A Behavioral Model of a DSP Processor with Scalable Structure

Ulrik Lindblad
Patrik Thalin

Reg nr: LiTH-ISY-EX-3289-2002
2002-10-29
A Behavioral Model of a DSP Processor with Scalable Structure

Thesis for Degree of Master of Science
Department of Electrical Engineering
Division of Electronic Systems
Linköping University

by
Ulrik Lindblad
Patrik Thalin

Reg nr: LiTH-ISY-EX-3289-2002
In mobile digital devices, low power consumption is an important matter to reduce the need for a heavy and big battery. One way of reducing the power consumption is to construct the hardware so that the performance is optimal for the application. The demand of performance is dependent of the tasks that the device will be performing. This is where scalable structure of the hardware is an idea to solve the problem.

This master thesis serve as a starting point for developing a digital signal processor with scalable structure. The digital signal processor is a common and important part of digital processing. Scalable structure is in this case adding and removing parts of the memory and/or the instruction set, and to make the data wordlength variable. The development is simplified by modeling it on an existing processor. The result of this master thesis is an instruction simulator written in C language. The simulator will be a model for development of the hardware.

Nyckelord
Keywords
Simulator, Simulering, Simulation, Signal processing
Table of Contents

Chapter 1 Introduction 1
  1.1 Disposition 1
  1.2 Background 1
  1.3 Objectives 1
  1.4 Glossary 2

Chapter 2 Design Methods 5
  2.1 About hardware design in general 5
    2.1.1 Prestudy and requirements engineering 5
    2.1.2 System decomposition 5
    2.1.3 Validation and verification 6
  2.2 Design adaptation for this project 6
    2.2.1 Project prestudy and requirements 6
    2.2.2 System decomposition in practice 7

Chapter 3 Requirements 9
  3.1 General 9
  3.2 Motorola DSP56002 9
  3.3 Model 0.0 – The Simulator 10
    3.3.1 General requirements 10
    3.3.2 Data requirements 11
    3.3.3 Simulation 11
    3.3.4 Suggested modifications 12
  3.4 Model 1.x 13
  3.5 Model 2.x 13
  3.6 Model 3.x 13

Chapter 4 Simulator Design 15
  4.1 Comprehensive design 15
    4.1.1 User Interface (UI) 16
    4.1.2 Command Interpreter (CI) 16
    4.1.3 Program Control (PCU) 16
    4.1.4 Address generation (AGU) 16
    4.1.5 Instruction interpreter 16
    4.1.6 Execution 16
    4.1.7 Registers 16
    4.1.8 Program and data memory 16
    4.1.9 Log 16
  4.2 Detail design 17
    4.2.1 User Interface 17
    4.2.2 Command Interpreter 18
    4.2.3 Program Control 19
    4.2.4 Address generation 22
    4.2.5 Instruction interpreter 23
    4.2.6 Execution 23

Chapter 5 Simulator Implementation 25
  5.1 Description 25
    5.1.1 Language 25
    5.1.2 Files 26
    5.1.3 Doxygen 26
  5.2 Tools 27

Chapter 6 Result 29
  6.1 Implementation 29
    6.1.1 User Interface 29
    6.1.2 Command Interpreter 29
    6.1.3 Program Control 29
    6.1.4 Address generation 29
    6.1.5 Instruction interpreter 30
    6.1.6 Execution 30
    6.1.7 Registers 30
    6.1.8 Program and data memory 30
    6.1.9 Log 30
  6.2 Feasibility study 30
    6.2.1 Variable data wordlength 30
    6.2.2 Memories 31
    6.2.3 Instruction pipeline 31
    6.2.4 Interrupt 31
  6.3 Verification and validation 31
    6.3.1 Instruction decoding 31
    6.3.2 Binary arithmetics 32
    6.3.3 Operations 32
  6.4 Conclusions 32
  6.5 Planning of future work 33
    6.5.1 Proposed planning 33

Chapter 7 References 35
  7.1 Books 35
  7.2 Reports and papers 35
  7.3 Internet 35

Appendix A Instruction Set 37
  A.1 Instruction groups 37

Index 41
CHAPTER 1

INTRODUCTION

This chapter will explain the disposition of the report, the background and objectives of the project.

1.1 Disposition

Chapter 2 briefly describes the process of designing a DSP processor. The requirements in Chapter 3 lead to our design in Chapter 4. Chapter 5 describes the implementation, and in Chapter 6 the result and conclusions are presented.

1.2 Background

This project has been carried out at the division of Electronic Systems (ES) at the Department of Electrical Engineering (ISY) at Linköping University (LiU). The project was performed over a span of 20 weeks.

ES has a demand of a DSP processor kernel for low power research and education in SOC. This master thesis is the start of this long-time project of developing a DSP processor kernel at ES. The design of the processor is simplified by modelling it on an existing DSP processor architecture, in this case the Motorola DSP56002. This is a general purpose DSP processor with fixed point arithmetics and Harvard architecture.

1.3 Objectives

The goal of this project is to make a scalable behavioral model of the kernel that should be instruction compatible with the Motorola DSP56002. The scalability refers to variable data wordlength and addition or removal of memories and instructions. The goal with scalability is to reduce the power consumption,
however this investigation is not a part of this report. Instead the objective is to study if the architecture is suitable for scalability.

