. If overflow occurs O is set to one.

**Example**
SUB16 GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>h700a3</td>
<td>h7ff2a</td>
</tr>
<tr>
<td>GR2</td>
<td>h10101</td>
<td>h10101</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00000010</td>
<td>b00000100</td>
</tr>
</tbody>
</table>

Appendix A
**SUB20**

**Type of instruction**
Arithmetic instruction, subtraction of two 20 bits number.

**Syntax**
SUB20 GRx GRy

**Operands**
GRx: GR0 – GR13  
GRy: GR0 – GR13

**Execution**
\[\text{GRx} - \text{GRy} \rightarrow \text{GRx}\]
- \(\text{if } (\text{GRx} - \text{GRy}) < 0\)
  - \(N=1, Z=0\)
- \(\text{else if } (\text{GRx} - \text{GRy}) = 0\)
  - \(N=0, Z=1\)
- \(\text{else}\)
  - \(N=0, Z=0\)
  - \(\text{if } (\text{GRx} - \text{GRy}) > h7FFFF \text{ or } (\text{GRx} - \text{GRy}) < h80000\)
    - \(O=1\)
  - \(\text{else}\)
    - \(O=0\)

**Op code**

![binary code]

“RRRR” is GRy and “rrrr” is GRx.

**Description**
GRy is substracted from GRx and the result is stored in GRx. The flag Z is set to one if GRx and GRy are equal. The flag N is set to one if GRy > GRx. If overflow occurs O is set to one.

**Example**
SUB20 GR1 GR2

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR1</td>
<td>h700a3</td>
<td>h5fa2</td>
</tr>
<tr>
<td>GR2</td>
<td>h10101</td>
<td>h10101</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b00000110</td>
<td>b00000000</td>
</tr>
</tbody>
</table>

Appendix A
**XOR**

**Type of instruction**
Logic instruction, bitwise xor between two registers.

**Syntax**
XOR GRx GRy

**Operands**
GRx: GR0 – GR13
GRy: GR0 – GR13

**Execution**

\[
\text{GRx} \text{ xor } \text{GRy} \rightarrow \text{GRx}
\]

- if \((\text{GRx xor GRy}) < 0\)
  - \(N=1, Z=0\)
- else if \((\text{GRx xor GRy}) = 0\)
  - \(N=0, Z=1\)
- else
  - \(N=0, Z=0\)

**Op code**

\[
\begin{array}{c|c|c}
1111 & 0010 & \text{rrrr} \quad \text{RRRR} \\
\end{array}
\]

“RRRR” is GRy and “rrrr” is GRx.

**Description**

Bitwise xor between the values in GRx and GRy. The result is stored in GRx. Operates on all 20 bits, therefore it is not possible to operate on GR14 or GR15. The flags N and Z are updated.

**Example**

XOR GR4 GR5

<table>
<thead>
<tr>
<th>Register</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR4</td>
<td>h00fff</td>
<td>h00000</td>
</tr>
<tr>
<td>GR5</td>
<td>h00fff</td>
<td>h00fff</td>
</tr>
<tr>
<td>Flags(F1-F5,N,O,Z)</td>
<td>b01000000</td>
<td>b01000001</td>
</tr>
</tbody>
</table>

Appendix A
Appendix B, Application

*Flags:
* ARP request   F1
* New packet   F2
* Output register empty   F3

*Reserved Registers:
* number of packets to buffer   GR3
* number of received packets   GR4
* smallest   GR5
* largest   GR6
* last   GR7
* temp   GR8
* Offset   GR12
* ARP interface   GR14
* Stereo codec interface   GR15

*Jump to PP configuration
LH GR1 #h0
LDI GR1 #399  *Load address to PP configuration
BSR GR1

*Main loop
LH GR1 #0  *Load ARP routine address
LDI GR1 #138
BF1 GR1  *Jump to ARP routine
BF2R #38  *Jump to sort buffer routine
CMP16 GR4 GR3
BGTR #2
JMPR #-7
BSF3R #2  *Jump to output empty subroutine
JMPR #-9

