Benchmarking a DSP processor

Per Lennartsson & Lars Nordlander

LITH-ISY-EX-3261-2002

Linköping 2002-12-05
Benchmarking a DSP processor

Per Lennartsson & Lars Nordlander

Reg nr: LiTH-ISY-EX-3261

Supervisor: Eric Tell
Examiner: Dake Liu

Linköping 2002
Benchmarking av en DSP processor

Per Lennartsson and Lars Nordlander

Sammanfattning

This Master thesis describes the benchmarking of a DSP processor. Benchmarking means measuring the performance in some way. In this report, we have focused on the number of instruction cycles needed to execute certain algorithms. The algorithms we have used in the benchmark are all very common in signal processing today.

The results we have reached in this thesis have been compared to benchmarks for other processors, performed by Berkeley Design Technology, Inc.

The algorithms were programmed in assembly code and then executed on the instruction set simulator. After that, we proposed changes to the instruction set, with the aim to reduce the execution time for the algorithms.

The results from the benchmark show that our processor is at the same level as the ones tested by BDTI. Probably would a more experienced programmer be able to reduce the cycle count, especially for some of the more complex benchmarks.

Nyckelord

Benchmarking, DSP processor, digital signal processing, algorithm kernels
Abstract

This Master thesis describes the benchmarking of a DSP processor. Benchmarking means measuring the performance in some way. In this report, we have focused on the number of instruction cycles needed to execute certain algorithms. The algorithms we have used in the benchmark are all very common in signal processing today.

The results we have reached in this thesis have been compared to benchmarks for other processors, performed by Berkeley Design Technology, Inc.

The algorithms were programmed in assembly code and then executed on the instruction set simulator. After that, we proposed changes to the instruction set, with the aim to reduce the execution time for the algorithms.

The results from the benchmark show that our processor is at the same level as the ones tested by BDTI. Though would a more experienced programmer probably be able to reduce the cycle count even more, especially for some of the more complex benchmarks.

Linköping in December 2002

Per Lennartsson and Lars Nordlander
# Contents

Chapter 1  About this thesis ................................. 5  
1.1 Purpose of the processor .................................. 5  
1.2 Way of work .................................................. 5  
1.3 Reading guidelines ......................................... 5  
1.4 Acknowledgements ......................................... 5  
Chapter 2  Introduction ........................................ 7  
2.1 Introduction to DSP ......................................... 7  
Chapter 3  Architecture of the processor ............... 9  
3.1 Processor features .......................................... 9  
3.2 Data Path ..................................................... 9  
3.2.1 Arithmetic Unit .......................................... 11  
3.2.2 Shift Unit .................................................. 11  
3.2.3 Logic Unit .................................................. 11  
3.2.4 MAC Unit ................................................... 11  
3.2.4.1 Rounding .............................................. 12  
3.2.4.2 Saturation Unit ........................................ 12  
3.2.5 Count Unit .................................................. 12  
3.2.6 Register File .............................................. 12  
3.2.7 The Accumulator Registers ......................... 13  
3.2.8 Addressing ............................................... 13  
3.2.8.1 Addressing Modes ................................... 13  
3.2.8.2 Modulo Addressing ................................ 14  
3.2.8.3 Bit-Reversed Addressing ......................... 14  
3.3 Control Path ................................................. 14  
3.3.1 The Pipeline .............................................. 15  
3.3.2 Instruction Decoder ...................................... 15  
3.3.3 Program Counter ........................................ 15  
3.3.4 Program Flow Controller ............................ 15  
3.3.4.1 Subroutine Calls - The PC Stack ................ 16  
3.3.4.2 Hardware Looping .................................. 16  
3.3.5 Pipeline Controller ...................................... 16  
3.3.6 Branch Controller ...................................... 17  
Chapter 4  Memory setup ...................................... 19  
4.1 Overview ...................................................... 19  
4.2 Status address .............................................. 19  
4.3 Control address ............................................. 19  
Chapter 5  Instruction set ....................................... 21  
5.1 The instruction word .................................... 21  
5.2 Parallel Memory Instructions ......................... 22  
5.2.1 Move to memory ....................................... 22  
5.2.2 MAC operation and load ............................ 22  
5.2.3 Arithmetic, logic, shift or move operation and load 22  
5.3 Instruction set restrictions .............................. 23  
5.3.1 Branch and jump instructions .................... 23  
5.3.2 Hardware loops ....................................... 23  
5.3.3 Modulo addressing ................................... 23  
5.4 Instruction encoding ..................................... 24
Chapter 1   About this thesis

This Master thesis describes the process of benchmarking a digital signal processor (DSP-processor), in the aspect of execution time. We performed the benchmark on an existing DSP-processor, created by PhD-student Eric Tell at the Department of Computer Engineering at Linköping Institute of Technology.

1.1 Purpose of the processor
The processor is intended to be used as a platform for hardware accelerators. That is, it should be possible to easily connect application specific hardware units to the processor core. For this reason the instruction set of the processor is quite simple and instead space for adding hardware accelerator instructions has been reserved.

1.2 Way of work
The benchmarking was performed by writing algorithms in assembly code and then executing them in the instruction set simulator, which simulates the execution of programs on the processor. The aim was to achieve as short execution time (number of instruction cycles) as possible, by optimizing the assembly code by hand. Different implementations on other processors were studied, to get as good knowledge as possible on how to write the assembly code as time efficient as possible.
After this was done, we suggested changes in the instruction set so that the execution time was shortened (number of instruction cycles), and re-implemented the algorithms with the modified instruction set.

1.3 Reading guidelines
In the first part of the thesis a thorough description of the architecture of the processor is found.

In the second part, the benchmarking is found. First, the algorithm is described and its implementation in “words”, so that a better understanding is achieved. After that, the assembly code is presented.

Changes that were performed on the processor are then presented. Notice that the code in the benchmarking chapter has been written with the modified instruction set. Finally, our conclusions and thoughts around this thesis are described. In the appendix, a summary and explanation of the instruction set is presented.

1.4 Acknowledgements
The work was performed at the Department of Computer Engineering at Linköping Institute of Technology. We wish to express our gratitude to our examiner, Professor Dake Liu, and our instructor PhD student Eric Tell, for your assistance during our work.
2.1 Introduction to DSP

Digital Signal Processing, abbreviated DSP, is the application of mathematical operations to digitally represented signals. Strictly speaking, any microprocessor that processes signals could be referred to as a DSP-processor, but in practice only processors that are specifically designed for signal processing are called DSP-processors. It is very common that signal processing is computationally and numerically intense, and hence a DSP-processor often has special architecture to support this. Maybe the most common feature is the MAC unit. MAC is an abbreviation for multiply and accumulate. This is very useful when computing a convolution for instance. Also, DSP-processors often contain architecture to support multiple memory accesses, for instance loading a filter coefficient in parallel with a filter sample and at the same time, loading an instruction. Specialized addressing modes are also typical for a DSP-processor. Often the processor has dedicated address generation units. This feature makes it possible to compute addresses in the “background”, and in parallel with arithmetic instructions.

The most common DSP-algorithms include speech coding and decoding, echo- and noise cancellation, spectral estimation, image compression and decompression and so on. The high performance processors are found in for instance radar systems and seismic imaging.

The main components of a DSP-processor can be seen in the figure below. Two different memories are used. The data memory contains the samples of the signal we wish to process. In the tap memory, also called coefficient memory, for instance filter coefficients are stored.

![Figure 2.1, overview of a DSP - processor.](image)

Fixed point processor data paths, as the one used in this thesis, mainly incorporates a multiplier, an arithmetic logic unit (ALU), shifters, registers and accumulators. Address calculation units are often separated from the data path and use their own hardware to compute addresses. A structural model of a MAC-unit can be seen in figure 2.2 below.
Also incorporated in the data path is the ALU. It has dedicated hardware for arithmetic operations, logic operations and a variety of shift operations.

Buses in a DSP-chip connect the memories with the computing components. The bus connections are controlled by the control path. The bus systems differs from a normal CPU because data allocation and execution scheduling is relatively known, and hence a more parallel architecture can be used. This concept makes it possible to design DSP-processors with higher performance compared to a regular CPU.

The main task of the control path is to control the data path. It can be seen in the following figure. Basically it consists of two finite state machines, FSM:s. The first FSM takes care of the booting sequence, and the other one is the program flow controller, PFC. The PFC reads instructions one by one into the instruction decoder, which in turn sends out control signals to the data path.

\[ ACR <= ACR + A * B \]
Chapter 3  Architecture of the processor

In this chapter we describe the architecture of the processor. In the figure below, an overview of the architecture is shown.

3.1 Processor features
This processor uses a 32-bit instruction word. The native data width is 16 bits and it uses fixed point representation. It has a multiply and accumulate unit (MAC) with a 32-bit multiplier and a 40-bit accumulator. This means that eight guard bits are used. It also contains a 40-bit barrel shifter for scaling operations and other purposes. The processor supports some parallelism, especially for convolution based algorithms. Other features are support for zero overhead hardware looping, modulo- and bitreversed addressing. The instruction set has a great deal of free space, which can be used for future additions. This is because the processor is intended as a platform for hardware acceleration units.

3.2 Data Path
The computational units of the processor are the following:
Arithmetic unit for addition, subtraction and other common arithmetic operations
Logic unit for bitwise ‘and’, ’or’ ’xor’ and ‘not’ operations.
Shift unit for arithmetic shift, logic shift and rotation operations.
MAC unit for multiplication and multiply-accumulate operations. The MAC can also perform 32-bit arithmetic shift and 32-bit addition/subtraction.
Count unit for counting leading ones or zeros.
The data path architecture can be seen in figure 3.2.
Operands are always taken from the 32x16 bit register file. All of the 32 registers can be used as general purpose registers for common arithmetic, shift and logic operations, but most of them also have other functions. Particularly four of these 32 registers are also used as 40-bit accumulator registers for MAC operations. Some of the registers are also used as address
registers or for other address generation purposes. All of the registers are memory mapped, which means that they can be addressed externally. Data exchange between computational units, register file and memories are facilitated by the following busses:
Two 16-bit data busses DA and DB that provides operands for computational units and data to memories.
Two 16-bit result buses RA and RB for sending result from computations and data from memory to the register file.
One 40-bit accumulator register bus from register file to the MAC unit and one 40-bit bus from the MAC unit back to the register file.

At the most one 40-bit word and two 16-bit words can be written to the register file in one clock cycle.
The architecture also has two address busses AA and AB so two memories can be addressed simultaneously.

3.2.1 Arithmetic Unit
The arithmetic unit performs 16-bit addition, subtraction, absolute value and average value computations in one clock cycle. The first operand comes either from DA or is immediate data from the instruction word. The second operand (if there is one) is always from DB. Addition and subtraction is done with or without saturation depending on the saturation mode control bit.

3.2.2 Shift Unit
The shift unit performs 16-bit logic and arithmetic shift operations and rotation, with or without intermediate carry, in one clock cycle. All operations are specified as left shift operations. Right shift is accomplished by specifying a negative number of steps. The value to be shifted is always provided on DB. The number of steps is either given by the five least significant bits of DA or by a 5-bit immediate data value in the instruction word.

3.2.3 Logic Unit
The logic unit performs bitwise 'and', 'or', 'xor' and 'not' operations between 16-bit words. For 'and', 'or', and 'xor' operations the first operand is either on DA or immediate data from the instruction word and the second operand is on DB. The single operand for the 'not' operation is always on DA.

3.2.4 MAC Unit
The MAC unit performs multiplication with or without rounding, and multiply and accumulate operations in two clock cycles. It also performs 40-bit shift, 32-bit addition/subtraction and round operations in one clock cycle. All operations can be executed with or without saturation. The multiplication uses integer or fractional number representation depending on the fractional mode control bit. The MAC unit consists of the following parts:
A 32-bit multiplier multiplying two 16-bit operands from DA and DB into a 32-bit result. Both operands can be taken as signed or unsigned values independently. The result of the multiplication is stored in an internal pipeline register in the MAC unit.
A 40-bit adder where the registered result of the multiplication, sign extended with eight guard bits to a total of 40 bits, can be added to or subtracted from one of the 40-bit accumulator registers. A 16- or 32-bit value from DA, or DA concatenated with DB, can also
be added or subtracted directly to an accumulator register. The adder also facilitates rounding (see below).

A **40-bit barrel shifter** which enables the value from the accumulator to be arithmetically shifted before reaching the adder. The number of steps to shift is either the six least significant bits of DA or 6-bit immediate data from the instruction word (positive value for left shift and negative for right shift).

### 3.2.4.1 Rounding
Rounding is executed by adding 1 to the 17:th bit position (i.e. bit 16) of the 40-bit value, if the 16 least significant bits are larger than h7FFF. This means that the 24 most significant bits (16 bits plus 8 guard bits) of the result is the rounded value and the 16 least significant bits are set to zero. (The equivalent operation using decimal numbers would be to add one if the decimal part was greater than or equal to 0.5 and then truncate the decimals).

### 3.2.4.2 Saturation Unit
All MAC unit operations can be performed with or without saturation. If saturation is enabled, the result will be saturated to the smallest or largest possible 32-bit values (hFF80000000 and h007FFFFFFF respectively), whenever the result from the adder is smaller or larger than these values.

### 3.2.5 Count Unit
The count unit counts the leading ones or zeros in a 16-bit word in one clock cycle. The result is written to the register file via the RB-bus.

### 3.2.6 Register File
As mentioned before, the processor has a register file consisting of 30 16-bit registers and two 24-bit registers. All registers are listed in figure 3.3. All registers can be used as general purpose registers for holding operands and results for computational operations.

GPR0 - GPR7 (AP0 - AP7) can also be used as address registers for addressing data and tap memory.

GPR8 (STEP0) holds the step length for updating the address register AP0 during post increment addressing.

<table>
<thead>
<tr>
<th>GPR0/AP0</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR1/AP1</td>
</tr>
<tr>
<td>GPR2/AP2</td>
</tr>
<tr>
<td>GPR3/AP3</td>
</tr>
<tr>
<td>GPR4/AP4</td>
</tr>
<tr>
<td>GPR5/AP5</td>
</tr>
<tr>
<td>GPR6/AP6</td>
</tr>
<tr>
<td>GPR7/AP7</td>
</tr>
<tr>
<td>GPR8/STEP0</td>
</tr>
<tr>
<td>GPR9/STEP1-4</td>
</tr>
<tr>
<td>GPR10/TOP0</td>
</tr>
<tr>
<td>GPR11/BOTTOM0</td>
</tr>
<tr>
<td>GPR12/TOP1</td>
</tr>
<tr>
<td>GPR13/BOTTOM1</td>
</tr>
<tr>
<td>GPR14/LOOP</td>
</tr>
<tr>
<td>GPR15</td>
</tr>
</tbody>
</table>
GPR9 This register contains the step sizes for address registers AP1 – AP4. The register is divided into 4 times 4 bits, which means that each step size can maximally be 16 during post increment addressing.

GPR10 - GPR 13 Also holds top (TOP0/TOP1) and bottom (BOTTOM0/BOTTOM1) registers for modulo addressing.

GPR14 (LOOP) is also used to hold the loop counter value during hardware loops.

GPR28 - GPR31 (ACC0/ACC1) holds the accumulator registers, described in section 3.2.7. The architecture of the register file can be seen in figure 3.3.

3.2.7 The Accumulator Registers
The Register file has two 40-bit accumulator registers, ACC0 and ACC1, for storing results of MAC unit operations. ACC0 consists of GPR28 and GPR29. GPR28 holds the 16 least significant bits and GPR29 holds the 24 most significant bits, where the 8 most significant bits are guard bits. ACC1 consists of GPR30 and GPR31 and works in the same way. When the 24-bit registers GPR29 and GPR31 are used for other purposes, the 8 guard bits are sign extended.

The data on the 40-bit bus to the MAC is always either ACC0 or ACC1 and the data on the 40-bit bus from the MAC can only be written to either ACC0 or ACC1.

3.2.8 Addressing
The processor should be able to address up to 64 kWord program memory (PM), 64 kWord tap memory (TM), 64 kWord data memory (DM).

3.2.8.1 Addressing Modes
Here follows descriptions of all addressing modes.

**Register Direct**
The data is taken from a register, GPR0-GPR32, pointed out by the instruction.

**Example:** add GPR5 GPR6
**Register Indirect With Post Increment**
The data is taken from a memory address pointed out by an address register, AP0-AP8, given in the instruction. The address register is then updated by adding the value in the corresponding STEP register.

*Example:* `loadm AP1++ GPR5`

**Immediate Address**
The data is taken from a memory address from the instruction.

*Example:* `loadmi #hFF03 GPR4`

**Immediate Data**
The data is taken from the instruction.

*Example:* `addi #34 GPR5`

**Register Indirect With Offset**
The data is taken from an address given by an address register plus an offset from the instruction.

*Example:* `loadm AP0 #4 GPR3`

*Note:* Register indirect addressing without post increment can be achieved either by using offset addressing with zero offset or by using Register indirect addressing with post increment with the step length set to zero.

### 3.2.8.2 Modulo Addressing
Address registers AP0 and AP1 supports modulo addressing when the flags M0/M1 are enabled in the control register. The top and bottom registers TOP0/TOP1 and BOTTOM0/BOTTOM1 are implied in the operation. Modulo addressing mode affects both post increment and offset addressing.

### 3.2.8.3 Bit-Reversed Addressing
If the flag BR is set the address from address register AP0 is bit-reversed (regardless of which of the addressing methods in 3.2.8 is used), that is the least significant bit of the register becomes the most significant bit of the address etc.