This model will also be used as an instruction simulator and a reference for development of the hardware description. Some features of the Motorola DSP56002 will not be implemented at this stage, e.g. pipeline and interrupts. The model will be designed in a manner that enables the introduction of these features later.

1.4 Glossary

This section describes the words and acronyms used in this thesis.

ALU
Arithmetic Logic Unit.

AGU
Address Generation Unit.

CI
Command Interpreter.

DSP
Digital Signal Processing.

Fixed-point
Number representation where the binary point remains in the same position.

Floating-point
Number represented as $a \cdot r^e$, where $a$ is called the mantissa, $r$ the radix and $e$ the exponent.

Golden Model
A reference model (verified to be working correctly) that will be used for verification of later models during development.

GUI
Graphical User Interface.

HDL
Hardware Description Language.

Harvard Architecture
Separation of data and program memories and buses.

Instruction
Used as an abbreviation for an assembler instruction. See also operation.

LA
Loop Address Register.
LC
Loop Counter Register.

LOD
A file format consisting of variable-length text records. Also called ASCII object module format (OMF).

MAC
Multiply and Accumulate.

Native data width
The native data width is the width of the data memories and most buses.

OMR
Operating Mode Register.

Opcode
Binary representation of an assembler instruction.

Operation
An assembler instruction can be divided into one or more operations. An operation is typically calculation or moving data.

PC
Program Counter Register.

PCU
Program Control Unit.

Reverse-carry
When using this address generation mode, the address modification is performed in hardware by propagating the carry in reverse direction. Reverse carry is equivalent to bit reversing the contents of address registers (Rn).

RTL
Register Transfer Level.

SOC
System On Chip.

SP
Stack Pointer Register.

SR
Status Register.

UI
User Interface.
This chapter gives an overview of the design methodology and the adaptations made for this project.

2.1 About hardware design in general

Several well-known and reliable design flows exist for hardware development, and although a thorough description of these would be beyond the scope of this thesis, many of them include typical activities well worth mentioning. For a more detailed design flow description, please refer to [1], [5] and [6].

2.1.1 Prestudy and requirements engineering

Before the actual design can take place, a prestudy has to be performed [5]. This activity will among other things typically involve investigation of the intended market and studies of already existing products. It will also serve as a feasibility study to assess whether or not there exist a solution to the problem which is both economically and technically feasible. The prestudy phase will result in a document called a requirements specification that in detail specify the tasks that the system shall perform. In the case of a hardware system, these requirements will often include design considerations such as physical size, power consumption and maximum life cycle etc.

2.1.2 System decomposition

Since many designs today are quite complex, the systems being developed often need to be divided into more manageable parts. This activity - often referred to as system partitioning - provides a good way of excluding details that, for some reason, are not of interest at a specific time during the development process.
In the case of DSP processor design it is therefore common to, at an early stage of development, implement a so called behavioral model using a high-level programming language such as C/C++ or Java. Typically this software model will at first include details like finite length data and finite length computing buffers while excluding timing aspects and pipeline effects.

An extensive discussion about partitioning techniques can be found in [1].

2.1.3 Validation and verification

Although the behavioral representation describes the functionality of the system, it does not include any details about how the actual hardware implementation should be done. Instead the main benefits of the behavioral model are:

1. It can be used by DSP application programmers for developing software for the processor. The benefit of this is that the software can be developed and debugged concurrently with the actual development of the processors hardware architecture.

2. If properly implemented, the behavioral model will provide measurable output (for example a log file containing memory and register dumps) that later can serve as a golden model for the actual hardware. This means that the output from the behavioral model can be used for verifying the functional correctness of the processor in every step of further development.

The verification of a hardware design project is of course a very important issue, and therefore a great deal of development time should be spent on this activity.

Explanations of the general concepts of validation and verification can be found in [1]. It also contains references to other literature within the area. A more detailed review of test and verification can be found in [6].

2.2 Design adaptation for this project

In the previous sections, some of the typical development activities for a general hardware project were described. This section instead focus on the practical design flow adaptations that were needed to be done for this project.

2.2.1 Project prestudy and requirements

Since the goal of this project is to develop a behavioral processor model used for research and educational purposes only, the prestudy needed not include a investigation of intended markets. Instead the main efforts were spent on gathering as much information about - and tools for - the target processor as possible. Since knowing that this is a project that will continue for a long time,
stretching far beyond the scope of this thesis, doing this seemed like time well spent. Another important activity was to check whether or not it would be feasible to modify the proposed processor to include variable data wordlength and support for addition or removal of parts of the memories and instruction set.

Since this project starts from an already existing design, many of the usual design issues need not be considered. For example, one of the first encountered activities will often be the design of an instruction set for the processor. Other design considerations would typically also include choosing native data length, degree of instruction set parallelism, number of memory accesses per clock cycle, number of pipeline stages, etc.

None of these design decisions were made, and instead our aim was to make the system almost compatible with the Motorola DSP56002. One exception from this was, of course, the previously mentioned modifications.

Requirements for the behavioral model were based upon several meetings with our supervisors and other colleagues at the department. It was decided that the general focus, at this time, should be on implementing solutions that could be easily understood. This in contrast to optimized, but often more complicated, solutions. The reason for making this decision was to make it easier to introduce new developers.

Requirements concerning issues mentioned in chapter 2.1.1 (chip size etc.) were not applicable since no actual hardware will be implemented yet. Instead the requirements for this project can be found in Chapter 3, “Requirements”.