*Subroutine output empty
LHOD #2 GR1 GR5
LDOD #3 GR1 GR5
LDO GR15 GR1  *Load output register(GR15) with data from memory
LDI GR11 #1
LH GR11 #0
ADD20 GR12 GR11  *offset = offset +1
LDI GR11 #489  *Load blocksize
LH GR11 #0
CMP20 GR12 GR11
NER #20  *offset=blocksize then goto (A)
LDI GR12 #1  *offset=1
LDH GR12 #0
LDLOD #4 GR0 GR5  *Load smallest sequence number to GR0
LDLOD #6 GR2 GR5
LDHOD #5 GR2 GR5
LDLOD #4 GR1 GR2  *Load next sequence number to GR1
LDLI GR2 #1
SUB16 GR1 GR0  *next.seq - smallest.seq
SUB16 GR1 GR2  *next.seq - smallest.seq - 1
BNER #2  *next.seq - smallest != 1 goto (L)
JMPR #4  *(M)
ADD16 GR0 GR2  *(L)
STLOD #4 GR0 GR5  *smallest.seq = smallest.seq + 1
RTS
MOVE GR11 GR5  *(M)
LDLOD #6 GR5 GR11  *smallest=smallest.next
LDHOD #5 GR5 GR11
RTS  *(A)
*End subroutine output empty

*Routine sort buffer(Can't use GR12)
LDLI GR13 #h0010  *Address to memory mapped register
LDHI GR13 #h8
LDH GR11 GR13  *Copy the address to the last block
LDLOD #1 GR11 GR13
CMP16 GR4 GR3
BGTR #4  *recived packets > packets to buffer then goto(K)
LDLI GR10 #1
ADD16 GR4 GR10
LDLI GR10 #0  *(K)
LDHI GR10 #0
CMP20 GR5 GR10
BNER #11  *smallest!=0 then goto (B)
LDLI GR5 #h0000  *smallest=start
LDHI GR5 #h7
MOVE GR6 GR5  *largest=start
MOVE GR7 GR5  *last= start
LDLOD #0 GR9 GR11  *Load last sequence number into GR9
STLOD #3 GR11 GR5  *Store address in memory
STHOD #2 GR11 GR5
STLOD #4 GR9 GR5  *Store sequence number in memory
JMPR #-63  *Jump to main loop
LDLI GR9 #7  *(B) Load constant
LDHI GR9 #0
ADD20 GR7 GR9  *Last=last+constant
LDLI GR9 #hFF00
LDHI GR9 #h7
CMP20 GR7 GR9
BGTR #2  *last > "last pos in buffer" then goto (C)
JMPR #4  *(D)
LDLI GR7 #h0000  *(C) Load start address
LDHI GR7 #h7
STLOD #3 GR11 GR7
STHOD #2 GR11 GR7
LDLOD #0 GR9 GR11
STLOD #4 GR9 GR7  *Store sequence number in memory
CMP16 GR4 GR3
BGTR #2  *This section check if the packet is to late

Appendix B
JMPR #5  *Buffering => always sort
LDLOD #4 GR10 GR5  *Load smallest.seq to GR10
CMP16 GR9 GR10
BGTR #2  *Packet is not to late, start sorting
JMPR #87  *Packet is to late,discard, jump to main loop

MOVE GR8 GR6  *Temp=largest
LDLOD #4 GR10 GR8  *(H) Load temp sequence number
CMP16 GR9 GR10
BGTR #20  *Last.seq > temp.seq then goto (I)
BNER #2
JMPR #37  *last.seq = temp.seq, discard packet goto (G)
CMP20 GR8 GR5
BNER #2  *temp != smallest goto (J)
JMPR #9  *(J)
MOVE GR11 GR8  *
LDLOD #1 GR8 GR11 *(H) Load temp sequence number
LDHOD #0 GR8 GR11
CMP16 GR4 GR3
BGTR #2
JMPR #13  *(J)
BSF3R #-96  *Output empty then goto subroutine
JMPR #-15  *Goto (H)