*Example: For executing a 32 point FFT, first AP0 could be loaded with b0000000000000000 and STEP0 with b0000100000000000. Then when memory is accessed using post increment addressing the sequence of addresses used would be:

```
b0000000000000000 = 0
b0000000000010000 = 16
b0000000000001000 = 8
b0000000000011000 = 24
b0000000000000100 = 4
```

3.3 Control Path

The control path of the processor consists of the following blocks:
The Program Memory (PM) contains the program.
The Instruction decoder (ID) decodes the control signals from the instruction word.
The Program Counter (PC) Generates the address for the program memory.
The **Program flow controller (PFC)** controls the program counter. It also contains the PC stack, the loop stack and the repeat register. The **Pipeline Controller** monitors the pipeline and halts the execution when a conflict appears. The **Branch controller** keeps track of jump conditions.

### 3.3.1 The Pipeline

The processor uses a variable length pipeline with three or four steps. Instructions that incorporate a multiplication operation (‘mult’, 'mac') are executed in four steps and all other instructions in three steps. The reason for this is that the MAC unit is pipelined in two steps. The first pipeline step is the *instruction fetch*. Here the instruction is fetched from program memory and loaded into the instruction register. In the second step, *instruction decoding*, the control signals are decoded from the instruction word by the instruction decoder and stored in control registers. The third step differs slightly between three- and four step instructions. For three step instructions operands are fetched, the operation is performed and the result is written back to the register file. For four step instructions operands are fetched, multiplication is executed and the product is written to the pipeline register in the MAC unit. In the fourth step of a four step instruction the value of an accumulator register is fetched (if it is a mac instruction) the 40-bit addition/subtraction is executed and the result is written back to the accumulator register. The variable pipeline depth causes problems in some cases. These are handled by the pipeline controller which is described in 3.3.5.

### 3.3.2 Instruction Decoder

The instruction decoder decodes all control signals to the data path and some to the control path. It also manages the variable pipeline depth. For this purpose the control signals are stored in three registers:

- The *ctrl* register stores all control signals that are always only used in the third pipeline step.
- The *ctrl2* register stores signals that can be used either in step three or step four.
- The *pipe4* register is used *during the third step to store control signals that will be used in the fourth step*.

A special control signal in the *ctrl* register controls whether the signals in *ctrl2* are for the third or fourth step.

### 3.3.3 Program Counter

The program counter produces the address for the program memory. During normal execution, the PC is increased by one every clock cycle, but due to program flow instructions and pipeline complications this can change. The next address can be loaded either from the register file or from an immediate address in the instruction. It can also keep its old value to perform the ‘repeat’ instruction or when the pipeline has to be halted and it can be loaded with the loop start value from loop stack. Finally it can be loaded with the top value of the PC stack in case of a subroutine return and is set to zero on reset. The program counter is 16 bits wide, which means 64 kWords of program memory can be addressed.

### 3.3.4 Program Flow Controller

This block controls the updating of the PC. It also monitors and controls hardware looping (loop stack and repeat register) and subroutine calls (the PC stack). The program flow controller is the most complicated and tricky part of the whole processor and the design details will not be presented in this thesis. However the information below, regarding the PC stack and hardware looping, should be enough for understanding how to program the processor.
3.3.4.1 Subroutine Calls - The PC Stack
When a subroutine call is made (the ‘call’ instruction is executed) the program counter is loaded with the starting address of the subroutine and the old program counter address is pushed to the PC stack. When the program returns from the subroutine (the ‘rts’ instruction is executed) the PC stack is popped and the top value is loaded to the program counter. The PC stack depth is four, so up to four nested subroutine calls are possible.

3.3.4.2 Hardware Looping
The processor has two instructions for zero overhead hardware looping: The simple ’repeat’ instruction that repeats the following instruction a number of times and the more complex ’loop’ instruction that repeats any number of instructions larger than one.

Repeat
The ’repeat’ instruction is facilitated by the repeat register. During normal execution this register holds the value one, every instruction is executed once and the program counter is increased by one every clock cycle. However as soon as the value of the repeat register is not one, the PC, the instruction register and the control registers are no longer updated. Instead the repeat register is decreased by one every clock cycle until its value is one again. The ’repeat’ instruction simply loads a value (larger than one) into the repeat register thereby causing the next instruction to be repeated the specified number of times.

Loop
The ’loop’ instruction works in a completely different way than the ’repeat’ instruction. Before the ’loop’ instruction is executed the number of repetitions must be loaded into the loop register (GPR14). The code section to be looped starts with the instruction following the ’loop’ instruction and ends at an address specified in the instruction. When the ’loop’ instruction is executed, these two program addresses and the value of the loop register are pushed to the loop stack. When the PC reaches the address equal to the loop end address on the top of the loop stack, the PC is set to the corresponding loop start address and the corresponding loop counter value is decreased by one. The loop counter value is then copied back to the loop register in the register file. In that way the current loop counter value is accessible from the program. If the loop counter value is one when the PC reaches the loop end, the loop stack is popped. Since the loop stack depth is four, up to four nested loops are possible.

3.3.5 Pipeline Controller
The pipeline controller monitors the pipeline. By looking at what type of instruction is currently executing (if it is a three or four step instruction and which accumulator register it uses) and which is the next instruction to be executed, it determines if the pipeline has to be halted for one clock cycle, before the next instruction is executed. Halting the pipeline means in practice that the value of PC and instruction registers are kept and control signals of a ’nop’ operation is loaded to the control registers. Halting the pipeline is necessary if a four step instruction is followed by a three step instruction and one of the following is true: 1. The source register of the three step instruction is in the accumulator register used by the four step instruction. 2. The three step instruction is dependent on status flags generated by the four step instruction. 3. The three step instruction uses the MAC unit (but not the multiplier, because then it would not be a three step instruction).
3.3.6 Branch Controller
The branch controller is a very small block whose only purpose is to keep track of status flags and branch conditions and tell the PFC when to branch.
This chapter describes the control part of the memory and its contents. The control part resides in the 64 last positions of the tap memory. We refer to it below as control memory.

4.1 Overview
The first ten addresses in the control memory are used for Direct Memory Access (DMA). The following eighteen addresses are reserved for interrupt handling. The next two addresses are used for status and control of the data flow, described below. The last 32 addresses contain the memory mapped register file.

4.2 Status address
The status address has the following status flags:
N - Negative Value
Z - Equal zero Value
C - Carry/borrow bit
O - Overflow has occurred
The status flags are used to generate conditions for branch instructions. The following branch conditions are available:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than</td>
<td>N=0 and Z=0</td>
</tr>
<tr>
<td>Greater or equal</td>
<td>N=0</td>
</tr>
<tr>
<td>Equal</td>
<td>Z=1</td>
</tr>
<tr>
<td>Less or equal</td>
<td>N=1 or Z=1</td>
</tr>
<tr>
<td>Less than</td>
<td>N=1</td>
</tr>
<tr>
<td>Not equal</td>
<td>Z=0</td>
</tr>
<tr>
<td>Carry</td>
<td>C=1</td>
</tr>
<tr>
<td>Not carry</td>
<td>C=0</td>
</tr>
<tr>
<td>Overflow</td>
<td>O=1</td>
</tr>
<tr>
<td>Not overflow</td>
<td>O=0</td>
</tr>
</tbody>
</table>

See appendix A for information on which instructions affects which flags. Note however especially that the borrow bit is inverted, that is C is set when unsigned addition generates carry and when unsigned subtraction does not generate borrow.

4.3 Control address
The control address holds the following control bits:

| P_dma2 | P_dma1 | P_hp | P_cpu | Dma2 | Dma1 | GIE | BR | M1 | M0 | S | F |

**P_dma2, P_dma1, P_hp and P_cpu bits**
These bits states if the respective units have high or low priority when memory access conflicts occur.
**Dma2 and Dma1 bits**
The Dma bits states if the DMA-channels are active or not.

**The gie (global interrupt enabled) bit**
When this bit is set interrupts can be used.

**The BR (Bit Reverse) Control Bit**
When the BR control bit is set the address from AP0 is bit-reversed. See 3.2.8.3 for more information on bit-reversed addressing.

**The MO and M1 (Modulo Addressing) Control Bits**
M0 and M1 enables modulo addressing for the address registers AP0 and AP1 respectively. See 3.2.8.5 for more information on modulo addressing.

**The S (Saturation Mode) Control Bit**
This control bit enables saturation mode for the arithmetic and shift units. Note that saturation in the MAC unit is *not* affected by this control bit.

**The F (Fractional Mode) Control Bit**
When this control bit is enabled data words represent fractional numbers in the range [-1, 1] instead of integers. Only multiplication operations are affected by this.
In this chapter we describe the instruction set and how it can be utilized.

### 5.1 The instruction word

The processor uses a 32-bit instruction set, with one redundant bit. The instruction word is divided into subfields, described below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mux</td>
<td>31:27</td>
<td>Multiplexor switching between different instruction groups.</td>
</tr>
<tr>
<td>Op</td>
<td>26:22</td>
<td>Operation code choosing the actual operation.</td>
</tr>
<tr>
<td>M/C</td>
<td>21</td>
<td>1 for conditional execution, 0 for memory operations</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>1 for memory write operations, 0 for read.</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>1 when DM is used /*</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>1 when TM is used /*</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>not used /*</td>
</tr>
<tr>
<td>Ad1</td>
<td>16:14</td>
<td>Address register.</td>
</tr>
<tr>
<td>Sreg</td>
<td>13:9</td>
<td>Source register.</td>
</tr>
<tr>
<td>Ad2</td>
<td>8:6</td>
<td>Address register.</td>
</tr>
<tr>
<td>Dreg</td>
<td>5:1</td>
<td>Destination register.</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>Accumulator register, 0 for ACC0, 1 for ACC1.</td>
</tr>
</tbody>
</table>
Address/
Constant  21:6  Immediate address or constant value.
Offset     8:1  Immediate offset address value.
Prog Addr  16:1  Immediate program address.

* Used for condition, see appendix A.

For some operations the register fields have other meanings than described above. They can
be source registers instead of destination. For more information see the instruction encoding
chapter below.

5.2 Parallel Memory Instructions
Under certain conditions memory operations can be performed in parallel with other
instructions. The different possibilities are described below. In all cases Sreg and Dreg are
used both as source and destination for the two operations. When conditional execution is
used, parallel memory instructions can’t be used.
In assembly code, parallel instructions are separated by a “,” with space on both sides.

5.2.1 Move to memory
Two moves from registers to memory, both using register indirect addressing with post
increment, can be executed in parallel. The first move must use tm and the second one must
use dm.
Example: move2tm APx++ GPRx, move2dm APy++ GPRy

5.2.2 MAC operation and load
Up to two load instructions, using register indirect addressing with post increment, can be
executed in parallel with any MAC unit operation (mpy[u][s], mac[sub][u][s], rnd, sat, clracc,
addacr, subacr, add32 or sub32). If one load is parallel with a MAC operation, the destination
register for the load must be the same as the first operand register for the MAC operation, if it
has any operand registers (for example ‘rnd’ has no operands so any register can be loaded). If
two loads are parallel with a MAC operation the first load must use tm and its destination
must be the same as the first operand of the MAC operation. The second load must use dm and
its destination must be the same as the second operand register of the MAC operation (if it has
two operands). The destination for the second load must also always be one of GPR16 -
GPR31. It is also possible to execute the two load operations in parallel without the MAC
operation. In that case any register can be used.
Example 1: mac GPRx GPRy ACCz , loadtm APx++ GPRx , loaddm APy++ GPRy (where GPRy must be one of GPR16 - GPR31)
Example 2: loadtm APx++ GPRx , loaddm APy++ GPRy (GPRy can be
any register)

5.2.3 Arithmetic, logic, shift or move operation and load
One load instruction, using register indirect addressing with post increment, can be executed
in parallel with any arithmetic, logic or shift operation using registered operands (or in other
words: all arithmetic, logic and shift operations that do not use immediate data) or with the
‘move’ instruction. The destination for the load must be the same as the first/source operand
of the other operation.
Example: add GPRx GPRy , loaddm APx++ GPRx
5.3 Instruction set restrictions
Because of the pipeline there are some restrictions on how to use the instructions, these are described below.

5.3.1 Branch and jump instructions
This processor uses delayed jump, branch, subroutine call and subroutine return instructions. In other words the two instructions following the jump/branch etc is always executed whether the jump is taken or not. If one of these two instructions was also a jump instruction, or for example a loop instruction, that would cause complications, therefore ‘jmp’, ‘bra’, ‘call’ and ‘rts’ instructions must always be followed by two instructions that are not program flow instructions.

5.3.2 Hardware loops
The ’repeat’ Instruction
The only restriction that applies to the ’repeat’ instruction is that the repeated instruction may not be a program flow instruction.

The ’loop’ Instruction
The more complex ’loop’ instruction has the following restrictions:
1. The loop must consist of at least two instructions (otherwise use ’repeat’).
2. The two last instructions of the loop must not be program flow instructions.
3. Two nested loops may not end at the same address.
4. No more than four nested loops are allowed.

5.3.3 Modulo addressing
The implementation of modulo addressing does not allow the address to “wrap around” the modulo addressing area more than once. This results in the following restrictions:
1. When using modulo addressing and post increment, the step size should not be larger than the modulo addressing area (that is \( \text{STEPx} < \text{BOTTOMx} – \text{TOPx} \) should hold).
2. When using modulo addressing with offset addressing the offset should not be larger than the modulo addressing area.
### 5.4 Instruction encoding

A = Address register  
C = Constant data, address or offset  
D = Destination Register  
E = Program address register  
G= Condition / Memory R/W  
H= Condition/ Memory  
k = Condition  
M =Memory use  
r = Round accumulator  
S = Source Register  
s = Saturate accumulator  
P= Condition / Memory usage  
V= Condition / -  
Y = Accumulator register  
- = Don’t care

#### Mux OpCode Memory Addr1 Sreg Addr2 Dreg Instruction

**Data move instructions**

<table>
<thead>
<tr>
<th>Mux</th>
<th>OpCode</th>
<th>Memory</th>
<th>Addr1</th>
<th>Sreg</th>
<th>Addr2</th>
<th>Dreg</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00000</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>nop</td>
</tr>
<tr>
<td>00000</td>
<td>00001</td>
<td>00MM-</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>DDDDD</td>
<td>load{tm/dm} AP++ GPRy</td>
</tr>
<tr>
<td>00000</td>
<td>00010</td>
<td>00MM-</td>
<td>AAA</td>
<td>DDDDD</td>
<td>CCC</td>
<td>CCCCC</td>
<td>load{tm/dm} AP #offset GPRy</td>
</tr>
<tr>
<td>00000</td>
<td>00100</td>
<td>00MM-</td>
<td>AAA</td>
<td>DDDDD</td>
<td>CCC</td>
<td>CCCCC</td>
<td>move2{tm/dm} AP #offset GPRy</td>
</tr>
<tr>
<td>00000</td>
<td>01011</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>loadaddi #addr GPRy</td>
</tr>
<tr>
<td>00000</td>
<td>01111</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>movel #data GPRy</td>
</tr>
<tr>
<td>00000</td>
<td>01100</td>
<td>00MM-</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>load {tm/dm} GPRx</td>
</tr>
<tr>
<td>00000</td>
<td>100CC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CAAAA</td>
<td>movetmd #data #addr</td>
</tr>
</tbody>
</table>

**ALU instructions, arithmetic unit**

<table>
<thead>
<tr>
<th>Mux</th>
<th>OpCode</th>
<th>Memory</th>
<th>Addr1</th>
<th>Sreg</th>
<th>Addr2</th>
<th>Dreg</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>00000</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>abs GPRx GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>0001c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>add GPRx GPRy [C]</td>
</tr>
<tr>
<td>00010</td>
<td>00001</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>addi #data GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>00100</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>SSSSS</td>
<td>comp GPRx GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>00101</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>neg GPRx GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>0100c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>sub GPRx GPRy [C]</td>
</tr>
<tr>
<td>00010</td>
<td>01111</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>subj #data GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>01010</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>move GPRx GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>01011</td>
<td>HVVVV</td>
<td>SSSS-</td>
<td>SSSS-</td>
<td>---</td>
<td>DDDDD</td>
<td>move32 GPRx GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>01110</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>load #data GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>0110c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>DDD</td>
<td>SSSSS</td>
<td>add3 GPRx GPRy GPRz</td>
</tr>
<tr>
<td>00010</td>
<td>1000c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>DDD</td>
<td>SSSSS</td>
<td>sub3 GPRx GPRy GPRz</td>
</tr>
<tr>
<td>00010</td>
<td>01111</td>
<td>00MM-</td>
<td>AAA</td>
<td>SSSSS</td>
<td>DDD</td>
<td>SDDDD</td>
<td>max GPRx APx++ GPRy APy</td>
</tr>
<tr>
<td>00010</td>
<td>10010</td>
<td>00MM-</td>
<td>AAA</td>
<td>SSSSS</td>
<td>DDD</td>
<td>SDDDD</td>
<td>min GPRx APx++ GPRy APy</td>
</tr>
<tr>
<td>00010</td>
<td>11000</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>count GPRx GPRy</td>
</tr>
<tr>
<td>00010</td>
<td>11001</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>DDD</td>
<td>YSSSS</td>
<td>butterfly GPRx GPRy GPRz ACCx</td>
</tr>
</tbody>
</table>
## ALU instructions, logic unit