2.2.2 System decomposition in practice

As mentioned before, this thesis will serve as a starting point for a much larger project taking place at ES, and a great deal of time have therefore been spent on planning forthcoming activities.

2.2.2.1 Planning the processor

When planning the work flow for implementing the processor, we decided to split the different implementations into so called models, as suggested in [1]. The models would all represent different stages of implementation refinement in which every new stage would add more detail to the system. The first one, called Model 0.0 or the instruction simulator, was the model that this thesis would focus on implementing.

A more detailed description of the different models and their intended use can be found in sections 3.3 to 3.6.
2.2.2.2 Planning the tools

Many of the algorithms used in digital signal processing today are quite complexed, and therefore it is often not reasonable implementing these using assembly language. Instead a high-level language such as C is a much more appropriate choice. Since there is no existing C compiler supporting the suggested modifications, this resulted in a thesis proposal for implementing a C compiler.

Verifying the functional correctness of the processor is a very time consuming and error prone task if performed manually. This means that time spent on developing verification tools can be easily justified. A tool for performing this verification has been designed and implemented. It compares the output from the simulator provided by Motorola with the output generated from our developed software model.
This chapter describes the requirements of the project and especially the simulator. To give a better understanding of the requirements, a short introduction to the Motorola DSP56002 core is given.

3.1 General

The reason for using the architecture from the Motorola DSP56002 is that it is a clean design that is straightforward without cache and floating-point arithmetics. There is also a lot of software developed for this core, for example compilers, debuggers, applications etc.

3.2 Motorola DSP56002

The Motorola DSP56002 is a general purpose DSP processor with triple-bus Harvard architecture which has a high degree of parallelism. It has fixed-point arithmetic and three function units: data arithmetic and logic unit (Data ALU), address generation unit (AGU) and program control unit (PCU). The Data ALU has a multiplication and accumulation (MAC) unit, two accumulators and two input registers. The AGU has two arithmetic units and 24 registers for calculation of addresses. The memories has its own buses for both data and addresses. This architecture makes it possible to do parallel data movement during data ALU operations. Figure 1 shows an overview of the architecture, for more details see [3].
3.3 Model 0.0 – The Simulator

This section describes the requirements for Model 0.0. The requirements are essentially a formalization of the description presented in section 1.3, “Objectives”. Requirements were extracted according to the description presented in section 2.2.1, “Project prestudy and requirements”.

3.3.1 General requirements

Descriptions of general requirements are presented in this section.

General requirement, user interface (GR1)

The provided user interface shall be very simple. The development of a graphical user interface can be done later as a project for a group of students in a software development course.

General requirement, instruction set (GR2)

The model shall support simulation of all instructions contained in the Motorola DSP56002 instruction set. The debugging instructions will not be supported at this stage since they are used only for debugging programs running on physical hardware. If instructions of this type are included in the program, the user will be informed about this. Support for adding and/or removing instructions should be possible to add in later versions of the model.
General requirement, variable data wordlength (GR3)
The model shall be designed in such a way that variable data wordlength can be introduced in later versions.

General requirement, memories (GR4)
The model shall be designed in such a way that additional memory units can be introduced or removed in later versions.

General requirement, instruction pipeline (GR5)
The model shall be designed in such a way that instruction pipeline can be introduced in later versions.

General requirement, interrupt (GR6)
The model shall be designed in such a way that interrupts can be simulated in later versions.

3.3.2 Data requirements
Descriptions of data requirements are presented in this section.

Input, loading memories (DR1)
It shall be possible to load a program into the program memory and data into the data memories by reading it from a file. The file format shall be a LOD-file containing hexadecimal information. The LOD-file is produced by converting an object file from the Motorola assembler. For details about the LOD-file see [3].

Output, general (DR2)
Contents of registers and memories shall, apart from the cases where readability is improved through use of other representation, be presented in hexadecimal form.

Output, saving contents of data memories (DR3)
It shall be possible to save the contents of data memories to a file. This file can then be used for verification of functionality.

3.3.3 Simulation
Descriptions of requirements related to simulation are presented in this section.
Simulation, logging of registers (SR1)

It shall be possible to receive a log-file that contains the contents of internal registers. The contents of all registers shall be saved after execution of any instruction. This is very important since it enables testing and verification as described in section 2.2.2.2, “Planning the tools”.

Simulation, simulation mode (SR2)

Simulations shall be possible to do in one of the following simulation modes:

- Continuous simulation
  Simulation runs continuously until one of the following events occur:
  - the user suspends execution
  - the program completes it’s execution
  - a breakpoint is encountered.

- Stepwise simulation
  Instructions are simulated one at a time by the user.

- Non-executing mode
  The simulation is stopped when in this mode, and the results of the simulation can be observed.

Simulation, changing simulation mode (SR3)

The user shall have the possibility to change simulation mode, this should only be possible while the simulator is in the non-executing mode.

3.3.4 Suggested modifications

This section suggests a suitable implementation order for further development of Model 0.0.

3.3.4.1 Model 0.1

Variable data wordlength.

3.3.4.2 Model 0.2

Graphical user interface.

3.3.4.3 Model 0.3

Adding of additional memory units and support for adding and/or removing instructions to/from the instruction set.
3.4 Model 1.x

The model differs from 0.x only by including timing and simulation of the instruction pipeline. Model 1.x can be used for studying the execution time (required number of instruction cycles) for different assembler instructions.