MOVE GR5 GR7  *(I)
STLOD #6 GR8 GR5  *(L) smallest=last
STHOD #5 GR8 GR5
STLOD #1 GR5 GR8  *temp.previous = smallest
STHOD #0 GR5 GR8
JMPR #20  *Goto (G)

CMP20 GR6 GR8  *(I)
BNER #2  *Largest != temp goto (F)
JMPR #12  *(F) last.previous=last
STLOD #1 GR8 GR7
STHOD #0 GR8 GR7
LDLOD #6 GR11 GR8  *Load temp.next
LDHOD #5 GR11 GR8
STLOD #1 GR7 GR11  *temp.next=last
STHOD #0 GR7 GR11
STLOD #6 GR11 GR7  *last.next=temp.next
STHOD #5 GR11 GR7
STLOD #6 GR7 GR8  *temp.next=last
STHOD #5 GR7 GR8
JMPR #6  *Goto (G)

STLOD #6 GR7 GR6  *(E) largest.next=last
STHOD #5 GR7 GR6
STLOD #1 GR6 GR7
STHOD #0 GR6 GR7
MOVE GR6 GR7  *largest=last
JMPR #-128  *(G) Return from routine

*End routine sort buffer

*Routine ARP handling
LDLI GR13 #h0010  *
LDHI GR13 #h8
LDH GR5 GR13  *Copy the address to the incoming ARP packet

Appendix B
LDLOD #1 GR5 GR13  
LDLI GR6 #0100  
LDHI GR6 #7  
LDLI GR12 #0  
LDHI GR12 #0  
LDLI GR11 #1  
LDHI GR11 #0  
LDLOD #4 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #5 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #6 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #0123  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #4567  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #89AB  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #0806  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDL GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #1 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #2 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #0002  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #0123  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #4567  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #89AB  
STLO GR7 GR6  
ADD20 GR12 GR11

*Load the address to the outgoing ARP packet
*offset=0
*Store 1 in GR11
*Store destination ethernet address
*Store source ethernet address(your own)
*Store Etherner type(ARP)
*Store hardware type
*Store protocol type
*Store HLen and PLen
*Store operation type
*Store target ethernet address

Appendix B
LDLOD #12 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #13 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  

LDLOD #4 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #5 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #6 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #7 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLOD #8 GR7 GR5  
STLO GR7 GR6  
ADD20 GR12 GR11  
LDLI GR7 #h5555  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
ADD20 GR12 GR11  
STLO GR7 GR6  
LDLI GR12 #0  
LDHI GR12 #0  
LDLI GR8 #1  
LDHI GR8 #0  
LDLI GR0 #h0  
LDHI GR0 #hC  
LDLI GR1 #1  
STL GR1 GR0  
LDLO GR14 GR6  
ADD20 GR12 GR8  
NOP  
NOP  

*Store target IP address  
*Store source ethernet address  
*Store source IP address  
*Padding 9*2=18 bytes  
*Offset=0  
*Configure memory arbiter  
*Write to ARP register  
*Repeated 30 times  
Appendix B
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
Appendix B
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8
NOP
NOP
LDLO GR14 GR6
ADD20 GR12 GR8

LDLI GR1 #0 *Configure memory arbiter
STL GR1 GR0

Appendix B
LDLI GR3 #9
LDHI GR3 #0
LDLI GR4 #0  *reset registers
LDHI GR4 #0
LDLI GR5 #0
LDHI GR5 #0
LDLI GR6 #0
LDHI GR6 #0
LDLI GR7 #0
LDHI GR7 #0
LDLI GR8 #0
LDHI GR8 #0
LDLI GR12 #1
LDHI GR12 #0
LDLI GR13 #h0007  *Load address to main loop
LDHI GR13 #h0
JMP GR13  *Jump to main loop
*End ARP routine

*Subroutine configure protocol processor
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Kristoffer Martinsson]