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000000</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>and GPRx GPRy</td>
</tr>
<tr>
<td>001000001</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>andi #data GPRy</td>
</tr>
<tr>
<td>001000010</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>or GPRx GPRy</td>
</tr>
<tr>
<td>001000011</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>ori #data GPRy</td>
</tr>
<tr>
<td>001000100</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>xor GPRx GPRy</td>
</tr>
<tr>
<td>001000101</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>xori #data GPRy</td>
</tr>
<tr>
<td>001000110</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>not GPRx GPRy</td>
</tr>
<tr>
<td>001001000</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>SSSSS</td>
<td>bittest #data GPRx</td>
</tr>
<tr>
<td>001001001</td>
<td>HVVVV</td>
<td>SSS</td>
<td>SSSSS</td>
<td>SSS</td>
<td>DDDDD</td>
<td>setfield GPRx GPRy GPRz GPRw</td>
</tr>
<tr>
<td>001001011</td>
<td>HVVVV</td>
<td>SSS</td>
<td>SSSSS</td>
<td>SSS</td>
<td>DDDDD</td>
<td>getfield GPRx GPRy GPRz GPRw</td>
</tr>
<tr>
<td>001010000</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>xor32 GPRx GPRy</td>
</tr>
<tr>
<td>001010001</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>xori32 #data GPRy</td>
</tr>
<tr>
<td>001010010</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>not32 GPRx GPRy</td>
</tr>
<tr>
<td>001010011</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>xori32 #data GPRy</td>
</tr>
</tbody>
</table>

## ALU instructions, shift unit

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110000c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>asl GPRx GPRy</td>
</tr>
<tr>
<td>00110001c</td>
<td>HVVVV</td>
<td>---</td>
<td>---CC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>asli #data GPRy</td>
</tr>
<tr>
<td>00110010c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>lsl GPRx GPRy</td>
</tr>
<tr>
<td>00110100c</td>
<td>HVVVV</td>
<td>---</td>
<td>---CC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>lsli #data GPRy</td>
</tr>
<tr>
<td>00111000c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>---</td>
<td>DDDDD</td>
<td>rsl GPRx GPRy</td>
</tr>
<tr>
<td>00111100c</td>
<td>HVVVV</td>
<td>---</td>
<td>---CC</td>
<td>CCC</td>
<td>DDDDD</td>
<td>rsli #data GPRy</td>
</tr>
</tbody>
</table>

## MAC instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>010s00000</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>mpy GPRx GPRy ACCx [SAT/RND]</td>
</tr>
<tr>
<td>010s00001</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>mpyu GPRx GPRy ACCx [SAT/RND]</td>
</tr>
<tr>
<td>010s00010</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>mpyus GPRx GPRy ACCx [SAT/RND]</td>
</tr>
<tr>
<td>010s00011</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>mpyus GPRx GPRy ACCx [SAT/RND]</td>
</tr>
<tr>
<td>010s00100</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>mac GPRx GPRy ACCx [SAT]</td>
</tr>
<tr>
<td>010s00101</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>macu GPRx GPRy ACCx [SAT]</td>
</tr>
<tr>
<td>010s00110</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>macus GPRx GPRy ACCx [SAT]</td>
</tr>
<tr>
<td>010s00111</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>macsub GPRx GPRy ACCx [SAT]</td>
</tr>
<tr>
<td>010s0011L</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>YSSSS</td>
<td>macsubu GPRx GPRy ACCx [SAT]</td>
</tr>
<tr>
<td>010s10000</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>rnd ACCx [SAT]</td>
</tr>
<tr>
<td>010s10001</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>sat ACCx</td>
</tr>
<tr>
<td>010s00111</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>clacc ACCx</td>
</tr>
<tr>
<td>010s10111</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>addacc GPRx ACCx [SAT]</td>
</tr>
<tr>
<td>010s11000</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>subacc GPRx ACCx [SAT]</td>
</tr>
<tr>
<td>010s11001</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>add32 GPRx ACCx [SAT]</td>
</tr>
<tr>
<td>010s11010</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>sub32 GPRx ACCx [SAT]</td>
</tr>
<tr>
<td>010s11011</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>addadd GPRx GPRy ACCx [SAT]</td>
</tr>
<tr>
<td>010s11100</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>subsub GPRx GPRy ACCx [SAT]</td>
</tr>
<tr>
<td>010s11111</td>
<td>HGPPV</td>
<td>AAA</td>
<td>DDDDD</td>
<td>AAA</td>
<td>YDDDD</td>
<td>loadacc GPRx ACCx</td>
</tr>
<tr>
<td>010s1111c</td>
<td>HGPPV</td>
<td>AAA</td>
<td>SSSSS</td>
<td>AAA</td>
<td>Y---</td>
<td>lshli GPRx ACCx [SAT]</td>
</tr>
</tbody>
</table>

## Program flow instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>011000000</td>
<td>kkkk-</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>EEEEE</td>
<td>bra{cond} GPRx</td>
</tr>
<tr>
<td>011000001</td>
<td>kkkk-</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>bra{cond} #addr</td>
</tr>
<tr>
<td>011000010</td>
<td>kkkk-</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>EEEEE</td>
<td>bra{cond}r GPRx</td>
</tr>
<tr>
<td>011000011</td>
<td>kkkk-</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>bra{cond}r #addr</td>
</tr>
<tr>
<td>011000100</td>
<td>kkkk-</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>EEEEE</td>
<td>jmp GPRx</td>
</tr>
<tr>
<td>011000101</td>
<td>kkkk-</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>jmp #addr</td>
</tr>
<tr>
<td>011000110</td>
<td>kkkk-</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>jmp #addr</td>
</tr>
<tr>
<td>011000111</td>
<td>kkkk-</td>
<td>CCC</td>
<td>CCCCC</td>
<td>CCC</td>
<td>CCCCC</td>
<td>call GPRx</td>
</tr>
<tr>
<td>Opcode</td>
<td>Function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 01001</td>
<td>CCC CCCCC CCC CCCCC call #addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 01010</td>
<td>--- --- --- EEEEE callr GPRx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 01011</td>
<td>CCC CCCCC CCC CCCCC callr #addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 01100</td>
<td>--- --- --- EEEEE loop GPRx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 01101</td>
<td>CCC CCCCC CCC CCCCC loop #addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 01110</td>
<td>--- --- --- EEEEE loopr GPRx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 01111</td>
<td>CCC CCCCC CCC CCCCC loopr #data repeat GPRx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 10000</td>
<td>--- --- --- CCC CCCCC repeat #data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 10001</td>
<td>--- --- --- --- ----- rts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100 10010</td>
<td>--- --- --- --- ----I rti</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction format**

| xxxx | xxxx | xxxx | xxx | xxxx | xxx | xxxx |
Chapter 6  Benchmarking and profiling

6.1 Introduction

Benchmarking by definition is some kind of measure of performance, in this case of a DSP – processor. This can be measured in many ways, but one common metric is the time required to accomplish a defined task. Other metrics can though be at least as important, for instance memory usage or power consumption.

One purpose of benchmarking could be to compare performance between different processors. Another purpose of benchmarking is to check and improve the performance of the assembly instruction set.

Benchmarking can further be divided into algorithm kernel benchmarking and application level benchmarking. The algorithm kernel benchmark is used to test basic algorithm based DSP subroutines to check the general performance when executing any or most DSP applications. Examples of kernel algorithms are FIR, IIR, FFT and others on this level. An application level benchmark is used to check the performance for a specific application, e.g. MPEG4 for image compression.

Measuring processor performance in a way that allows fair comparisons between processor families is difficult. Furthermore, performance measurements are only useful to the engineer if the measurements can be related to the requirements of particular applications.

Traditional metrics of measuring performance are often simple. One is the MIPS-metric, millions of instructions per second. This is not always a good metric though, an instruction on one processor may accomplish far more work than an instruction on another processor. This is especially true for signal processors, which often have highly specialised instructions sets. Also the MOPS-metric, millions of operations per second, suffers from a similar problem.

The purpose of this thesis is primarily to evaluate the instruction set of the processor at hand. We have used a cycle true ISS, instruction set simulator, to measure the number of required instruction cycles to execute certain algorithm kernels. The algorithms have been hand-optimised and written directly in assembly code and then run in the ISS. Figure 6.1 below shows an overview of the algorithm kernels and the results.
<table>
<thead>
<tr>
<th>Algorithm Kernels</th>
<th>Number of samples</th>
<th>Taps</th>
<th>Cycle cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocktransfer</td>
<td>40</td>
<td>-----</td>
<td>88</td>
</tr>
<tr>
<td>256 point complex FFT</td>
<td>256</td>
<td>-----</td>
<td>18763</td>
</tr>
<tr>
<td>Single FIR</td>
<td>1</td>
<td>16</td>
<td>60</td>
</tr>
<tr>
<td>Frame FIR</td>
<td>40</td>
<td>16</td>
<td>921</td>
</tr>
<tr>
<td>Complex FIR</td>
<td>40</td>
<td>16</td>
<td>3696</td>
</tr>
<tr>
<td>IIR DF I</td>
<td>40</td>
<td>40</td>
<td>2450</td>
</tr>
<tr>
<td>IIR DF II</td>
<td>40</td>
<td>40</td>
<td>3500</td>
</tr>
<tr>
<td>LMS Adaptive FIR</td>
<td>40</td>
<td>16</td>
<td>4384</td>
</tr>
<tr>
<td>Wave digital filter</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>16-bit division</td>
<td>-----</td>
<td>-----</td>
<td>67</td>
</tr>
<tr>
<td>Vector add</td>
<td>40</td>
<td>-----</td>
<td>131</td>
</tr>
<tr>
<td>Vector dot</td>
<td>40</td>
<td>-----</td>
<td>53</td>
</tr>
<tr>
<td>Vector Max</td>
<td>40</td>
<td>-----</td>
<td>55</td>
</tr>
<tr>
<td>Floating to fixed</td>
<td>1</td>
<td>-----</td>
<td>17</td>
</tr>
<tr>
<td>Fixed to floating</td>
<td>1</td>
<td>-----</td>
<td>58</td>
</tr>
<tr>
<td>FSM</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>DCT</td>
<td>64</td>
<td>-----</td>
<td>874</td>
</tr>
<tr>
<td>Cross Correlation</td>
<td>40</td>
<td>-----</td>
<td>753</td>
</tr>
<tr>
<td>Auto Correlation</td>
<td>1</td>
<td>-----</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 6.1, Number of samples is the size of the input data frame. Taps is the number of coefficients. Cycle cost refers to the number of instruction cycles needed for executing the algorithm kernel.

The secondary purpose of this thesis was to analyse the assembly code, and propose changes to the instruction set in order to reduce the number of instruction cycles required to execute the algorithm kernels. Limits were set on additional hardware. No additional multiplier or adder was allowed. We have listed all of the algorithms we implemented with the modified instruction set in chapter 7. A behaviour level description is given to make it easier to understand the assembly code.

6.2 BDTI

Berkeley Design Technology, Inc is a company that has developed an independent set of benchmarks to provide an objective basis for comparing processor performance. BDTI continuously performs benchmarks on processors and present the results. They use application kernels to represent larger applications. This technique has several advantages:

- The application kernels can be selected from the most important and time consuming parts of the applications.
- They are easy to specify.
- They can be optimized in a reasonable amount of time.

The strategy also has some limitations:

- It is hard to capture all important factors of the performance.
- The numerical accuracy.
- The processor performance is very dependent on the system design.
BDTI refers to the benchmarks as program macros, parameterized blocks of code that is expanded in-line in the final program. BDTI divides each benchmark program into four sections:

- Power-up initialization, which is only required once.
- Benchmark function initialization, setup actions that must be executed each time the benchmark function is executed.
- Kernel, is the main computation section of the benchmark.
- Termination, wrap-up.

There are some guidelines to the benchmark programs in the BDTI specification.

- The benchmark functions are not implemented as subroutines or use subroutines internally.
- Each benchmark function should be written as if it was one of many functions in an application, which can be executed multiple of times.
- For processors that use caches, both results for empty and preloaded caches are presented.
- In most benchmarks the memory usage is not the primary goal and all memory accesses can be assumed to be on-chip.
- Excessive loop unrolling and table look-ups are not allowed.

Our ambition in this thesis has been to follow the BDTI requirements and specifications for kernel benchmarking, and hence to be able to do comparisons to the processors benchmarked by BDTI. Though, it’s important to emphasize that we haven’t had access to all specifications set by BDTI on the algorithms, and hence we can’t guarantee that our programs fulfill all requirements. Comparisons to BDTI will be treated later in this report.
In this chapter, the programs implementing the different algorithms are listed. Some background facts about the algorithm are presented, followed by the assembly code written with the new instruction set. In some cases, in connection to the assembly code, a behaviour level description is presented in order to make it easier to understand the assembly code and the implementation.

All of the programs begin with a setup phase. The purpose of the setup phase is to set step sizes and start addresses for the address pointers, load registers with start values and so on.

We also use a so called status buffer. When executing a DSP application, several of the below listed algorithms might be needed to form a certain application. One application can consist of an IIR filtering followed by an FFT for instance. One might after this want to continue the IIR filtering. In that case, it is important that the result is stored at another place in memory, so that the result of the first IIR filtering is not overwritten. Therefore, we store the out pointer value in the status buffer, so that the next IIR filtering starts writing its result in an adjacent place in memory. That is one example that shows the need of a status buffer.

When forming applications, the algorithms are stored in adjacent places in memory and the different algorithms that form the applications are called as subroutines.

### 7.1 Blocktransfer

Program that measures how many cycles it takes to transfer a block in memory to another position.

**Behavioural code:**

```plaintext
{ /* Get pointer values from status buffer */
    For I=0 to 39 do (
        Reg1 <= DM(DP1)
        DP1 <= DP1 + 1
        DM(DP2) <= Reg1
        DP2 <= DP2 + 1
    )
}
```

**Assembly code:**

```plaintext
*-------Initiation----------
lod #2000 GPR0
movemd+ #0 GPR0  *start of transfer source
movemd+ #100 GPR0 *start of transfer destination
movemd+ #40 GPR0  *number of samples in block

*-------status buffer--------
loaddmi #2000 AP5  *load source-pointer
```

31
loaddmi #2001 AP6  *load destination-pointer
loaddmi #2002 GPR14 *load loop-reg with number of samples

*-------- Start transfer-------
loopr #2
loaddmi AP5++ GPR0
move2dm AP6++ GPR0
7.2 FFT, Fast Fourier Transform

The formal definition for the Discrete Fourier Transform, DFT, is

\[ X(k) = \sum_{n=0}^{N-1} x(n) \cdot W^{nk}, \quad k = 0, 1, \ldots, N - 1 \]

In this form, it takes a great deal of time to compute the DFT and it is unpractical to implement. But with some mathematical manipulations the computing time can be radically reduced. Below we show the principle of how to derive FFT – algorithms. The basic idea is to divide the DFT into smaller sections and to use symmetry.

In the first step, we simply express the definition in another form [1].

\[ X(z) = X_0(z) + Z^{-N/2} \cdot X_1(z) \]

where \( X_0(z) = \sum_{n=0}^{(N/2)-1} x[n]z^{-n} \) , \( X_1(z) = \sum_{n=0}^{(N/2)-1} x[N/2 + n]z^{-n} \)

If we now evaluate \( X(z) \) on the unit circle at \( z = W_N^{-k} \), we get from the previous equation

\[ X[k] = \sum_{n=0}^{(N/2)-1} \left( x[n] + (-1)^k x[N/2 + n] \right) W_N^{nk} \]

where we have used that \( W_N^{(N/2)k} = (-1)^k \)

This equation represents the N/2-point DFT’s of the following sequences, depending on whether \( k \) is odd or even:

\[ x_0[n] = \left( x[n] + x[N/2 + n] \right) \quad \text{and} \quad x_1[n] = \left( x[n] - x[N/2 + n] \right) W_N^{n} \]

for \( n = 0, 1, \ldots, N - 1 \).

This separation can continue until we only have 2-point DFT’s left. A scheme of computation is shown in the figure below. One can understand the benefit from performing this separation into smaller DFT’s, due to the fact that an N-point DFT requires approximately \( 2N^2 \) arithmetic operations.
The dashed pattern in the figure above is called a butterfly because of its particular shape [3]. The butterfly is calculated in the following way and it is the real kernel of the FFT computation:

\[
\begin{align*}
\text{X} & \oplus \text{X+Y} \\
\text{Y} & \oplus (\text{X-Y})^* W
\end{align*}
\]

Figure 7.2, Sande-Tukey butterfly

The value of \( W_k \) is calculated from the formula \( e^{\frac{2\pi i k}{N}} \). The values are pre-calculated and stored in tap memory. The input samples are assumed to be 16 bit fractional values and stored in data memory. The program consists basically of three loops. In the inner loop, a butterfly is calculated and then multiplied with the corresponding value of \( W \). The middle loop is needed in order to keep track of how many continuous groups of butterflies there are in each column, and the last and final loop keeps track of which of the columns we are operating in.

Our implementation is a 256-point radix-2 in-place complex FFT, using the Sandy – Tukey algorithm. That means first, that the samples are complex-valued. By other words, there are 512 numbers. In memory, the real and imaginary parts are stored in adjacent places. Radix-2 means that the length must be a power of 2. There exist also for instance radix-4 algorithms, which are slightly more efficient. In-place means that no extra memory space is needed when computing the FFT. The values that are used as in-data to the butterfly will be replaced with the corresponding result in the same row. The assembly code has been written in a way so it can easily be modified to operate on a different number of samples than 256. Though notice, the number of samples must be a power of two.