3.5 Model 2.x

This model constitute an architectural simulator built in accordance with Motorolas block diagram structure. The functionality of the individual blocks will be described at a behavioral level using a HDL.

3.6 Model 3.x

The model differs from 2.x only by implementing block functionality that can be synthesized (RTL). The blocks included in Model 2.x should, when replaced with synthesizable code, be used as a golden model for the blocks included in Model 3.x.
This chapter describes the design of Model 0.0. Note that Model 0.0 is also referred to as the instruction simulator, or just the simulator. The design is partitioned according to a partitioning technique called top-down, and therefore divided into two parts “Comprehensive design” and “Detail design”. The first part decompose the design into smaller parts called units. It will also give a brief description of the units, without any details. The second part describes the design of all the units in detail.

4.1 Comprehensive design

The top-level is decomposed into different units that perform a dedicated task. The partitioning is an iterative process that is done by grouping related functionality until manageable units are achieved. In Figure 2 the result of the partitioning is presented.

Figure 2  Block diagram
4.1.1 User Interface (UI)
This unit communicates with the user by accepting commands and presenting results. It also controls the flow of the simulation by telling the Command Interpreter to process the next instruction.

4.1.2 Command Interpreter (CI)
This unit serves as an interface between the UI and the rest of the simulator. The reason for separating it from the UI is that the UI shall easily be replaceable with a graphical user interface.

4.1.3 Program Control (PCU)
This unit handles and updates the system stack, condition codes, operating modes and the program counter. The registers used are, like all other, placed in the register unit.

4.1.4 Address generation (AGU)
This unit generates addresses to the program and data memories. Supported arithmetics are linear, modulo and reverse-carry.

4.1.5 Instruction interpreter
This unit fetches and decodes the instructions and decides which operations to invoke.

4.1.6 Execution
This unit performs the operations corresponding to the instruction decoded in the instruction interpreter.

4.1.7 Registers
All internal registers and the system stack are represented in this unit.

4.1.8 Program and data memory
Representation of the memories is situated in this unit.

4.1.9 Log
This unit can write the contents of all registers to a log-file.
4.2 Detail design

This part describes the details of the functionality of the units. Sequence diagrams are used to determine which functions that are needed. Since many of the operations are trivial there are no diagrams for them. One case that is more complicated is the execution of an instruction. The diagram in Figure 3 shows which functions that are needed for execution of an instruction.

The user invokes the step command through the UI. This information is passed to the CI which updates the program counter and reads the instruction from the program memory. The instruction is then decoded and the corresponding operations are called in the execution unit. The control is then returned to the UI. Void means returning no value.

**Figure 3** Sequence diagram for execution of instruction

Below, all the units and their functions are described. The arguments for the functions are not included, for details about this please refer to [4].

4.2.1 User Interface

The UI calls functions in the CI and has its own functions for printing register contents when requested by the user.
4.2.1.1 Calls to Command Interpreter

**Breakpoint**
Add, remove, and list breakpoints.

**System command**
Execute a system command, e.g. ls or pwd.

**Load**
Load the contents of a LOD-file into the memories.

**Log**
Start and stop logging of registers to file.

**Run**
Start continuous simulation.

**Step**
Execute one instruction.

4.2.1.2 Internal functions

**Quit**
Exit the simulator.

**Display register**
Display contents of a register.

**Help**
Print help for the simulator.

**Reset**
Restore the program counter to its initial value.

4.2.2 Command Interpreter
This unit controls the simulation with the following functionality.

4.2.2.1 Breakpoint
Breakpoints are used for halting the simulation at a specific point in the program. The simulation stops before execution of the instruction at the breakpoint. These are the functions for breakpoint handling:
Add breakpoint
Add a breakpoint at a given address.

Remove breakpoint
Remove a breakpoint at a given address.

Test breakpoint
Test if a breakpoint is set for an address.

4.2.2.2 Simulation
Two functions are used for simulation of instructions:

Step
This function execute one instruction.

Run
This function call the step function repeatedly until one of the following events occur:
- The user press CTRL + C
- The STOP instruction is found
- A break point is set for the next instruction.

Note
If none of this events occur, the program will execute forever.

4.2.3 Program Control
The PCU performs the standard program flow control using a stack, the status register and by updating the program counter. It also handles the operating mode register.

4.2.3.1 Status register (SR)
The SR has the following condition code bits:
- LF - Loop flag
- DM - Double precision multiply mode
- T - Trace mode
- S1, S0 - Scaling mode
- I1, I0 - Interrupt mask
- S - Scaling
- L - Limit
- E - Extension
• U - Unnormalized
• N - Negative
• Z - Zero
• V - Overflow
• C - Carry.

For an explanation of the functionality of the bits see [3] and [4].

Functions for the SR are:

**Set SR**  
Set a bit in SR to one.

**Clear SR**  
Set a bit in SR to zero.

**Get SR**  
Read a bit in SR.

**Update**  
Update a bit in SR according to the condition code computation for that bit. For definitions of the condition code computations, see [3].

### 4.2.3.2 Operating mode register (OMR)

The operating mode register determine the memory maps for program and data memories, and the start-up procedure when the DSP processor leaves the reset state. Most of this functionality is not interesting at this stage, but it is still implemented for compatibility reasons. The bits in OMR are:

• MA - Operating mode A
• MB - Operating mode B
• MC - Operating mode C
• DE - Data ROM enable
• YD - Internally Y memory disable
• SD - Stop delay.

For an explanation of the functionality of the bits see [3] and [4].

Functions for the OMR are:

**Set OMR**  
Set a bit in OMR to one.
Clear OMR
Set a bit in OMR to zero.

Get OMR
Read a bit in OMR.

4.2.3.3 Stack
The stack stores the PC and SR during subroutine calls, interrupts and program looping. It will also store LA and LC to support nested loops. A stack error occurs when trying to pop an empty stack or push a full stack. Underflow only occurs when trying to pop an empty stack.

Set stack underflow
Set stack underflow bit (UF) to one.

Clear stack underflow
Set stack underflow bit (UF) to zero.

Get stack underflow
Read stack underflow bit (UF).

Set stack error
Set stack error bit (SE) to one.

Clear stack error
Set stack error bit (SE) to zero.

Get stack error
Read stack error bit (SE).

Set SP
Set stack pointer to an address.

Clear SP
Set stack pointer to zero.

Get SP
Read stack pointer address.

Push
Push selected registers on to the system stack.
Pop
Pop the system stack.

Top
Read top of system stack.

4.2.3.4 Program counter
The program counter contains the address of the next instruction in the program memory.

Update PC
Increment program counter.

4.2.4 Address generation
The AGU handles three types of registers. Address registers, offset registers and modifier registers. The address registers contain the actual addresses that are transferred to the address buses. The offset registers contain values used for updating the address registers. Modifier registers specify the type of arithmetics used during address register update calculations.

Address increment
Add one to an address register.

Address decrement
Add minus one to an address register.

Offset add
Add an offset register to an address register.

Offset add 2’s-complement
Add the 2’s-complement of an offset register to an address register.

Modulo
Perform the mathematical mod operation. Modulo value minus one is stored in a modifier register.

Reverse-carry increment
Add one to an address register with reverse-carry propagation.

Reverse-carry decrement
Add minus one to an address register with reverse-carry propagation.
Reverse-carry sub
Subtract an offset register from an address register with reverse-carry propagation.

4.2.5 Instruction interpreter
The instruction interpreter has the main functions listed below.

Decode instruction
Decode an opcode and decide which operations that will be executed.

Decode parallel move
Decode an opcode and decide which parallel move operations that will be performed.

4.2.6 Execution
This unit has operations for all instructions, some instructions has more than one corresponding operation. Since there are 67 instructions they will not be listed here. For a list of instructions see Appendix A, and for details about operation definitions see [4].

4.2.7 Registers
The registers has the datatype array with values 0 or 1. Some registers are a part of a longer register, see Figure 4. When writing to X this means the concatenation of X0 and X1, for A it is A0, A1 and A2 etc.

Figure 4 Example of registers

<table>
<thead>
<tr>
<th>X</th>
<th>X0</th>
<th>X1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A0</td>
<td>A1</td>
</tr>
</tbody>
</table>

4.2.8 Program and data memory
The memories are internally represented as arrays of integers. At first they are typically filled with data from a LOD-file through a load operation invoked in the CI. The hexadecimal data in the LOD-file is then converted before inserted into the memories.

Initialize memory
Initialize a memory by filling the entire memory with a specified value.
Read memory
Read data from a specific memory at a specified address.

Write memory
Write data to a specific memory at a specified address.

4.2.9 Log
For log-functionality the two functions below are used.

Start log
Create a new empty log-file.

Write to log
Append a new log-entry, with all register contents, to the end of the log-file.
This chapter describes how the implementation of the simulator is done and the choice of language is motivated. Also the tools and methods used are described.

### 5.1 Description

The simulator is written in C and documented with Doxygen, see section 5.1.3, “Doxygen”. The code is split in separate files for convenience. All the files are in a CVS for version management. For a description of the tools used, please refer to section 5.2, “Tools”.

### 5.1.1 Language

As mentioned in section 2.2.1, “Project prestudy and requirements” the implementation should be easily understood. Therefore an object oriented programing language like C++ or Java would be unsuitable since they tend to be hard to understand and maintain for long-time projects like this. Instead a language like C gives a better result for this purpose. In this project ANSI-C is used [2]. The compiler is Gcc under UNIX, the simulator is also compiled under MS Windows using Cygwin [8]. Cygwin is a UNIX environment for Windows.
5.1.2 Files

All the C code source files are arranged according to the units. There are also some files that contain general functionality used by the other files. Table 1 shows all the files for the simulator.