To the right in figure 7.1 above one can see that the output from the last column in the computation is in an unscrambled order. To obtain order in this it is necessary to use something called bit-reversed addressing, which is supported by this processor. It works in the
following way: Once again, study the case with eight samples in figure 7.3. With binary representation order is obtained by just simply reading the binary word from the right, hence the name bit-reversed.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>100</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>110</td>
<td>011</td>
<td>011</td>
<td>011</td>
<td>011</td>
<td>011</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>001</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

*Figure 7.3, bit reversed addressing*

The behaviour level code given here describes a complex FFT operating on 256 samples. The code has been written so it can be easily modified according to the number of samples one wishes to apply the FFT operation on. Though, the number of samples must be a power of two. The behaviour level code for the kernel part is given as follows:

**Behaviour level code, kernel part:**

```c
{
    /* Calculate Butterfly */
    Reg1 <= DM(DP0) + DM(DP1)                /* Real part of sample */
    Reg2 <= DM(DP0) – DM(DP1)
    /* Get complex part of sample */
    Reg3 <= DM(DP0+1) + DM(DP1+1)
    Reg4 <= DM(DP0+1) – DM(DP1+1)
    /* Start multiplication with w */
    ACR0 <= 0 , ACR1 <= 0
    ACR0 <=  TM(TP0) * Reg2 – Reg4 * TM(TP0+1) /* Re */
    ACR1 <=  Reg4 * TM(TP0) + Reg2 * TM(TP0+1) /* IM */
    Round ACR Sat
    MEM(DP0) <= Reg1, DP0 <= DP0 + 1
    MEM(DP0) <= Reg3
    MEM(DP1) <= ACR0, DP1 <= DP1 + 1
    MEM(DP1) <= ACR1
}
```

**Assembly code:**

*256-point complex FFT radix-2.
*initialization

load #2001 GPR0
move2dmd+ #1 GPR0
move2dmd+ #2 GPR0
move2dmd+ #0 GPR0
move2dmd+ #256 GPR0  *** change according to number of samples
move2dmd+ #1 GPR0
move2dmd+ #128 GPR0 *** change according to number of samples (256 if 512-point fft and so on)
move2dmd+ #2 GPR0
move2dmd+ #2 GPR0
move2dmd+ #0 GPR0
move2dmd+ #8 GPR0 *** change according to number of samples, 8 because 256 point(2^8)
move2dmd+ #3000 GPR0 *** Address for result
move2dmd+ #128 GPR0 *** Step size for bit reversing
* status buffer starts at address 2000
move2tmd #1 CONTROL
loaddmi #2002 STEP0
loaddmi #2003 GPR0
loaddmi #2004 GPR6
loaddmi #2004 GPR3
loaddmi #2005 GPR15
loaddmi #2006 GPR25
loaddmi #2007 GPR4
loaddmi #2009 GPR5
loaddmi #2010 GPR14
*--------------FFT comp. starts here-----------
loopr #31
move GPR15 GPR14
loopr #22
move GPR25 GPR14
loopr #17
loaddm AP5 #0 GPR18
loaddm AP6 #0 GPR7
loadtm AP0 #1 GPR22
butterfly GPR18 GPR7 GPR23 ACC0, loadtm AP0++ GPR18
loaddm AP5 #1 GPR20
loaddm AP6 #1 GPR2
butterfly GPR20 GPR2 GPR21 ACC1
move2dm AP5++ GPR28
move2dm AP5++ GPR30
mpy GPR23 GPR18 ACC0
macsub GPR21 GPR22 ACC0
mpy GPR23 GPR22 ACC1
rnd ACC0 SAT
mac GPR21 GPR18 ACC1
move2dm AP6++ GPR29
rnd ACC1 SAT
move2dm AP6++ GPR31
load #0 GPR0
add GPR3 GPR5
add GPR3 GPR6
asli #-1 GPR25 *inner loop halved by factor 2.
asli #1 GPR15
asli #-1 GPR3
asli #1 GPR4
load #0 GPR5
move GPR3 GPR6
move GPR4 STEP0

*---------------------bitreversing---------------------
move2tmd #17 CONTROL
loaddmi #2009 GPR0
setstep #2 #2
loaddmi #2011 GPR2
loaddmi #2012 STEP0
loaddmi #2008 GPR14

loopr #5
loaddmi #2004 GPR14
loopr #2
    loaddrm AP0++ GPR19
    move2dm AP2++ GPR19
load #3001 GPR2
7.3 Digital filters

Digital filters are heavily used in DSP-based applications like, audio processing, speech compression and motor control. Digital filters have many advantages from analogue filters. For example can they be programmable, implemented at lower cost and they often have sharper transfer functions. Three types of digital filters are described below, FIR, IIR and adaptive FIR.

7.3.1 FIR filters, Finite-duration Impulse Response filters

A FIR filter is a weighted sum of a finite set of inputs.

\[ y(n) = \sum_{k=-d}^{n-1} a_k x(n - k) \]

- \( x(n) \) is the input
- \( y(n) \) is the output
- \( a_k \) is a vector of filter coefficients

Figure 7.4, Non recursive FIR filter

FIR filters have some especially good properties. They are always stable and they can be implemented with linear phase.

Implementation

We have implemented three different FIR filters. One real sample based and two frame-based. One frame has real input values, while the other one has complex.

The tap coefficients are stored in the tap memory, in reversed order. That means that the coefficient with the highest \( k \), is stored first. This is done because the FIR loop processes the oldest input first. The inputs are stored in a circular buffer, with the same length as the number of taps.

The program starts with initiation of the status buffer and address pointers. The kernel of the program starts with that the first tap and the input are collected. The input is then written to the computing buffer and the oldest value in the buffer is received. After that all the taps and
input samples are multiplied and accumulated. Then the result is written back to memory and the next input sample is processed.

<table>
<thead>
<tr>
<th>Computing buffer</th>
<th>Tap memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>k=0, newest sample</td>
<td>k=4</td>
</tr>
<tr>
<td>k=4, oldest sample</td>
<td>k=3</td>
</tr>
<tr>
<td>k=3</td>
<td>k=2</td>
</tr>
<tr>
<td>k=2</td>
<td>k=1</td>
</tr>
<tr>
<td>k=1</td>
<td>k=0</td>
</tr>
</tbody>
</table>

*Figure 7.5, Example of memory allocation for a five tap FIR.*

There is a small difference to the other two filters. In the complex FIR some more computations in the inner loop have to be performed, to handle the complex inputs and coefficients. The sample based only differs in the lack of the outer loop.

Below is a table of the kernel part of the program. The table shows which units are used and which instructions that can be reduced.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clr</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Move2dm</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>Loaddm</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Repeat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Mac</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>rnd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Move2dm</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

*Figure 7.6, Kernel operations of a FIR filter.*

### 7.3.1.1 Singlesample FIR

In this program, one sample a time is retrieved and stored in a computing buffer in DM. The computing buffer has the same size as the frame in TM containing the filter taps. This is for an efficient use of the pointers. Both computing buffer and the tap-frame are circular FIFO buffers.

**Behaviour level code:**

```{res
Reset ACR
DM(DP) <= Newest Sample
DP <= DP + 1
/*Store newest sample in computing buffer, and then load from the buffer the oldest sample, using same pointer. */
For i=0 to 15 do (  
    ACR =< ACR + DM(DP)*TM(TP) /* convolution for a sample */
    DP <= DP + 1, TP <= TP + 1 )
Round and Sat ACR
```
Store pointers.

Assembly code:

load #2000 GPR0
move2dmd+ #100 GPR0 *top CB
move2dmd+ #115 GPR0 *Bottom CB
move2dmd+ #100 GPR0 *Last CB data
move2dmd+ #3000 GPR0 *out
move2dmd+ #0 GPR0 *Frame start
move2dmd+ #0 GPR0 *top/start Tap
move2dmd+ #15 GPR0 *bottom tap

load #101 GPR0
repeat #15
move2dmd+ #0 GPR0 *clear CB

move2tmd #13 CONTROL *modular addressing + fractional
load addi #2000 TOP0 *top CB
load addi #2001 BOTTOM0 *bottom CB
load addi #2002 AP0 *last CB
load addi #2003 AP5 *out
load #1 STEP0 *step 0
setstep #1 #1 *step 1
load addi #2004 AP6 *frame start
load addi #2005 TOP1 *top T
load addi #2005 AP1 *last T
load addi #2006 BOTTOM1 *bottom T

clracc ACC0, loadtm AP1++ GPR21, loaddm AP6++ GPR20 *load tap + sample from frame
move2dm AP0++ GPR20 *store sample in CB
loaddm AP0++ GPR20 *load oldest sample in CB
repeat #15 *number of taps
mac GPR21 GPR20 ACC0, loadtm AP1++ GPR21, loaddm AP0++ GPR20 *x(n-k)ak

mac GPR21 GPR20 ACC0 *x(n-k)ak
rnd ACC0 SAT *round and sat
move2dm AP5++ GPR29 *store sample
move2dmi #2003 AP5 *store out pointer
move2dmi #2002 AP0 *store last CB pointer

........
7.3.1.2 Frame based FIR

Differs from the above in the way that sample arrives not by one by one, but instead in frames of size 40. An outer loop on top of the above behaviour code with 40 repetitions completes the code.

Assembly code:

```
load #2000 GPR0
move2dmd+ #100 GPR0 *top CB
move2dmd+ #115 GPR0 *Bottom CB
move2dmd+ #100 GPR0 *Last CB data
move2dmd+ #3000 GPR0 *out
move2dmd+ #0 GPR0 *Frame start
move2dmd+ #40 GPR0 *frame length
move2dmd+ #0 GPR0 *top/start Tap
move2dmd+ #15 GPR0 *bottom tap

load #101 GPR0
repeat #15
move2dmd+ #0 GPR0 *clear CB

move2tdm #13 CONTROL *modulo addressing+ fractional
loaddmi #2000 TOP0 *top CB
loaddmi #2001 BOTTOM0 *bottom CB
loaddmi #2002 AP0 *last CB
loaddmi #2003 AP5 *out
load #1 STEP0 *step 0
setstep #1 #1 *step 1
loaddmi #2004 AP6 *frame start
loaddmi #2006 TOP1 *top T
loaddmi #2006 AP1 *last T
loaddmi #2007 BOTTOM1 *bottom T

loaddmi #2005 LOOPREG *number of samples in frame
loopr #8
cracc ACC0, loadtm AP1++, GPR21, loaddm AP6++, GPR20 *load tap+ sample from frame
move2dm AP0++, GPR20 *store sample in CB
loaddm AP0++, GPR20 *load oldest sample in CB
repeat #15 *number of taps
mac GPR21 GPR20 ACC0, loadtm AP1++, GPR21, loaddm AP0++, GPR20
*acc<=acc+x(n-k)*ak
mac GPR21 GPR20 ACC0
rnd ACC1 SAT *round and sat
move2dm AP5++, GPR29 *store sample
end loop
move2dmi #2003 AP5 *store out pointer
move2dmi #2002 AP0 *store last CB pointer
```
7.3.1.3 Complex frame based FIR

Works as the above frame based FIR. A complex value is stored in memory with the real-part in position A, and Im-part in A + 1. Behavioural code is identical to the above frame based FIR.

Assembly code:

load #2000 GPR0
move2dmd+ #100 GPR0 *top CB
move2dmd+ #131 GPR0 *Bottom CB
move2dmd+ #100 GPR0 *Last CB data
move2dmd+ #3000 GPR0 *out
move2dmd+ #0 GPR0 *Frame start
move2dmd+ #40 GPR0 *frame length
move2dmd+ #0 GPR0 *top/start Tap
move2dmd+ #31 GPR0 *bottom tap

load #102 GPR0
repeat #30
move2dmd+ #0 GPR0 *clear CB

move2tmd #13 CONTROL *modular addressing+ fractional
loadaddmi #2000 TOP0 *top CB
loadaddmi #2001 BOTTOM0 *bottom CB
loadaddmi #2002 AP0 *last CB
loadaddmi #2003 AP5 *out
load #1 STEP0 *step 0
setstep #1 #1 *step 1
loadaddmi #2004 AP6 *frame start
loadaddmi #2006 TOP1 *top T
loadaddmi #2006 AP1 *last T
loadaddmi #2007 BOTTOM1 *bottom T
loadaddmi #2005 GPR14 *number of samples in frame
loopr #21 *start loop 1
clrac ACC0 , loadtm AP1++ GPR22 , loadaddm AP6++ GPR20 *load re tap+ sample from frame
clrac ACC1 , loadtm AP1++ GPR23 , loadaddm AP6++ GPR21 *load im tap+ sample from frame
move2dm AP0++ GPR20 *store re sample in CB
move2dm AP0++ GPR21 *store im of sample in CB
loadaddm AP0++ GPR20 *load oldest re sample in CB
loadaddm AP0++ GPR21 *load oldest im sample in CB
load #15 GPR14 *number of taps
loopr #5 *start loop 2 mult and acum RE-part in ACC0 IM-part in ACC1
mac GPR20 GPR22 ACC0 *RE[x(n-k)]*RE[ak]
macsub GPR21 GPR23 ACC0 *IM[x(n-k)]*IM[ak]
mac GPR20 GPR23 ACC1 , loadaddm AP0++ GPR20 *RE[x(n-k)]*IM[ak]
7.3.2 IIR filters, Infinite-duration Impulse Response filters

IIR filters can realize some transfer functions that the FIR filters can’t, because it has both poles and zeros in the complex plane. But this also means that they are not always stable. IIR filters can often implement a better transfer function than FIR filters, at the same order. Because of the recursive structure of IIR filters overflow can become a serious problem.

There are many ways to represent an IIR filter[2][3]. An early class of filters are the direct forms. Generally, these structures suffer from high coefficient sensitivity when implementing filters of higher orders. Usually filters up to grade two are implemented in this form. A filter with grade two is sometimes referred to as a biquad. One way to create filters with higher order is to cascade the biquads, see figure 7.7. In this way some of the coefficient sensitivity is suppressed. The filters we have implemented contain eight cascaded biquad sections. We have implemented the biquads in both direct form I and II.

Figure 7.7, Eight biquad sections in cascade
7.3.2.1 Direct Form I

The signal flow graph for a biquad section implemented in direct form I is shown in figure 7.8. It corresponds to the transfer function

\[
H(z) = \frac{a_0 z^2 + a_1 z + a_2}{z^2 - b_1 z - b_2}
\]

This form is well suited for implementation; it basically consists of pure MAC operations. In the tap memory, the coefficients are stored in the order \(a_{00} \ a_0 \ a_1 \ b_{01} \ b_{02}\) where the first index indicates which section the coefficient belongs to.

The program processes the 40 samples through one section at a time, the complete output from section one is computed before it enters section two and so forth. Since one might want to filter sequences bigger than 40 samples, the program stores the values in the delay line. It means that when the 40:th sample has been processed, the values that are supposed to be multiplied in the next clock cycle through \(a_{i1} \ a_{i2} \ b_{i1} \ b_{i2}\) are stored in memory, so they can be fetched when a new frame of samples arrive.

The first two MAC “blocks” before the main loop load values from memory that were “left” from the previous frame, as explained above. In the main MAC block, there are three pointers to the data memory. This is due to the fact that three adjacent samples are constantly needed to perform the filtering, since we have two delay elements. In the recursive loop, the previous sums are inputs to the \(b_{ij}\)-coefficients. The sum computed one cycle back in time is stored in a register, while the sum two cycles back is fetched from memory.

The result of the program is stored in “increasing order”. It means that the result of each section is stored with a 40-step offset compared to the result of the previous section. This is because in-place computation is a bit complicated since we are not allowed to write over a sample as soon as it has been processed, since it will be used in the two consecutive cycles.