Table 1  Files for the simulator

<table>
<thead>
<tr>
<th>Code</th>
<th>Header</th>
<th>Unit</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.c</td>
<td>ui.h</td>
<td>User Interface</td>
<td></td>
</tr>
<tr>
<td>ci.c</td>
<td>ci.h</td>
<td>Command Interpreter</td>
<td></td>
</tr>
<tr>
<td>pcu.c</td>
<td>pcu.h</td>
<td>Program Control (PCU)</td>
<td></td>
</tr>
<tr>
<td>agu.c</td>
<td>agu.h</td>
<td>Address generation (AGU)</td>
<td></td>
</tr>
<tr>
<td>decode.c</td>
<td>decode.h</td>
<td>Instruction interpreter</td>
<td></td>
</tr>
<tr>
<td>operations.c</td>
<td>operations.h</td>
<td>Execution</td>
<td></td>
</tr>
<tr>
<td>global.c</td>
<td>global.h</td>
<td>Registers</td>
<td>Also global variables</td>
</tr>
<tr>
<td>memory.c</td>
<td>memory.h</td>
<td>Program and data memory</td>
<td></td>
</tr>
<tr>
<td>log.c</td>
<td>log.h</td>
<td>Log</td>
<td></td>
</tr>
<tr>
<td>dsp_conv.c</td>
<td>dsp_conv.h</td>
<td>Used by other units</td>
<td>Converting data types</td>
</tr>
<tr>
<td>dsp_math.c</td>
<td>dsp_math.h</td>
<td>Used by other units</td>
<td>Binary arithmetics</td>
</tr>
<tr>
<td>makefile</td>
<td></td>
<td></td>
<td>Used to build the code</td>
</tr>
<tr>
<td>dsp.dox</td>
<td></td>
<td></td>
<td>Configuration for doxygen</td>
</tr>
</tbody>
</table>

5.1.3 Doxygen

All the source code is documented with Doxygen to make it easier for developers to understand the implementation. Doxygen generates HTML and LaTeX files that contain hyperlinks to files and functions. LaTeX files can then be converted to postscript and pdf files. Since the documentation is kept in the source files, it is easy to keep it consistent with the code. Below is a typical Doxygen source code comment:

```c
unsigned long power(unsigned long x, unsigned long y)
/**
 * Calculate x^y.
 * \param x x-value
 * \param y y-value
 * \return x^y as unsigned long (Integer)
 */
{
...
}
```
Below the corresponding generated documentation is listed, note that functions that use this function are also automatically listed.

```c
unsigned long power ( unsigned long x
                             unsigned long y
                     )

Calculate \(x^y\).
Parameters:
  x  x-value
  y  y-value

Returns:
\(x^y\) as unsigned long (Integer)
Definition at line 4 of file dsp_math.c.
Referenced by convert_array2long(), and convert_long2array().
```

Doxygen can also generate diagrams that show file dependencies. Figure 5 show a diagram where the files decode.c, dsp_conv.c, memory.c and operations.c all include the header file dsp_conv.h. This is useful for understanding how files are used and can be used.

**Figure 5** Include graph

![Include graph](decode.c dsp_conv.c memory.c operations.c dsp_conv.h)

### 5.2 Tools

The tools listed below are used in the implementation.

- Emacs - editor
- Gcc - C compiler
- Make - build the source code
- GDB - debugger
- CVS - version control system
• Doxygen - source code documentation [7]
• Cygwin - gnu tools under MS Windows [8].
This chapter describes the result and conclusions of the project.

6.1 Implementation
This section describes the status of the implementation of the units. Due to lack of time, some of the units are not yet fully functional.

6.1.1 User Interface
This unit is fully functional.

6.1.2 Command Interpreter
This unit is fully functional.

6.1.3 Program Control
The implementation of some condition codes and the updating of the program counter is not completely implemented.

6.1.4 Address generation
This unit is not implemented, and therefore the x- and y-memories cannot be accessed yet.
6.1.5 Instruction interpreter

Most of the functionality of this unit is implemented. Decoding of condition codes and extended precision register references need to be added, see [3]. Also, only one type of parallel data movement - immediate short - is supported at this time.

6.1.6 Execution

Operations for the jump instructions, hardware loop and some other instructions are not implemented. All the MAC unit instructions, with only a few exceptions, are implemented and therefore simple programs can be executed. However some of the condition codes are not supported yet, but the most important such as zero, negative, carry and a few other flags are implemented. For more details about the instruction set see Appendix A.

6.1.7 Registers

This unit is fully functional.

6.1.8 Program and data memory

Most of the functionality of this unit is implemented. However, saving memory contents is not yet implemented. Also it is not possible, at this time, to perform long read and write operations accessing both data memories concurrently. For details about long read and write operations please refer to [3].

6.1.9 Log

This unit is fully functional.

6.2 Feasibility study

This section describes what we believe will be possible to do with this architecture in the future. The focus of this project has been the feasibility study of introducing variable wordlength. Since introducing variable wordlength will affect many parts of the design, we thought this had to be considered at an early stage of the development. This to make sure that it would be possible to implement. Memories, instruction pipeline and interrupts could be considered later due to the fact that they do not affect the design as much as the wordlength.

6.2.1 Variable data wordlength

Since future models shall have variable wordlength in the data path, all implemented functionality already support this. Simple tests made so far
indicate that it works correctly, but no formal verification has been done at this time. The supported data wordlength variation is minimum 16 bits and maximum 24 bits. The lower limit is due to the fact that the contents of the address bus, consisting of 16 bits, can be transferred via the data bus. This could cause addressing errors if the width of the data bus would be less than the width of the address bus. The upper limit is not really a limit, but since the original processor has 24 bits and the goal was to reduce power consumption, exceeding that limit would make no sense.

6.2.2 Memories
Since the memories are not used yet it is hard to tell if it is possible to add new memories, but nothing so far have indicated any problems.

6.2.3 Instruction pipeline
Adding an instruction pipeline will probably cause some problems. No detailed documentation is available about the implementation of the pipeline, and therefore it will have to be implemented using only the timing information provided in [3].

6.2.4 Interrupt
This part has unfortunately not been studied in detail, no conclusion can be drawn at this time.

6.3 Verification and validation
This section describes the approach used for, and the result of, the verification process in this project.

6.3.1 Instruction decoding
Instruction decoding has been tested by writing test programs (in assembly language) that use all the instructions in the instruction set. Although this decoding has been verified to work correctly, we would recommend further testing in the future. The decoding should be tested with different operands and with random ordering of the assembly instructions. Of course a large number of tests will result in a better verification certainty, and therefore it might be a good idea to develop a program that generate random (but syntactically correct) assembly language programs. The decoding and invoking of operations can then be verified by using the verification tool that was created within this project. See section 2.2.2.2, “Planning the tools” for further details about this tool.
6.3.2 Binary arithmetics

The binary arithmetic in `dsp_math.h` has been tested by comparing results from the implementation of the operation and a normal standard C operation. Since the arithmetic uses an array of bits, the comparison cannot be done directly. Therefore, the functions implemented in `dsp_conv.h` are used for conversion to and from binary representation and floating-point types. The main functions in `dsp_math.h` are addition and multiplication, in Figure 6 there is a diagram that shows the work flow when testing addition. The test values have been both random values and selected values that are critical e.g. one, minus one and zero. The extent of the test is enough to be sure that the operations work as they are supposed to.

![Figure 6](image_url)  
Test of binary addition

6.3.3 Operations

The operations are not formally tested, but simple manual inspections have been done with good results. Since many of the operations use the binary arithmetics in `dsp_math.h`, this is not as critical as the updating of condition codes. Since not all condition codes are supported yet, a more extensive test is hard to accomplish.

6.4 Conclusions

The aim of this thesis has been to serve as the starting point for a development project taking place at ES. It was decided that an already existing processor, the
Motorola DSP56002, should be used as the architectural model for the implementation. This approach has the advantage of reducing the risk of ending up with an erroneous design. Due to our lack of experience in DSP processor design, we feel that this idea has been a good approach. A disadvantage of this method is that it sometimes has been hard to understand the more complex parts of the design and also to find the correct information in the extensive documentation.

6.5 Planning of future work

Since this project is intended to have a long life cycle, the documentation is important, especially when new developers join the project. A set of documents describing development and usage within the project have to be produced.

To verify functional correctness, no new models will be implemented before the completion of model 0.0. One exception from this will be the implementation of model 0.2.0, adding a graphical user interface to the simulator. This will not affect the functionality of the simulator, and it can therefore be implemented concurrently with model 0.0.

The verification process will first focus on verifying correct functionality of condition code generation and instruction decoding. This will be done using the original data wordlength so that the simulator for the Motorola 56002 can be used as a golden model. This means that although variable data wordlength have already been implemented in model 0.0, the verification of this will not be done until the development of model 0.1.

In Figure 7, the dependency graph of the implementation order for the different models are shown. The arrows indicate dependency between development of models. Developing model 0.2.1, for example, would require completion of model 0.2.0 as well as model 0.1.

6.5.1 Proposed planning

As seen in Figure 7, there are several possibilities of development order for the models. However, we propose that the models to the left in the figure (0.0, 0.2.0, 1.0, 2.0 and 3.0) are implemented first. This implementation order will yield a basic foundation to be used for further development. The C compiler and the assembler can be developed independently from the models, and the development should start as soon as possible.

The development of an assembler, a C compiler and a graphical user interface for model 0.0 are intended to be a part of a master thesis and/or as an assignment for a group of students in a software development course. Project proposals for these activities have already been written, but we need to specify
the goals more in detail. We also need to appoint responsibilities among us for supervising these different projects.

**Figure 7**  Dependency graph for the implementation order

Model Description  
M0.x  Simulator (Software)  
M1.x  Simulator with timing  
M2.x  Behavioral HDL model  
M3.x  Synthesized RTL model  
Mx.0  Basic model  
Mx.1  Variable data wordlength  
Mx.2.x GUI  
Mx.3  Memory and instruction set modifications.
CHAPTER 7

REFERENCES

7.1 Books

7.2 Reports and papers

7.3 Internet
This appendix lists all instructions for the Motorola DSP56002 and the status of implementation in model 0.0.

A.1 Instruction groups

The instructions are arranged in groups according to their functionality. The * indicate that the instruction allows parallel data move.