In the table below, the kernel part of the program is listed. This indicates the efficient parallel structure.
<table>
<thead>
<tr>
<th>Inst. nr.</th>
<th>Instruction</th>
<th>MAC</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mpy // loaddm</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Mac // loaddm</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Mac // loaddm</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Mac // loaddm</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Mac // loaddm</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Rnd and Sat</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>Move2dm // move2dm</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

*Figure 7.9, Kernel part of IIR Direct Form I.*

**Behaviour level code:**

```c
{ /* One section at a time is calculated, repeat eight times to complete filtering. */
/* DP3,4,5,6 starts at same position, that is where the first sample resides */
ACR <= 0
ACR <= a0 * DM(DP3) + a1 * DM(DP0) + a2 * DM(DP0+1) + b1 * DM(DP0+2) + b2 * DM(DP0+3); /* a_i and b_i stored in registers */
/* DP0 points in memory to values that were “left” from the previous frame */
Round ACR SAT;
MEM (DP1) <= ACR, Reg1 <= ACR /* Sum available for next cycle */
DP1 <= DP1 +1 , DP3 <= DP3 + 1;
ACR <= 0;
ACR <= a0 * DM(DP3) + a1 * DM(DP4) + a2 * DM(DP0) + b1 * Reg1 + b2 * DM(DP0+2);
Round ACR SAT;
MEM (DP1) <= ACR, Reg1 <= ACR;
DP1 <= DP1 +1 , DP3 <= DP3 + 1, DP4 <= DP4 + 1;
For I=2 to 40 do [
   /* Remaining 38 samples filtered through the section */
   ACR <= 0 ;
   ACR <= a0 * DM(DP3) + a1 * DM(DP4) + a2 * DM(DP5) + b1 * Reg1 + b2 * DM(DP6); /* DP6 follows DP1 two steps behind */
   Round ACR SAT;
   MEM (DP1) <= ACR, Reg1 <= ACR;
   DP1 <= DP1 +1 , DP3 <= DP3 + 1, DP4 <= DP4 + 1, DP5 <= DP5 + 1, DP6 <= DP6 + 1.
]
/* One biquad-section completed. */
/* Save values in registers to memory, so they are available when next frame of samples arrive */
}
```

**Assembly code:**

```
load #2000 GPR0
move2dmd+ #100 GPR0  *start computing buffer
```
move2dmd+ #500 GPR0 *intermediate storage
move2dmd+ #140 GPR0 *output address
move2dmd+ #0 GPR0 *start address in TM
move2dmd+ #8 GPR0 *Number of sections in filter
move2dmd+ #40 GPR0 *Frame size
move2dmd+ #37 GPR0 *Frame size -3
move2tm #1 CONTROL *Fractional mode

*load the intermediate buffer with zeros. This is only done for the first frame
loaddmi #2001 GPR0
repeat #32
move2dmd+ #0 GPR0

*load frame to computing buffer:
loaddm #2000 GPR0   *Start address of comp. buffer is stated in the status buffer

load #0 GPR6
loaddm #2005 GPR14
loopr #2
loaddm AP6++ GPR25
move2dm AP5++ GPR25

loaddm #2001 GPR0
loaddm #2002 GPR1
loaddm #2003 GPR2
loaddm #2000 GPR3
loaddm #2000 GPR4
loaddm #2000 GPR5
loaddm #2002 GPR6
load #4 STEP0
setstep #1 #1
setstep #1 #2
setstep #1 #3
setstep #1 #4

* Computation of the sections starts here
loaddmi #2004 GPR14 *Number of sections in filter
loopr #51

*-----------one section is completed here-----
loadtm AP2++ GPR17 *a0i
loadtm AP2++ GPR18 *a1i
loadtm AP2++ GPR19 , loaddm AP5++ GPR24 *a2i
loadtm AP2++ GPR20 , loaddm AP4++ GPR23 *b1i
loadtm AP2++ GPR21 , loaddm AP3++ GPR22 *b2i

loaddm AP0 #0 GPR16 *200
loaddm AP0 #1 GPR25 *201
loaddm AP0 #2 GPR26 *202
loaddm AP0 #3 GPR10 *203

mpy GPR22 GPR17 ACC0, loaddm AP3++ GPR22
mac GPR16 GPR18 ACC0
mac GPR25 GPR19 ACC0
mac GPR26 GPR20 ACC0
mac GPR10 GPR21 ACC0
rnd ACC0 SAT
move2dm AP1++ GPR29
move GPR29 GPR11

mpy GPR23 GPR18 ACC0, loaddm AP4++ GPR23
mac GPR22 GPR17 ACC0, loaddm AP3++ GPR22
mac GPR11 GPR20 ACC0
mac GPR16 GPR19 ACC0
mac GPR26 GPR21 ACC0
rnd ACC0 SAT
move2dm AP1++ GPR29
move GPR29 GPR11

loaddm AP6++ GPR15
loaddmi #2006 GPR14
loopr #8
mpy GPR24 GPR19 ACC0, loaddm AP5++ GPR24
mac GPR23 GPR18 ACC0, loaddm AP4++ GPR23
mac GPR22 GPR17 ACC0, loaddm AP3++ GPR22
mac GPR11 GPR20 ACC0
mac GPR15 GPR21 ACC0, loaddm AP6++ GPR15
rnd ACC0 SAT
move2dm AP1++ GPR29
move GPR29 GPR11

mpy GPR24 GPR19 ACC0
mac GPR23 GPR18 ACC0
mac GPR22 GPR17 ACC0
mac GPR11 GPR20 ACC0
mac GPR15 GPR21 ACC0
rnd ACC0 SAT
move2dm AP1++ GPR29
move GPR29 GPR11

*Save for next frame

move2dm AP0 #1 GPR24 *X38
move2dm AP0 #2 GPR29 *Sum(-1)
move2dm AP0 #3 GPR15 *Sum(-2)
move2dm AP0++ GPR23 *X39
*-------------end of section computation----------------
addi #1 GPR4
addi #2 GPR5
addi #2 GPR6
*----------all sections completed---------------------------
7.3.2.2 Direct Form II

This structure realizes the same transfer function as the direct form I. The order of computation differs a bit though. Direct form II is computationally a more sequential algorithm and the structures also differ with respect to finite word length effects. The following scheme of computation is used:

\[
\begin{align*}
    w(n) &= x(n) - a_1 w(n-1) - a_2 w(n-2) \\
    y(n) &= b_0 w(n) + b_1 w(n-1) + b_2 w(n-2)
\end{align*}
\]

where

- \( x(n) \) is the input
- \( y(n) \) is the output
- \( w(n) \) is intermediate values
- \( a_k \) and \( b_k \) is filter coefficients

The taps are stored in the tap memory, order \( a_2, a_1, b_2, b_1, b_0 \), for each section. In the delay line buffer \( w(n-2) \) and \( w(n-1) \) for the biquads are stored.

First the pointers and status buffer are initiated. Then the input is loaded to the accumulator register and the backward loop is calculated. \( w(n) \) is temporary stored in a register. Then the forward loop is calculated and the values in the delay line are updated. The output are not saturated or rounded and is left in the accumulator register as input to the next biquad. When all biquads are done, the result is written to memory.

\[
\begin{array}{c|c}
\text{Delay line} & \text{Tap memory} \\
\hline
w_1(n-2) & a_{12} \\
\hline
w_1(n-1) & a_{11} \\
\hline
w_2(n-2) & b_{12} \\
\hline
w_2(n-1) & b_{11} \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
\text{T} & a_{22} \\
\hline
\text{T} & a_{21} \\
\hline
\text{T} & b_{22} \\
\hline
\text{T} & b_{21} \\
\hline
\text{T} & b_{20} \\
\end{array}
\]

Figure 7.10, IIR biquad section in direct form II.

Figure 7.11, Memory allocation for a two-stage IIR filter.
Below is a table of the kernel part of the program.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Clr</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Load</td>
<td></td>
<td></td>
<td>x</td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>Mac // load</td>
<td>x</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>Mac // load</td>
<td>x</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>rnd // sat</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Move</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Mpy // load</td>
<td>x</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>14</td>
<td>Move2dm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Mac // load</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Mac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Move2dm</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Figure 7.12, Kernel operations of IIR filter II.

Assembly code:

load #2200 GPR0
move2dmd+ #400 GPR0 *CB-pointer
move2dmd+ #0 GPR0  *tap top/start
move2dmd+ #39 GPR0  *bottom tap
move2dmd+ #400 GPR0 *CB-pointer
move2dmd+ #3300 GPR0 *out
move2dmd+ #0 GPR0  *input
move2dmd+ #40 GPR0  *frame size
move2dmd+ #8 GPR0   *nr of biquads
load #400 GPR0
repeat #25
move2dmd+ #0 GPR0   *clear CB

move2tmd #9 CONTROL

loaddmi #2201 AP1   *tap
loaddmi #2204 AP3   *out
loaddmi #2205 AP4   *input

move2tmd #4369 #105 *STEP 1-4
loadmmi #2201 TOP1
loadmmi #2202 BOTTOM1

loadmmi #2206 GPR14  *frame size
loopr #19       *start loop 1

loadmmi #2200 AP6  *get CB
loadmmi #2200 AP2  *store CB
loadmmi #2200 AP5  *get 2 CB

clracc ACC0
loadmm AP4++ GPR29  *acc=x(n)
loadmmi #2207 GPR14
loopr #10       *start loop 2
loadtm AP1++ GPR21 , loadmm AP6++ GPR20
macsub GPR21 GPR20 ACC0 , loadtm AP1++ GPR21 , loadmm AP6++ GPR20 *acc=x(n)-a2w(n-2)
macsub GPR21 GPR20 ACC0 , loadtm AP1++ GPR21 *acc=x(n)-a2w(n-2)-a1w(n-1)
rm ACC0 SAT , loadmm AP5++ GPR20
move GPR29 GPR17  *w(n)
mpy GPR21 GPR20 ACC0 , loadtm AP1++ GPR21 , loadmm AP5++GPR20 *acc=b2w(n-2)
move2dm AP2++ GPR20 *w(n-2)<=w(n-1)
mac GPR21 GPR20 ACC0 , loadtm AP1++ GPR21  *acc=b2w(n-2)+b1w(n-1)
mac GPR21 GPR17 ACC0  *acc=b2w(n-2)+b1w(n-1)+b0w(n)
move2dm AP2++ GPR17 *w(n-1)<=w(n)        end loop 2
rm ACC0 SAT
move2dm AP3++ GPR29  
*save output end loop 1
7.3.3 Adaptive FIR filter

Speakerphones and hands-free cellular phones are widely used around the world today. One problem that might occur in these applications is echo. The speech from the far-end side are reflected by the walls of the room and then transmitted back to the sender. To solve this problem, adaptive filters are used. Adaptive filters change their coefficients during run time. We use the least-mean-square (LMS) algorithm to update the coefficients.

\[ b_k(i+1) = b_k(i) + 2\beta e(i)x(i-k) \]

where
\[ e(i) = d(i) - y(i) \]
\[ \beta \] is a user defined constant

and

\[ d(i) \] is the desired output

\[ y(i) = \sum_{k=0}^{m-1} b_k x(n - k) \]

We have implemented an adaptive FIR filter for modelling of an unknown transfer function, e.g. an echo path. This can be described by the block schedule below. \( H(z) \) is the unknown transfer function and \( W(z) \) is the adaptive filter.

![System identification using adaptive filter](image)

**Figure 7.13, System identification using adaptive filter**

**Implementation**

The input and the desired output are stored in the data memory. The taps are stored in the tap memory. It is a framed-based program and uses the frame based FIR program for filtering. The single sample based program is not described below. Basically it is the same without the outer loop.

The program starts with a set-up phase like the one in the FIR filter. The difference is that the taps are initiated to a start value. Then the input is passed through a FIR filter. The output is compared with the desired output and the coefficients are updated according to the LMS-algorithm.
Below is a table of the kernel part of the program.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loaddm</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>address</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sub</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>mpy</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>clr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>macro</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>rnd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>move2tm</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7.14, Kernel operations of LMS part of an adaptive filter.*

**LMS Adaptive FIR**

Each frame contains 40 samples. We use a computing buffer that contains 16 samples, the same size as the frame in TM containing the 16 filter coefficients. This is convenient, due to the fact that pointers to coefficients in TM and samples in the computing buffer in DM can be increased at the same time using circular buffers. This gains speed when implementing in assembly code.

**Behaviour level code:**

```c
{  
  For i=0 to 39 do [  
    ACR <= 0;  
    For k=0 to 15 do [  /* Identical to FIR program */  
      ACR <= ACR + DM(DP1) * TM(TP1);  /* Y(i) = sum (k=0 -> 15) b_k * x(i-k) */  
      DP1 <= DP1 + 1 , TP1 <= TP1 + 1  
    ]  
    Round ACR SAT;  
    e(i) = d(i) – y(i);  /* Create error signal , d(i) = desired output, stored in TM*/  
    /* Update coefficients using least mean square algorithm */  
    For k=0 to 15 do [  
      ACR <= b_k(i);  
      ACR <= ACR + Reg1 * DM(DP2) /* Reg1 = 2 * e(i) * β */  
      /* b_k(i+1) = b_k(i) + 2 * β * e(i) * x(i-k) */  
      Round ACR SAT;  
      MEM(TP2) <= ACR;  /* Store coefficient */  
      TP2 <= TP2 + 1 , DP2 <= DP2 + 1;  
    ]
  ]}
```

53
Assembly code:

load #2000 GPR0
move2dmd+ #100 GPR0  *top CB
move2dmd+ #115 GPR0  *bottom CB
move2dmd+ #100 GPR0  *last CB data
move2dmd+ #3000 GPR0  *out
move2dmd+ #0 GPR0  *frame start
move2dmd+ #40 GPR0  *frame length
move2dmd+ #0 GPR0  *top/start Tap
move2dmd+ #15 GPR0  *bottom tap
move2dmd+ #10 GPR0  *B coefficient
move2dmd+ #40 GPR0  *start of desired output

load #101 GPR0
repeat #15
move2dmd+ #0 GPR0  *clear CB

*----- LMS -----*
load #1000 GPR0
load #0 AP7  *initialize tm
repeat #16
move2tm AP7++ GPR0

*-----
move2tmd #13 CONTROL  *modular addressing + fractional
loadaddmi #2000 TOP0  *top CB
loadaddmi #2001 BOTTOM0  *bottom CB
loadaddmi #2002 AP0  *last CB
loadaddmi #2003 AP5  *out
load #1 STEP0  *step 0
setstep #1 #1  *step 1
loadaddmi #2004 AP6  *frame start
loadaddmi #2006 TOP1  *top T
loadaddmi #2006 AP1  *last T
loadaddmi #2007 BOTTOM1  *bottom T

*----- LMS -----*
loadaddmi #2009 AP7  *start of desired output

*-----
loadaddmi #2008 GPR23  *B
loadaddmi #2005 GPR14  *number of samples in frame
loopr #22  *start loop 1

clracc ACC0 , loadtm AP1++ GPR21 , loaddm AP6++ GPR20  *load tap+ sample from frame
move2dm AP0++ GPR20  *store sample in CB
loaddm AP0++ GPR20  *load oldest sample in CB
repeat #15  *number of taps -1
7.3.4 Wave digital filters

Wave digital filters is a type of IIR filter that have many advantageous properties, for example they are the stable and have low sensitive to variations in element values. The wave digital filter is derived from an analog reference filter. The filter is described by current and voltage waves that pass through the filter. Time delays and multiplications are used to describe the elements. Different types of adaptors are needed connect them together. The adaptor type depends on the interconnect structure between the elements. There are mainly three different methods to construct the filters, Richards’, ladder and lattice structures[2].
**Two-port adaptor**

The most fundamental adaptor is the symmetric two-port adaptor. The $\alpha$ coefficient depends on the impedance of the network.

![Two-port adaptor interconnect](image)

*Figure 7.15, Two-port adaptor interconnect.*

The adaptor is described by the wave-flow graph below.

![Wave-flow graph of symmetric two-port adaptor](image)

*Figure 7.16, Wave-flow graph of symmetric two-port adaptor.*

With only the symmetric two-port adaptor and delay elements, Richards’ structure filters can be constructed, see below. Different types of adaptors are needed if the system is more complex.

![WDF of third order doubly terminated Richards' structure](image)

*Figure 7.17, WDF of third order doubly terminated Richards' structure.*
**Series three-port adaptor**

There are two types of three-port series adaptors, general and reflection free. The reflection free adaptor is used to prevent delay free loops. It is a special case of the general adaptor, when one of the coefficients is equal to one. Below is the wave-flow graph for the general adaptor shown. The computation works according to the data dependencies. It starts with the inputs \((A_1, A_2, A_3)\) and work toward the outputs \((B_1, B_2, B_3)\).

![Wave-flow graph of series three-port adaptor](image)

**Parallel three-port adaptor**

There are two types of parallel three-port adaptors, general and reflection free. The reflection free adaptor is used to prevent delay free loops. It is a special case of the general adaptor, when one of the coefficients is equal to one. Below is the wave-flow graph for the general adaptor shown. The computation works according to the data dependencies. It starts with the inputs \((A_1, A_2, A_3)\) and work toward the outputs \((B_1, B_2, B_3)\).

![Wave-flow graph of parallel three-port adaptor](image)
Implementation
When implementing a WDF in practice, the filter is broken down to differential equations and the computations are directly mapped from them. This gives the best and the most simple solution to implement a certain filter. To simplify the benchmark we have instead only implemented the different adaptors. This is because although the computations can be in different orders the starting point is the wave-flow graphs for the adaptors. We also used a simple ordering of the input and output data in registers, with no extra moves. This is because in a real application the data flow has to be mapped after the differential equations and then a better use of the registers and memory is possible anyway.

Below is a table of the kernel part of a two-port adaptor. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clracc/load</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>addadd</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>add3</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

*Figure 7.20, kernel part for two-port adaptor.*

Below is a table of the kernel part of a parallel three-port adaptor. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadacc//load</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>subsub</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>macsub//load</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>rnd // sat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>add3</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

*Figure 7.21, kernel part for parallel three-port adaptor.*

**Assembly code:**
The code of the benchmark isn’t presented here. To do a good implementation the structure has to be directly mapped. The above tables give a little insight of which operations that are needed.
7.4 Division
This program performs integer division of two signed numbers.

\[ \frac{A}{D} = Q + \frac{R}{D} \]

where

A is the dividend
D is the divisor
Q is the quotient
R is the remainder

For fixed-point division, the range of operands must be predetermined. If D is n-bits, A must be 2n-bits. This gives that Q and R becomes n-bits long.

Implementation
We use a recursive algorithm [4]. It is based on binary integers.

\[
R_0 = A \cdot 2^{-n} \\
R_{j+1} = 2R_j - q_{n-(j+1)}D \\
\text{for } j=0, 1, \ldots n-1
\]

when

\[
q_{n-(j+1)} = \begin{cases} 0 & 2R_j < D \\ 1 & 2R_j \geq D \end{cases}
\]

This gives

\[
Q = \sum_{i=0}^{n-1} q_i 2^i
\]

\[
R_{j+1} = \begin{cases} 2R_j - D & q_{n-(j+1)} = 1 \\ 2R_j & q_{n-(j+1)} = 0 \end{cases}
\]

The division can be performed using absolute values and update the output with

\[
q_n = a_n \oplus d_n \\
r_n = a_n
\]

We implemented this algorithm for a 16-bit dividend (A) and an 8-bit divisor (D), taken from registers.

First some set-up code in needed. Then the sign of A and D are stored. If any of them are negative they are inverted. Then D is subtracted from R_0. If the result is positive the result of the division is 0 with the rest A. Else the kernel of the program starts. 2R_n is compared with D. If 2R_n is larger, q_n is set and the result is stored as R_{n+1}. If D is largest, R_n is just shifted one
bit. The algorithm starts with the highest bit and works its way down. When all bits are done, the sign of the results are updated.

Below is a table of the kernel part of the program. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Move</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Sub</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Move</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Or</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Shift</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7.22, Kernel operations of Division program.*

**Assembly code:**

move2tmd #1 CONTROL
clracc ACC0
load #-41 GPR20  *A
load #7 GPR21  *D
load #7 GPR24  *number o bits in divider, result and rest
load #0 GPR22  *Q=0
load #64 GPR23  *qn vector, ska bero på antal bitar
neg GPR24 GPR25  *-n
count GPR20 GPR16  *pos/neg rest
neg GPR20 GPR20 ne
count GPR21 GPR17  *pos/neg value
neg GPR21 GPR21 ne
loadacc GPR20 ACC0
lshl #1 ACC0  * 2*Rn
move GPR29 GPR20  * Ro
sub GPR21 GPR20  * Ro-D
bragtr #10
nop
move GPR24 GPR14  *loop n times
loopr #6
lshli #1 ACC0  * 2*Rn
move GPR29 GPR20
sub GPR21 GPR20
move GPR20 GPR29 ge  *R=2R if R<0
or GPR23 GPR22 ge
rsli #-1 GPR23  *1->qn
*next qn

*----- for neg -----*
7.5 Vector functions

7.5.1 Vector addition

This program adds two vectors and stores the output vector.

\[
\begin{bmatrix}
A \\
B
\end{bmatrix} + \begin{bmatrix}
A_0+B_0 \\
A_1+B_1
\end{bmatrix} = \begin{bmatrix}
A_0+B_0 \\
A_1+B_1
\end{bmatrix}
\]

Figure 7.23, Vector addition

Implementation

The two in vectors are assumed to be located in separate memories, one in tap and the other in the data memory. If both vectors are in the same memory, it requires an extra cycle.

First some initiation is performed. Then the two first values from the vectors are read and the addition is performed. Parallel with the addition, one memory fetch is performed. The other fetch is done after the result has been written to memory. All values are added and the results are written to memory.

The program now requires the vector length to be more than 2.

Below is a table of the kernel part of the program. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add // load</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>move2dm</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>load</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.24, Kernel operations of vector addition.

Assembly code:

load #2110 GPR0 *start of control field
move2dmd+ #0 GPR0 *start of vector 1
move2dmd+ #0 GPR0 *start of vector 2
move2dmd+ #20 GPR0 *vector length /2
move2dmd+ #3100 GPR0 *start of output vector
move2tmd #2 CONTROL
loaddmi #2110 AP5    *vector1
loaddmi #2111 AP6    *vector2
loaddmi #2113 AP7    *out

loaddmi #2112 GPR14
loopr #5
loadm AP6++ GPR21, loaddm AP5++ GPR20    *load inputs
loadm AP6++ GPR23, loaddm AP5++ GPR22
addadd GPR20 GPR21 GPR22 GPR23 ACC0    *calculate results
move2dm AP7++ GPR29    *save output
move2dm AP7++ GPR28

7.5.2 Vector dot product
This program calculates the vector dot product of two vectors

\[
\begin{bmatrix} A \\ B \end{bmatrix} = A_0B_0 + A_1B_1 + \ldots
\]

Figure 7.25, Vector dot product

Implementation
The two in vectors are presumed to be located in separate memories, one in tap and the other in the data memory. If both vectors are in the same memory, it requires \(2+N\) extra cycles.

First some initiation is performed. Then the two first values from the vectors are read. The two values are then multiplied with each other. Parallel with the multiplication, two memory fetches are performed. The rest of the values are multiplied and accumulated, parallel with the memory fetches. When all are done, the result is written to memory.

The program now requires the vector length to be more than 2.

Below is a table of the kernel part of the program.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mac // load</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Figure 7.26, Kernel operations of vector dot program.

Assembly code:

load #2105 GPR0    *start of status buffer
move2dmd+ #0 GPR0  *start of vector 1
move2dmd+ #0 GPR0  *start of vector 2
move2dmd+ #3200 GPR0  *output
move2tmd #0 CONTROL
loaddmi #2105 AP5
loaddmi #2106 AP6

clracc ACC0, loadtm AP6++ GPR21, loaddm AP5++ GPR20
repeat #39
mac GPR21 GPR20 ACC0, loadtm AP6++ GPR21, loaddm AP5++ GPR20 *acc<=acc + a(k)*b(k)
mac GPR20 GPR21 ACC0 *acc<=acc + a(k)*b(k)
rnd ACC0 SAT
loaddmi #2107 AP7 *output pointer
move2dm AP7++ GPR28 *save lower bits
move2dm AP0 #0 GPR29 *save higher bits

7.5.3 Vector maximum
The program computes the largest value in a vector. It also saves the internal position of the value.

Implementation
If there are two or more data samples with the same value, the first is kept as the largest.

The program begins with some initialization. Then the first value is fetched from memory and stored as the largest. After that the next value is fetched. The temporary maximum value and the recently fetched are compared. If the new value is bigger, it and its address is saved. Regardless of which of the values is biggest the next value in the vector is fetched. When the whole vector is done, the maximum value and address are written to memory.

The vectors must consist of at least four values.

Below is a table of the kernel part of the program. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>max</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7.27, Kernel operations of program for maximum search.*

Assembly code:

load #2100 GPR0 *start of status buffer
move2dmd+ #0 GPR0 *start of vector
move2dmd+ #3202 GPR0 *start of output

move2tmd #2 CONTROL
loaddmi #2100 AP5

move AP5 GPR21 *max adress
loaddm AP5++ GPR20 *max value
move GPR20 GPR22 *initial value
repeat #39
max AP5++ GPR22 GPR20 GPR21

comp GPR20 GPR22 *compare max and temp values
move AP0 GPR21 gt
move GPR22 GPR20 gt

loaddmi #2101 AP6 *output pointer
move2dm AP6++ GPR20 *save max value
move2dm AP6++ GPR21 *save max adress

7.6 Number representation conversion

The processor works with 16-bit fixed-point numbers. Some interfaces can use other number representations, and then some type of conversion has to be performed. We have looked at an IEEE standard for 32-bit floating point numbers.

\[
\begin{align*}
\text{Sign Bit} & \\
-2^0 & 2^{-1} & 2^{-2} & \ldots &
\end{align*}
\]

_Figure 7.28, 16-bit fix point_

\[
\begin{align*}
\text{Sign Bit} & & \text{Implied mantissa bit, 1.} \\
\text{Mantissa} & & \text{Exponent} \\
-2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} & \ldots & -2^3 & 2^2 & 2^1 & 2^0
\end{align*}
\]

_Figure 7.29, 12-bit floating point_

7.6.1 Fixed to floating

This program converts a 16-bit integer to a 32-bit floating-point value, according to the IEEE standard.

**Implementation**

This program only use registers, not memory.

First the sign of the integer is checked. If it is negative, the value is inverted and a bit is set in a register. Then the exponent is calculated. It is done by subtract the number of zeros in the beginning of the value, from the maximal possible. The implied bit is then shifted out and the sign bit is added.
Below is a table of the kernel part of the program. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bittest</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>con. jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sub</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>shift</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.30, Kernel operations of fixed to floating-point conversion.

Assembly code:

load #20 GPR20  *input
move2tmd #0 CONTROL

count GPR20 GPR19
neg GPR20 GPR20 ne

count GPR20 GPR21

ori #0 GPR20
braeqr #10
clracc ACC0
load #143 GPR18  *start value of exponent (offset +128)
sub GPR21 GPR18  *exponent value
lsl GPR21 GPR20   *shift to first number+ imp. bit and update exponent

move GPR18 GPR29
ori #0 GPR19
braeqr #4
move GPR20 GPR28
lshli #7 ACC0
ori #b1000000000000000 GPR29  *sign bit

*answer in high GPR29, low GPR28

7.6.2 Floating to fixed
This program converts a 32-bit floating-point value, according to the IEEE standard to a 16-bit integer.

Implementation
This program only use registers, not memory.

First the sign bit is stored. Then the program checks so the exponent is in between 0 and 15. If the exponent is less than 0 the output is set to 0 and the program is finished. If the exponent is larger than 15, the value is set to 16384 and the program jump to the sign stage, see below. If
the exponent is between 0 and 15, the implied bit is inserted and the value is left shifted according to the exponent. In the last step the sign of the value is set by checking the stored sign bit.

Below is a table of the kernel part of the program. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>shift</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>and</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7.31, Kernel operations of program for floating-point value to integer conversion.*

**Assembly code:**

load #b0100001000100000 GPR1 *input-high
load #b0000000000000000 GPR0 *input-low
move2tmd #0 CONTROL
move32 GPR0 GPR28
bittest #b1000000000000000 GPR29

braner #4
load #0 GPR19 *temp register for sign bit
lshli #-7 ACC0 *set temp-sign bit
load #1 GPR19
andi #hFF GPR29
subi #128 GPR29
braltr #12
clracc ACC0 lt
bittest #hF0 GPR29 *if less than one set output to 0
bragtr #6 *if too big set output to max value
move GPR29 GPR21
load #1 GPR29 *implied bit
jmp #6
load #b01111111111111 GPR27
nop
lshl GPR21 ACC0 *shift mantissa according to exponent
andi #1 GPR19 *set sign of data
neg GPR29 GPR29 ne *answer in register GPR29
7.7 FSM, Finite State Machine

The benchmark should test control processing code like, switch/case statements, if-then statements, bit manipulation, stack manipulation and conditional subroutine calls. For these types of applications the memory use is often the most important factor. Therefore implementations of this benchmark are optimized for minimum memory use, then for execution speed.

Implementation

We couldn’t find the program implemented by BDTI. We implemented a fictive FSM, see figure below.

![Figure 7.32, FSM interconnections.](image)

The program has a short setup phase, because the state of the system is only stored in register. The inputs are read from memory and the kernel starts checking in which state the program is. Then it checks for that state, what actions should be taken according to the input.

Below is a table of some part of the program. It shows how the program detects which state it is in, for a three state machine. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>comp</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>con. jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>comp</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>con. jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 7.33, State selection part of FSM.](image)

Assembly code:
The assembly code isn’t presented here, because the implemented FSM isn’t important. The table above shows the important state selection phase. The actual computations in the different states are often only short control applications.
7.8 DCT, Discrete Cosine Transform

The DCT transforms data into a format that easily can be compressed. This makes it ideal for image and speech compression algorithms, for example is it used in mpg decoding. The data is subdivided into smaller, one or two-dimensional blocks. The blocks are then processed individually. One-dimensional is used for speech and two-dimensional for image processing. The DCT itself doesn’t perform any compression, it just transform the data so other algorithms easily can compress it.

The definition for one-dimensional DCT is

\[ F(u) = \frac{2c(u)}{N} \sum_{m=0}^{N-1} f(m) \cos \left( \frac{(2m+1)u\pi}{2N} \right) \]

for \( u = 0, 1, 2, \ldots, N-1 \)

where

\[ c(u) = \begin{cases} \sqrt{2}, & \text{for } u = 0 \\ 1, & \text{for } u = 1, 2, 3, \ldots, N-1 \end{cases} \]

The two-dimensional DCT transforms a two-dimensional data array into another two-dimensional output. It performs one-dimensional DCT:s on each row and column of the matrix, see below.

\[ F(u, v) = \frac{4c(u, v)}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f(m, n) \cos \left( \frac{(2m+1)u\pi}{2N} \right) \cos \left( \frac{(2n+1)v\pi}{2N} \right) \]

for \( u = 0, 1, 2, \ldots, N-1 \)

Figure 7.34, DCT operation

It is mathematically defined as
where
\[ c(u,v) = \begin{cases} 
\frac{1}{2}, & \text{for } u=v=0 \\
1, & \text{for } u,v = 1, 2, 3, \ldots, N-1 
\end{cases} \]

**Implementation**
There exist many algorithms for accelerating the computation of the DCT. We use a Fast DCT algorithm presented by H. S Hou. It is based on that higher order DCT are generated from lower. Below is the computation algorithm shown for an 8-bit input.

---

**Figure 7.35, Hous Computation algorithm for 8 bits.**

The inputs are assumed to be 16-bit fractional numbers, taken from the data memory, row by row. The coefficients are stored in the tap memory in the order \( c_1, c_5, c_9, c_{13}, c_2, c_{10}, c_4 \).

The program starts with the setup of the status buffer and address pointers. The computes the results first row by row then column by column. The kernel part of the program starts with the fetch of several values that are needed for the computations. Then the actual computations are performed and the results are written back to memory, for each row. All eight rows are calculated according to the one-dimensional DCT, described above in figure 7.35.
address pointers are change to perform column based computations and the rest of the
algorithm is completed.

Below is a table of a part of the kernel for the program. The table shows which units are used.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load/load</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>load/load</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>sub/load</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>mpy/load</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>load</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>move</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>add</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>sub/load</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>11</td>
<td>mpy/load</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>add</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>move</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>15</td>
<td>mpy/load</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>16</td>
<td>butterfly</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>move</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>butterfly</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>butterfly</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>butterfly</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7.36, Kernel operations for middle subroutine of DCT program.*

**Assembly code:**

setup phase (AP0 AP1 AP5)

load #1 STEP0
setstep #1 #1
load #5 GPR12
load #1 GPR13

*------------------------------------------------
!b
load #8 GPR14
loop

loadtm AP5++ GPR15 , loaddm AP0++ GPR20
loaddm AP0 GPR12 GPR27
add3 GPR20 GPR27 GPR1
loadtm AP5++ GPR16 , loaddm AP0++ GPR21
sub GPR20 GPR27 , loaddm AP0++ GPR20
mpy GPR15 GPR27 ACC0 SAT RND, loadtm AP5++ GPR15, loaddm AP0++ GPR27
loadm AP0 GPR13 GPR22
move GPR29 GPR2
add3 GPR20 GPR22 GPR3
sub GPR22 GPR20 loaddm AP0++ GPR22
mpy GPR16 GPR20 RND SAT, loadtm AP5++ GPR16, loaddm AP0++ GPR20
add3 GPR22 GPR27 GPR5
move GPR29 GPR4
sub GPR27 GPR22, loaddm AP0++ GPR27
mpy GPR15 GPR22 RND SAT, loadtm AP5++ GPR15
butterfly GPR21 GPR27 GPR18 ACC1
move GPR29 GPR6
butterfly GPR30 GPR3 GPR19 ACC0
butterfly GPR5 GPR1 GPR20 ACC1
butterfly GPR28 GPR30 GPR21 ACC0
load #sq(2) GPR1
mpy GPR1 GPR28 ACC0 RND SAT, loadtm AP5++ GPR1
move GPR1 GPR2
move2dm AP6++ GPR29
mpy GPR16 GPR18 ACC0 RND SAT, loadtm AP5++ GPR16
butterfly GPR6 GPR2 GPR22 ACC1
butterfly GPR29 GPR6 GPR23 ACC0
asli #1 GPR1
butterfly GPR28 GPR31 GPR24 ACC0
move2dm AP6++ GPR28
move GPR28 GPR6
butterfly GPR20 GPR19 GPR20 ACC0
move2dm AP6++ GPR28
mpy GPR20 GPR1 ACC1 RND SAT
sub3 GPR31 GPR28 GPR16
butterfly GPR22 GPR23 GPR22 ACC0
mpy GPR22 GPR1 ACC1 RND SAT
nop
sub3 GPR28 GPR29 GPR23
asli #1 GPR28
sub3 GPR6 GPR8 GPR22
move2dm AP6++ GPR22
mpy GPR24 GPR1 ACC0 RND SAT
asli #1 GPR23
sub3 GPR22 GPR29 GPR20
mpy GPR2 GPR21 ACC1 RND SAT
sub GPR20 GPR23
move2dm AP6++ GPR21
move2dm AP6++ GPR20
move2dm AP6++ GPR16
move2dm AP6++ GPR23

*---------------------------------------------------*
xori #h0009 STEP0
braeq ?a
load #40 GPR12
setstep #8 #1

jmp ?b
load #8 GPR13
nop

nop

7.9 Crosscorrelation

The crosscorrelation is implemented as follows: \( R[d] = \sum_{i=1}^{N-1} x[i]y[i+d] \). N=40 and d=0-15.

The kernel part can be viewed below:

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loaddmi</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>loopr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Clracc//load</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>repeat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Mac//load</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>mac</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>rnd</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Move2dm</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>addi</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>move</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Behaviour level code:

{ 
For I=0 to 15 do [ 
    ACR <= 0;
    For I=0 to 31 do [ 
        ACR <= ACR + DM(DP) * TM(TP);
        DP <= DP + 1, TP <= TP + 1;
    ]
    Round ACR SAT;
    MEM(DP) <= ACR;
    DP <= DP + 1;
    TP <= Counter value; /* Store in a register which loop is executed, for each */
}
/* loop done, start a position higher in TM */
/* DM uses modulo addressing */
]

Assembly code:

*-------setup-------
load #2000 GPR0
move2dmd+ #3000 GPR0 *Output address R[0], R[1]...
move2dmd+ #0 GPR0 *Start address for y(i) TM
move2dmd+ #16 GPR0 *d=0:15
move2dmd+ #40 GPR0 *Number of samples
move2dmd+ #39 GPR0 *Samples -1

*-----Initiate------
loaddmi #2001 GPR0
loaddmi #2001 GPR1
loaddmi #2001 TOP0
loaddmi #2004 BOTTOM0
load #1 STEP0
setstep #1 #1
loaddmi #2000 GPR7
move2tmd #5 CONTROL

*------ Start computation---------------------
load #0 GPR20
loaddmi #2002 GPR14 *d:0-15
loopr #8
clracc ACC0 , loadtm AP1++ GPR25 , loaddm AP0++ GPR17
repeat #39
mac GPR25 GPR17 ACC0 , loadtm AP1++ GPR25 , loaddm AP0++ GPR17
mac GPR25 GPR17 ACC0
rnd ACC0 SAT *round and sat
move2dm AP7++ GPR29 *store sample
addi #1 GPR20 * --------------------------
move GPR20 GPR1 * y(0)-y(15), y(1)-y(16)...
*--------------------------------------------------
7.10 Autocorrelation

Autocorrelation is used heavily in signal processing and is quite similar to the crosscorrelation treated above. We have used the formula \( y(n) = ay(n-1) + m^*x(n)x(n-k) \) for calculating the autocorrelation. The interesting part in this algorithm is the 16 * 32 bit multiplication. The kernel part is found below. As can be seen, it takes 5 instructions to perform \( m^*x(n)x(n-k) \), that is row 4 to 8 in the figure below.