Table 2 Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ADC</td>
<td>Add long with carry*</td>
<td>Not implemented</td>
</tr>
<tr>
<td>ADD</td>
<td>Add*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ADDL</td>
<td>Shift left then add*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ADDR</td>
<td>Shift right then add*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ASL</td>
<td>Arithmetic shift accumulator left*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic shift accumulator right*</td>
<td>Implemented</td>
</tr>
<tr>
<td>CLR</td>
<td>Clear accumulator*</td>
<td>Implemented</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare*</td>
<td>Not implemented</td>
</tr>
<tr>
<td>CMPM</td>
<td>Compare magnitude*</td>
<td>Not implemented</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement accumulator</td>
<td>Implemented</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide iteration</td>
<td>Not implemented</td>
</tr>
<tr>
<td>INC</td>
<td>Increment accumulator</td>
<td>Implemented</td>
</tr>
<tr>
<td>MAC</td>
<td>Signed multiply-accumulate*</td>
<td>Implemented</td>
</tr>
<tr>
<td>MACR</td>
<td>Signed multiply-accumulate and round*</td>
<td>Implemented</td>
</tr>
<tr>
<td>MPY</td>
<td>Signed multiply*</td>
<td>Implemented</td>
</tr>
</tbody>
</table>
### Table 2  
**Arithmetic Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPYR</td>
<td>Signed multiply and round*</td>
<td>Implemented</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate accumulator*</td>
<td>Implemented</td>
</tr>
<tr>
<td>NORM</td>
<td>Normalize accumulator iteration</td>
<td>Not implemented</td>
</tr>
<tr>
<td>RND</td>
<td>Round accumulator*</td>
<td>Implemented</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract long with carry*</td>
<td>Implemented</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract*</td>
<td>Implemented</td>
</tr>
<tr>
<td>SUBL</td>
<td>Shift left then subtract*</td>
<td>Implemented</td>
</tr>
<tr>
<td>SUBR</td>
<td>Shift right then subtract*</td>
<td>Implemented</td>
</tr>
<tr>
<td>Tcc</td>
<td>Transfer Conditionally</td>
<td>Not implemented</td>
</tr>
<tr>
<td>TFR</td>
<td>Transfer data ALU register*</td>
<td>Not implemented</td>
</tr>
<tr>
<td>TST</td>
<td>Test accumulator*</td>
<td>Implemented</td>
</tr>
</tbody>
</table>

### Table 3  
**Logical Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Logical AND*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ANDI</td>
<td>AND Immediate with control register</td>
<td>Implemented</td>
</tr>
<tr>
<td>EOR</td>
<td>Logical exclusive OR*</td>
<td>Implemented</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical shift accumulator left*</td>
<td>Implemented</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift accumulator right*</td>
<td>Implemented</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical complement on accumulator*</td>
<td>Implemented</td>
</tr>
<tr>
<td>OR</td>
<td>Logical inclusive OR*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ORI</td>
<td>OR Immediate with control register</td>
<td>Implemented</td>
</tr>
<tr>
<td>ROL</td>
<td>Rotate accumulator left*</td>
<td>Implemented</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate accumulator right*</td>
<td>Implemented</td>
</tr>
</tbody>
</table>

### Table 4  
**Bit Manipulation Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCHG</td>
<td>Bit test and change</td>
<td>Not implemented</td>
</tr>
<tr>
<td>BCLR</td>
<td>Bit test and clear</td>
<td>Not implemented</td>
</tr>
<tr>
<td>BSET</td>
<td>Bit test and set</td>
<td>Not implemented</td>
</tr>
<tr>
<td>BTST</td>
<td>Bit test on memory</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>
### Table 5  Loop Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO</td>
<td>Start hardware loop</td>
<td>Not implemented</td>
</tr>
<tr>
<td>ENDDO</td>
<td>Exit from hardware loop</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

### Table 6  Move Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUA</td>
<td>Load updated address</td>
<td>Not implemented</td>
</tr>
<tr>
<td>MOVE</td>
<td>Move data</td>
<td>Not implemented</td>
</tr>
<tr>
<td>MOVEC</td>
<td>Move control register</td>
<td>Not implemented</td>
</tr>
<tr>
<td>MOVEM</td>
<td>Move program memory</td>
<td>Not implemented</td>
</tr>
<tr>
<td>MOVEP</td>
<td>Move peripheral data</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

### Table 7  Program Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG</td>
<td>Enter debug mode</td>
<td>Not supported</td>
</tr>
<tr>
<td>DEBUGcc</td>
<td>Enter debug mode conditionally</td>
<td>Not supported</td>
</tr>
<tr>
<td>ILLEGAL</td>
<td>Illegal instruction interrupt</td>
<td>Not supported</td>
</tr>
<tr>
<td>Jcc</td>
<td>Jump conditionally</td>
<td>Not implemented</td>
</tr>
<tr>
<td>JCLR</td>
<td>Jump if bit clear</td>
<td>Not implemented</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>Not implemented</td>
</tr>
<tr>
<td>JScc</td>
<td>Jump to subroutine conditionally</td>
<td>Not implemented</td>
</tr>
<tr>
<td>JSCLR</td>
<td>Jump to subroutine if bit clear</td>
<td>Not implemented</td>
</tr>
<tr>
<td>JSET</td>
<td>Jump if bit set</td>
<td>Not implemented</td>
</tr>
<tr>
<td>JSSET</td>
<td>Jump to subroutine if bit set</td>
<td>Not implemented</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to subroutine</td>
<td>Not implemented</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>Implemented</td>
</tr>
<tr>
<td>REP</td>
<td>Repeat next instruction</td>
<td>Not implemented</td>
</tr>
<tr>
<td>RESET</td>
<td>Reset on-chip peripheral devices</td>
<td>Not implemented</td>
</tr>
<tr>
<td>RTI</td>
<td>Return from interrupt</td>
<td>Not implemented</td>
</tr>
<tr>
<td>RTS</td>
<td>Return from subroutine</td>
<td>Not implemented</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop processing (low power standby)</td>
<td>Implemented</td>
</tr>
<tr>
<td>SWI</td>
<td>Software interrupt</td>
<td>Not supported</td>
</tr>
<tr>
<td>WAIT</td>
<td>Wait for interrupt (low power standby)</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
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 icke-komersiell 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 nämnt 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/

© Ulrik Lindblad and Patrik Thalin