<table>
<thead>
<tr>
<th>Ins nr</th>
<th>Ins</th>
<th>Mac</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Tap Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Loaddm, tm</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>loopr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Mpy//load</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Mpy</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>mpyus</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L_shli</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>8</td>
<td>Add32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Mpy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Add32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Rnd</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>Move2dm</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>13</td>
<td>Move</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembly code:

```assembly
loadtm AP6++ GPR8 , loaddm AP5++ GPR18
loaddm #2003 GPR14
loopr #10
mpy GPR8 GPR18 ACC0 , loadtm AP6++ GPR8 , loaddm AP5++ GPR18
mpy GPR29 GPR19 ACC1
mpyus GPR28 GPR19 ACC0
lshli #-16 ACC0
add32 GPR30 ACC0 SAT
mpy GPR17 GPR21 ACC1 *GPR21 is y(n-1)
add32 GPR30 ACC0
rnd ACC0 SAT
move2dm AP7++ GPR29
move GPR29 GPR21

mpy GPR8 GPR18 ACC0
mpy GPR29 GPR19 ACC1
mpyus GPR28 GPR19 ACC0
lshli #-16 ACC0
add32 GPR30 ACC0 SAT
mpy GPR17 GPR21 ACC1
add32 GPR30 ACC0
```

74
rnd ACC0 SAT
move2dm AP7++ GPR29
move GPR29 GPR21
Chapter 8  Improvements

8.1 Modifications on the processor
From the benchmarking we found four different areas of the DSP-processor that could be improved.

1. New instructions that are suited for the algorithms we have used in the benchmark. Though, we have tried to maintain the instruction set as general as possible and there is still space left in the instruction space. Below, we have listed these new instructions and a motivation why we have introduced them into the instruction set. In Appendix A, a description of how to use them is found.

2. Conditional execution for “smoother programming”. With conditional execution instructions can be performed or discarded, depending on the condition. This can give programs a more static behavior, which means that it take the same time to execute, independent on the inputs. It can also improve performance of programs with small execution branches. It is also described in Appendix A.

3. The need for a 40 bit shifter. With the previous 32 bit shifter, it was impossible to utilize the guard bits of the accumulator registers.

4. Change the accumulator register to consist of two registers instead of three, one 16 bit and one 24 bit. The 24 bit register can be used as a general purpose 16 bit register. This releases two registers at the programmer’s disposal.

5. The step registers consists of two registers instead of seven. It releases five registers at the programmer’s disposal.

8.2 Modifications to the Assembler
One part was to change the translation between the assembly code and the binary code, because we changed the instruction set. The biggest addition we made to the assembler was adding some programming support. Error messages are now presented with the corresponding line number, which simplifies the error trace. Another programming feature that we have added is the use of labels. Labels can be used to simplify the jump and call functions. Instead of giving an exact line number a label can be used. The label is then translated by the assembler to the corresponding line number. This makes it much easier to make changes in programs.
8.3 New instructions

See Appendix A, for more details about the instructions.

**Move2dmd**
Memory instruction – write to memory, immediately data  
**Syntax**
move2dmd #data GPRx  
**Description**
The instruction writes immediately data to the memory address pointed out by the register.  
**Reason**
This instruction makes it easier to store values in the memory. Before the value had to be loaded into a register before it could be written to memory.

**move2dmd+**
Memory instruction – write to memory, immediately data  
**Syntax**
move2dmd+ #data GPRx
**Description**
The instruction writes immediately data to the memory address pointed out by the register and the increases the register with its step length.  
**Reason**
This instruction makes it easier to store values in the memory and it is useful when several adjacent addresses are used. Before the value had to be loaded into a register before it could be written to memory.

**move2tmd**
Memory instruction – write to memory, immediately data and address.  
**Syntax**
move2tmd #data #address  
**Description**
The instruction writes immediately data to the memory address pointed out by the immediate address. The addresses that can be addressed are only those 128 in the control part of the memory, see chapter 4 for more information.  
**Reason**
The instruction makes it easy to write data to the control part of the memory.

**Loadacc**
Data move instruction - load acc with sign extension.  
**Syntax**
loadacc GPRx ACCx [L]  
**Description**
This instruction loads one accumulator register with a value from a register. Both registers in the accumulator are set, with sign extension of the guard bits.
**Reason**
This instruction makes it possible to load the accumulator register in one clock cycle, instead of two.

**Add3 and Sub3**
Arithmetic instruction – addition/ subtraction

**Syntax**
add3 GPRx GPRy GPRz
sub3 GPRx GPRy GPRz

**Description**
These two instructions add or sub two registers, and store the result into a third.

**Reason**
The instructions gives the programmer more freedom when choosing registers.

**Addadd and Subsub**
Arithmetic instruction – addition/ subtraction

**Syntax**
addadd GPRx GPRy GPRz GPRw ACCx
subsub GPRx GPRy GPRz GPRw ACCx

**Description**
These two instructions make two parallel additions or subtractions. The results are stored in one accumulator register.

**Reason**
The instructions take advantages of both adders on the processor and can therefore speedup some algorithms.

**Count**
Arithmetic instruction – count leading ones or zeros.

**Syntax**
count GPRx GPRy [z]

**Description**
The instruction counts the leading ones or zeros of a register and store the result in another.

**Reason**
The instruction can simplify control parts of programs.

**Butterfly**
Arithmetic instruction – butterfly addition and subtraction.

**Syntax**
butterfly GPRx GPRy GPRz ACCw

**Description**
The instruction performs one addition and one subtraction in parallel. The results are stored in one accumulator register and one other general purpose register.

**Reason**
The instruction takes advantages of both adders on the processor and can therefore speedup some algorithms.
Max and Min
Arithmetic instruction – maximum/minimum and address search with parallel load.
Syntax
max GPRx APx++ GPRy Apy
min GPRx APx++ GPRy APy
Description
The instruction compares the values from two registers and saves the desired one. It also stores the address pointer to that value.
Reason
The instruction is useful for max/min searches of vectors.

Setstep
Logic instruction – set step register.
Syntax
Setstep #data #step
Description
The instruction sets the step for one of the address pointers one to four.
Reason
The instruction makes it easier for the programmer to define the step length.

Bittest
Logic instruction – bitwise and with immediate data
Syntax
bittest #data GPRy
Description
The instruction performs bitwise and updates the flags. The result is not stored.
Reason
The instruction can be useful in control parts of programs.

Setfield
Logic instruction – set bit-field in register.
Syntax
setfield GPRx GPRy GPRz GPRw
Description
The instruction sets a field in a register. The value, length and start position are defined.
Reason
The instruction can be useful in control parts of programs and for protocol processing.

Getfield
Logic instruction – get bit-field from register.
Syntax
getfield GPRx GPRy GPRz GPRw
Description
The instruction retracts a field from a register to another. The length and start position are defined.
Reason
The instruction can be useful in control parts of programs.
**Rti**
Program flow instruction – return from interrupt.

**Syntax**
Rti [i]

**Description**
The instruction returns the program to normal after interrupt.

**Reason**
The instruction are needed when interrupts are used.

### 8.4 Modified instructions

**Loadmm and Loadtm**
The instructions addressing modes has been extended with variable offset, taken from a register. This can make programs more adaptive.

**Repeat**
The instructions addressing modes has been extended with indirect data. This can make programs more adaptive.

**Abs**
The instruction updates the N flag. Can be used for program control.

**Add and sub**
The add and sub instructions have markers to decide if the carry should be used or not. The addc and subc instructions have been removed.

**Shifts**
All shift instructions has the ability to use the carry or not.
Chapter 9  Results from the benchmark

This chapter describes the results from the benchmarks in comparison to the BDTI benchmarks. Below a table with the results is shown. The speedup column indicates the improvement after the changes to the processor were performed and the BDTI column indicates the average value of the processors examined by BDTI.

9.1 Overview

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>Total</th>
<th>Speedup</th>
<th>Avg BDTI * kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex FFT</td>
<td>18763</td>
<td>19816</td>
<td>30 %</td>
<td>10000</td>
</tr>
<tr>
<td>DCT</td>
<td>860</td>
<td>874</td>
<td>64 %</td>
<td>3298 (**)</td>
</tr>
<tr>
<td>IIR form I</td>
<td>2435</td>
<td>2450</td>
<td>0%</td>
<td>----</td>
</tr>
<tr>
<td>IIR form II</td>
<td>3485</td>
<td>3500</td>
<td>0%</td>
<td>----</td>
</tr>
<tr>
<td>Vector add</td>
<td>102</td>
<td>111</td>
<td>18 %</td>
<td>85</td>
</tr>
<tr>
<td>Vector dot</td>
<td>45</td>
<td>53</td>
<td>10 %</td>
<td>45</td>
</tr>
<tr>
<td>Vector max</td>
<td>50</td>
<td>55</td>
<td>76 %</td>
<td>123</td>
</tr>
<tr>
<td>LMS Adaptive FIR, single</td>
<td>87</td>
<td>119</td>
<td>9 %</td>
<td>65</td>
</tr>
<tr>
<td>LMS Adaptive FIR, frame</td>
<td>3692</td>
<td>4384</td>
<td>18 %</td>
<td>2600 (***</td>
</tr>
<tr>
<td>Single FIR</td>
<td>22</td>
<td>60</td>
<td>18 %</td>
<td>22</td>
</tr>
<tr>
<td>Frame FIR</td>
<td>882</td>
<td>921</td>
<td>1 %</td>
<td>900</td>
</tr>
<tr>
<td>Complex FIR</td>
<td>3644</td>
<td>3696</td>
<td>0 %</td>
<td>3000</td>
</tr>
<tr>
<td>Blocktransfer</td>
<td>80</td>
<td>88</td>
<td>0%</td>
<td>----</td>
</tr>
<tr>
<td>Cross Correlation</td>
<td>739</td>
<td>753</td>
<td>0%</td>
<td>----</td>
</tr>
<tr>
<td>Auto Correlation</td>
<td>8</td>
<td>-----</td>
<td>0%</td>
<td>----</td>
</tr>
<tr>
<td>Wave digital filter</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>FSM</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>Floating to integer</td>
<td>14</td>
<td>17</td>
<td>30 %</td>
<td>----</td>
</tr>
<tr>
<td>Integer to floating</td>
<td>14</td>
<td>16</td>
<td>80 %</td>
<td>----</td>
</tr>
<tr>
<td>Division</td>
<td>60</td>
<td>67</td>
<td>23 %</td>
<td>----</td>
</tr>
</tbody>
</table>

(*) Approximated average kernel cycle count, from BDTI benchmarks.
(**) Result from [11], total number of cycles.
(***) BDTI performs a sample based benchmark. Their single sample result is only multiplied with 40 samples.

9.2 General Observations

It is hard to do a comparison between our processor and the ones described by BDTI. For more information and results of the BDTI-tested processors, see Appendix B. Comparisons with the BDTI column above should be done carefully, it is only an approximate guideline. Our processor is often just above the average in terms of number of instruction cycles. For all of the benchmarks there are always processors that are better and those who are worse. It is in the most advanced functions the biggest differences occur between our programs and the ones
performed by BDTI. This is probably due to, to some extent, the difference in the programmer’s skills.

The instruction cycle count presented here and the comparisons, are done without the processors clock speed. For a more reliable indicator the instruction cycle time has to be taken into account. This is because manufactures often make trade-offs between a powerful instruction set and short cycle times. But that is beyond the work of this thesis. For example, processors with caches or memory instructions that require more than one cycle to be executed, can probably be clocked at a higher rate.

The processors can be divided into two groups, depending on whether they have one or two multipliers. The number of multipliers is one of the most important differences, but processors with two multipliers also often provide other hardware enhancements. These processors are in most benchmarks much more efficient. The rest of the comparisons will be with single multiplier processors, because of the otherwise large drawback with the lack of a multiplier.

Some important factors for the cycle count are the way loops, memories and the pipeline are utilized. Some processors lack the possibility of nested hardware loops. Some of the processors have a very limited room for parallel memory fetches and for some the memory operations require more cycles than others.

9.3 FFT
This benchmark resulted in 19000 cycles. Most of the processors in the BDTI benchmark require much less cycles. The bottleneck is the memory load/store instructions in the main loop. With the present architecture, almost none of them can be executed in parallel. 50% of the instructions in the main loop are used for data transfer between registers and memory. Processors that don’t support nested hardware loops, often need much more cycles in this benchmark. Some of the processors that use caches require extra cycles. It is because the caches can’t handle the large number of instructions and instructions executed outside the cache requires multiple cycles. Processors with extra long pipelines can also loose some cycles, because of pipeline stalls.

As with the digital filters, support for parallel moves is important and would improve the performance of our processor. Some processors have more advanced loops, so two of the loops can be combined and cycles saved. This could be interesting, but it requires more control hardware than we think is necessary.

The reduction in the number of cycles is mainly due to the introduction of the butterfly instruction.

9.4 Division
The division benchmark gives a cycle count of about 66 depending on the data. BDTI doesn’t use this benchmark. Compared to the processors tested by BDTI, this processor is probably quite good. The conditional execution and the relative simple and effective hardware loops reduce the cycle count. The cycle count can be reduced even further for some inputs, if more special program code is added. We tried to keep the program general without special additions to the algorithm. The division algorithm we have used is widely used in educational areas. Other algorithms can be used in the industry, which are more data dependent.
To improve the processors division capability additional hardware support for bitwise division could be added. This would result in 16 kernel instructions, plus some instructions for sign calculations and initialization. We didn’t add the extra hardware because DSP applications seldom require explicit division. DSP-algorithms usually utilize shifts or multiplications instead of divisions.

Without the new instruction set the number of cycles was about 87. The improvement is due to the conditional execution and the bittest instruction.

9.5 Vector addition
The vector addition benchmark gives a cycle count of 111. In comparison with the BDTI benchmarks our processor has a higher cycle count than most of the other processors. The real bottleneck is the lack of an addition with two parallel memory loads or the possibility to write to memory in parallel with an arithmetic calculation. The ones with higher cycle counts require three instructions for each result. It is due to the memory restrictions it is necessary with additional instructions. Some processors, as ours, can only do one parallel load with the addition and some processors requires two cycles for memory writes. Processors that lack the functionality of hardware loops will require more cycles. Most of the double-mac processors tested by BDTI can do all of the operations in one instruction cycle.

One way to improve the processor for this benchmark would be to make it possible to execute two parallel loads, together with the addition. This requires a longer instruction word or more dedicated registers, which we have tried to avoid. Another way could be a more advanced memory structure so that both the loading and storing would be possible to execute at the same time. This would however make the processor much more expensive.

Without the new instruction set the number of cycles was 136. The improvement is mainly due to the introduction of the addadd instruction and a shorter setup phase.

9.6 Vector dot product
The vector dot product benchmark gives a cycle count of 53. This is about the same as most of the BDTI tested processors. DSP processors are specialized on this type of algorithms, so it is not much difference between the processors and it is therefore hard to improve it. The small difference is due to the setup of pointers and initialization of the inner loop. Processors with two mac units are of course much better in this benchmark.

Without the new instruction set the number of cycles was 59. The improvement is mainly due to a shorter setup phase.

9.7 Vector maximum
The vector maximum benchmark gives a cycle count of about 55, depending on the data. This is one of few benchmarks in which our processor is one of the best. The good result is due to the specialized max instruction. Processors that have powerful and flexible conditional execution features achieve lower cycle counts on this benchmark. Some of the processors have very high cycle counts. They lack conditional execution and have costly conditional
branches. The lack of multi-instruction hardware looping also makes some of the numbers even higher. Some of the better processors use a conditional dual move, to store both the value and its address.

It is hard to improve the processor even further for this benchmark. One improvement to the max instruction would be to make it possible for the programmer to decide if the stored value should be strictly greater or greater and equal. The dual move instruction could maybe be useful in some applications, but we haven’t found it necessary to add it to the instruction set. Without the new instruction set the number of cycles were about 230, depending on the data. The improvement is mainly due to the introduction of the new max instruction.

9.8 Number representation conversion

The fixed to floating point benchmark gives a cycle count of about 16, depending on the data. For the floating to fixed point benchmark the cycle count is about 17, depending on the data. BDTI has not performed this benchmark so it is hard to do comparisons. The results are probably quite good, due to the conditional execution possibility and the count functionality. The count instruction is especially useful for the fixed to floating point conversion. It helps with the sizing of the value in a fast and simple way. Very few of the processors tested by BDTI have the count and conditional execution ability. For the processors without this functionality and with costly branching, these benchmarks would require much more cycles.

Some special processors have support for both fixed and floating point representation. These processors probably have dedicated instructions for fast conversion between the two formats. There are also processors that mix both techniques with block floating point. These processors probably also have support for fast conversion. Conversions between the formats are seldom used in industry, because systems are usually designed for one or the other and because of the big difference in resolution. If an application requires large amount of data conversion, we suggest that dedicated hardware is used.

Without the new instruction set the number of cycles were about 90, depending on the data, for the fixed to floating point. Without the new instruction set the number of cycles were about 24, depending on the data, for the floating to fixed. The improvement is due to the conditional execution and the count instruction.

9.9 FSM, Finite State Machine

It isn’t meaningful to compare the actual cycle counts with other processors because we don’t have access to BDTI’s specification of the benchmark. Compared to these processors our processor probably is a bit better than average. The branch operations lies at the same level and the conditional execution can reduce the cycle count. Each branch takes three cycles, with two delay slots. This should be compared to some of the other processors that require several instructions beyond the delay slots. Other processors have three delay slots. The fastest processors only have one delay slot and extra support for immediate operands. With the conditional execution small costly branches can be eliminated. Some processors require more than one cycle for memory instructions, this adds to the computational parts of the FSM.
Some of the added functions to the processor can be useful for some special applications. For example the field instructions used for header extraction and the count instruction can be used for fast size recognition of values.

To improve this type of heavy conditional branching, more specialized branching operations could be useful. For example more sophisticated branching techniques like dynamic branch prediction could improve the performance. Other pipeline structures could also improve the performance in this benchmark. Both of these suggestions would require big changes to the processor, so we didn’t examine them more closely.

One important factor is the use of labels. It simplifies much of the programming.

### 9.10 Wave digital filters

We haven’t found any other benchmark of WDF so we can’t compare the number of cycles. The new operations improve the programs, so the processor is quite good at this type of computations. The greatest improvement is the possibility to use both adders in parallel, which easily can be utilized when the filters are directly mapped. The possibility to write the result from arithmetic operations to a third register is also useful. The instructions shortcomings, of being able to write to relatively few destination registers, can in most cases be handled. Its the same for the extra instruction after a mac operation, it can usually be avoided. Most of the single mac processors examined by BDTI, lack instructions for two parallel arithmetic operations.

WDF’s in practice often have to be quite large and the result has to be exact. If more than 16-bit integers precision is required, the program will become very big and time demanding. For example could floating point representation probably give better results, in some cases. Some processors have support for both fractional and floating point representation. 32 or 24 bits data width would also improve the processors performance, for demanding applications. These two solutions however would require large changes to the processor.

### 9.11 DCT

The DCT benchmark gives a cycle count of 874. BDTI doesn’t perform this benchmark so we can’t compare it to other processors. In reference [11], a DCT program is described and their cycle count is 3200. It should be noted that the program was intended for describing the algorithm and not a benchmark program. In that program each one-dimensional DCT is split even further which requires much more cycles and memory accesses. Compared to the processors tested by BDTI the cycle count is probably quite good. The butterfly-instruction is very useful as is the multiply with two parallel loads. The butterfly-instruction drawback, of only one parallel load, is not seen so much in this program, because the values are mostly taken from registers. It is only in the first row of butterflies with the memory fetches this are causing extra cycles. Some of the processors lack the possibility to perform more than one memory access at the time, outside single instruction hardware loops. This would be a problem in this benchmark, because in each of the sixteen one-dimensional DCT’s fifteen values have to be loaded from memory. For some of the processors the memory operations takes more than one cycle. This would also be a problem, because of many memory
operations, at least 368. For the processors without hardware looping some extra cycles are needed.

Two problems with our processor is the extra cycle after multiplication with rounding and the few register available for writing after add3/sub3 and butterfly operations. Most of the extra cycles can however be avoided by rearranging of instructions. The possibility to take offsets for memory operations from a register reduces the program size. Without this feature, either separate code for the row and column calculations would be required or more instructions would be added.

It would be possible to improve the processor for this benchmark even further, if more of the memory operations could be done in parallel with other instructions. The biggest problem is in the first stage of the DCT, when all values have to be fetched out of order from memory. If it would be possible to do memory writes in parallel with other instructions the number of cycles would also be reduced. All of these changes would require longer instruction word or dedicated registers.

Without the new instruction set the number of cycles was 1143. The improvement is much due to the butterfly-instruction, which uses the both adders in parallel.

9.12 FIR and IIR Filters

The cycle counts for the different type of FIR filters are about average compared to the processors tested by BDTI. In the frame based and the complex benchmark the cycle count is mainly depending on the loop features. Processors that lack multi-instruction or nested hardware loops, require more cycles. It is often very expensive to setup the inner loop. The loop possibilities of our processor, that supports both these features, are good. Some processors have this functionality but have higher cycle counts because of cache or addressing mode problems. In the single sample benchmark the initialization and termination instructions play a much larger role, than in the block oriented benchmarks. The housekeeping instructions are for many of the processors more than the convolution itself. It is mostly cache and addressing problems that increase the cycle counts. It is hard to improve the processor for this benchmark because it is so highly optimized for this type of tasks from the beginning. One weak point in the frame based benchmark is the delay line. It requires three cycles for each sample, which adds up to 120 cycles. Somehow it should be possible to shrink that to 40 or 80 cycles.

Our single sample adaptive FIR benchmark is not good. It is most likely that we as programmers don’t utilize the full capabilities of the processor. The processor has most of the features to be at least around average. The good support for hardware loops and memory setup would be advantageous. Our and BDTI’s adaptive FIR benchmarks of frames are not compatible. This is because we use frame based inputs of 40 samples and the presented BDTI results only are direct scaling of single sample inputs. The frame based method is more used in industry. The frame based adaptive FIR filter benchmark of our processor would probably otherwise be better than average. The good support for nested hardware loops and memory setup would be even more advantageous of this benchmark than for the single sample. The speedup of almost 20 % is due to the new possibility to do parallel loads with no offset, instead of only with post increment. Some of the best single mac processors of this benchmark have many general purpose registers, a flexible instruction set and extensive support for parallel data moves. They can compute the FIR coefficients in parallel with the
coefficient updates, so one way to improve the processor would be to extend the possibilities for parallel moves. This would however probably require a longer instruction word.

The IIR benchmark can’t be compared on a fair basis. BDTI only performs a single sample test and we use a frame based technique, which is used more in industry. Our processor as in the adaptive FIR benchmark should have the possibility to be somewhat better than average. The nested hardware loops and the memory load features are important in this benchmark. Processors that lack them need extra cycles. There are processors with more extended support for parallel moves, which are useful in this benchmark. This makes it possible to store values in parallel with the biquad computations. So to improve the processor as for the adaptive filter, better support for parallel moves would be necessary.

Our processor has 16-bit native data length. For some filters this is not enough, for example it is not uncommon that precisions of 24 bits is required. 24 or 32-bits precision can be accomplished, but this would require much more cycles for each benchmark. The processors using floating-point or larger native data word size can in some cases be better than the benchmark shows. Some processors have additional support for scaling steps, especially for multiplications. This would maybe be a way to improve the processor and make it more general. But it would require many changes to the mac hardware so we haven’t implemented it.

9.13 Auto- and Crosscorrelation
The Auto- and crosscorrelation have not been measured by BDTI, and a comparison is difficult. In the autocorrelation algorithm, we choose to only look at the kernel part and focus on the execution of the 16*32 bit multiplication. It takes 5 instructions to execute the 16*32 bit multiplication. To decrease this number a larger native data width and a MAC – unit with higher precision is needed, but we think that is unnecessary. When it comes to the crosscorrelation, its structure is quite similar to the convolution based algorithms, and the same arguments can be applied here as for the FIR- and IIR filters treated above.
Chapter 10  Conclusions

10.1 Results
The results from the benchmarks show that our processor is at the same level as the ones tested by BDTI. Probably would a more experienced programmer be able to reduce the cycle count, especially for some of the more complex benchmarks. We haven’t had access to BDTI’s specifications so the programs are not guarantied to fulfill their requirements. We have followed their guidelines and due to the simple structure of the kernels, we think the programs would be approved. Benchmark results publicized by processor manufactures are often very aggressively optimized and ill-defined. We have worked with reasonable assumptions and restrictions that BDTI requires.

The DSP-processor platform this project started out with, included basic DSP-features and hence support for executing common, basic DSP-algorithms. Therefore the most basic algorithms, such as filtering, are hard to speed-up unless big changes in the structure of the hardware are done. Most of the instructions we introduced utilizes the hardware in a more efficient way in the view of the algorithms we investigated.

10.2 Considered improvements
During the work with this thesis, more or less reasonable improvements to the processor have been considered. The processors architecture is basic and has about the same features as the commercial processors in BDTI’s tests. The implemented and considered changes, would give the processor advantages against many of the other. The changes that we have made to the processor are described in the previous chapters. Changes that could improve the processor that are more application specific are described together with the results in the previous chapter. Below are some more general improvements that are not implemented, described. The most common reasons for changes not to be implemented are that they would require much extra hardware or large changes to the existing hardware.

The improvement that would have the biggest impact would be to add an additional mac unit. This would of course require much extra hardware but the speedup would be significant.

The instruction set is still rather simple and there is a lot of space left for additions. The most effective but also maybe the most expensive would be to increase the word length. A VLIW structure would make it possible for more parallel executions and data moves. The biggest impact would be more support for parallel data moves. In many of the benchmarks, parallel writes to the memory could be very useful. VLIW would also make it possible to use more of the computational units in parallel. Another way to increase the number of parallel processes would be to use dedicated registers.

In the benchmarks the following code section is quite common.

mac GPRx GPRy ACCx
(nop)
It requires four cycles. It occurs often in inner loop sections, so even a small improvement has a big impact on the final result. If the program section could be reduced to the instruction below, it would save many cycles in the application. The improvements would also be useful individually and in other applications.

mac GPRx GPRy ACCx rnd sat AP++
(nop)

To achieve this not too much has to be done, but we didn’t have the time. The mac operation already requires two instruction cycles and with some extra hardware, the round functionality could be achieved (the saturation is already possible). Some additional logic gates, in or after the adder, could function as a rounding unit. If the processor and memory would be able to handle direct writes from computation units to the memory, several unnecessary cycles could be saved.

Some of applications may require values in other ranges than possible now on the processor. For example some IIR filters have coefficients in the range –2 to 2. This is not a problem for floating-point processors, since they automatically can handle a large range of numbers. For fixed-point processors this can however cause difficulties. Block floating point or extra scaling support could be two realistic ways to solve this problem.

Aside from the processor, more advanced memories could improve the processors performance. Some memories allow more than one access per cycle. For example there are memories that allow blocks of data to be read at the same time. The bus-structure of the processor has to be adapted, but it wouldn’t be too difficult. The memories are probably also one of the bottlenecks for the clock speed of the processor.

10.3 Future

The benchmarks we have used have only been concentrating on the number of instruction cycles for some special application kernels. The number of cycles is one factor that can be important for the performance of the processor. But other factors are also important, for example execution time, energy consumption and memory use. Algorithm kernels also miss important parts of the whole application, that are vital to the performance. For example interrupt latency, exception handling and time outs.

The I/O functionality has not been tested by any benchmark. This is mainly due to the fact that no I/O is supported by the processor yet. The processor is supposed to be used as a platform for hardware accelerators. The extra hardware will change the performance of the DSP-processor.

Some of the suggested, but not implemented improvements to the processor, should be possible to implement if more time was available.
10.4 Personal

The project has run without any bigger difficulties, and we believe we have gained insight in the area of signal processing and different techniques for efficient implementation of DSP-algorithms on DSP-processors.
Chapter 11 Bibliography


This chapter has complete descriptions of all instructions of the processor. This includes:

**Type of instruction** - Instruction group, short description.

**Syntax** - What the assembly code looks like, addressing modes.

**Operands** - What data the instruction can use.

**Execution** - What the instruction does (“mathematically”)

**Description** - Description of the behaviour of the instruction, and which registers and statusflags are affected.

**Example** - A short example of use of the instruction.

In the end the use of conditional execution and labels are described.
abs

Type of instruction
Arithmetic instruction – absolute value

Syntax
Register direct:  

abs GPRx GPRy

Operands
GPRx: GPR0 – GPR31
GPRy: GPR0 – GPR31

Execution

|GPRx|  → GPRy

Description
The absolute value of register GPRx is stored in register GPRy.
The N flag is set according the GPRx register. The other flags are not updated.

Example

abs GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatusReg</td>
<td>b0000</td>
<td>b1000</td>
</tr>
<tr>
<td>GPRx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h00EE</td>
</tr>
</tbody>
</table>
add

**Type of instruction**
Arithmetic instruction - addition.

**Syntax**
Register direct: add GPRx GPRy [c]

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

**Execution**
GPRx + GPRy → GPRy

**Description**
The values in register GPRx and GPRy are added and the result is stored in register GPRy. The c bit is set to use addition with carry in. The flags N, Z, C and O are updated. C is set when unsigned addition generates carry. O is set when signed addition generates overflow.

**Example**
add GPRx, GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0010</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h0032</td>
</tr>
</tbody>
</table>
add3

Type of instruction
Arithmetic instruction - addition

Syntax
Register direct: add3 GPRx GPRy GPRz [c]

Operands
GPRx: GPR0- GPR31
GPRy: GPR0- GPR31
GPRz: GPR16- GPR23

Execution
GPRx + GPRy → GPRz

Description
The values in register GPRx and GPRy are added and the result is stored in GPRz. The c bit is set to use addition with carry in.
The flags N, Z, C and O are updated. C is set when unsigned addition generates carry. O is set when signed addition generates overflow.

Example
add2 GPRx GPRy GPRz

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0010</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h0020</td>
</tr>
<tr>
<td>GPRz</td>
<td>hxxxx</td>
<td>h0032</td>
</tr>
</tbody>
</table>
addadd

**Type of instruction**
Arithmetic instruction – two additions

**Syntax**
Register direct without carry: addadd GPRx GPRy GPRz GPRw ACCx

**Operands**
GPRx: GPR0-GPR31
GPRy: GPR16-GPR23
GPRz: GPR16-GPR31
GPRw: GPR16- GPR23
ACCx: ACC0-ACC1

**Execution**
GPRx + GPRy → ACCx [31:16]
GPRz + GPRw → ACCx [15:0]

**Description**
The values in register GPRx and GPRy are added and the result is stored in higher part of ACCx. At the same time the values in register GPRz and GPRw are added and the result is stored in lower part of ACCx.
The flags N, Z, and C are updated according to GPRx + GPRy. C is set when unsigned addition generates carry. O is set when any of the signed operations generates overflow.

**Example**
addadd GPRx GPRy GPRz GPRw ACCx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0010</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h0020</td>
</tr>
<tr>
<td>GPRz</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GPRw</td>
<td>h0004</td>
<td>h0004</td>
</tr>
<tr>
<td>ACCx (low)</td>
<td>hxxxx</td>
<td>h0006</td>
</tr>
<tr>
<td>ACCx (high)</td>
<td>hxxxxxx</td>
<td>h000032</td>
</tr>
</tbody>
</table>
addi

Type of instruction
Arithmetic instruction - addition with immediate data

Syntax
Immediate data without carry: addi #Data GPRy

Operands
h8000 ≤ Data ≤ h7FFF
GPRy: GPR0 - GPR31

Execution
Data + GPRy → GPRy

Description
The value in register GPRy and the Data value are added. The result is stored in register GPRy.
The flags N, Z, C and O are updated. C is set when unsigned addition generates carry. O is set when signed addition generates overflow.

Example
addi #h1234 GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0020</td>
<td>h1254</td>
</tr>
</tbody>
</table>
**and**

**Type of instruction**
Logic instruction - bitwise and.

**Syntax**
Register direct: and GPRx GPRy

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

**Execution**
GPRx & GPRy → GPRy

**Description**
Bitwise and between the values in register GPRx and GPRy. The result is stored in register GPRy.
The flags N and Z are updated.

**Example**
and GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1010</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h8010</td>
<td>h0010</td>
</tr>
</tbody>
</table>
**andi**

**Type of instruction**
Logic instruction - bitwise and with immediate data.

**Syntax**
Immediate data: \( \text{andi} \ #\text{Data} \ GPRy \)

**Operands**
h8000 ≤ Data ≤ h7FFF
GPRy: GPR0 - GPR31

**Execution**
Data & GPRy → GPRy

**Description**
Bitwise and between the value in register GPRy and the Data. The result is stored in register GPRy.
The flags N and Z are updated.

**Example**
\( \text{andi} \ #hFF \ GPRy \)

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRy</td>
<td>h8020</td>
<td>h0020</td>
</tr>
</tbody>
</table>
asl

Type of instruction
Shift instruction - arithmetic shift.

Syntax
Register direct: asl GPRx GPRy [c]

Operands
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

Execution
GPRy >> (GPRx & h001F) → GPRy

Description
If the value in GPRx is positive the value in register GPRy is shifted GPRx steps to the left. If the value in GPRx is negative the value in GPRy is arithmetically shifted -GPRx steps to the right. The result is stored in register GPRy. The c bit is set to use the carry bit.
The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

Example
asl GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0001</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0013</td>
<td>h0013</td>
</tr>
<tr>
<td>GPRy</td>
<td>h9F22</td>
<td>hFFFC</td>
</tr>
</tbody>
</table>
asli

Type of instruction
Shift instruction - arithmetic shift with immediate data.

Syntax
Immediate data: asli #Step GPRy [c]

Operand
-15 ≤ Step ≤ 15
GPRy: GPR0 - GPR31

Execution
GPRy >> Step → GPRy

Description
If the value Step is positive the value in register GPRy is shifted GPRx steps to the left. If the value in GPRx is negative the value in GPRy is arithmetically shifted -GPRx steps to the right. The result is stored in register GPRy. The c bit is set to use the carry in. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

Example
asli #-4 GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b1000</td>
</tr>
<tr>
<td>GPRy</td>
<td>hFF22</td>
<td>HFFF2</td>
</tr>
</tbody>
</table>
avg

**Type of instruction**
Arithmetic instruction - average value.

**Syntax**
Register direct: avg GPRx GPRy

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

**Execution**
(GPRx + GPRy) / 2 → GPRy

**Description**
The average value of the value in register GPRx and in register GPRy is stored in register GPRy.
The flags N and Z are updated.

**Example**
avg GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0023</td>
<td>h0023</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h0021</td>
</tr>
</tbody>
</table>
bittest

Type of instruction
Logic instruction – bitwise and with immediate data.

Syntax
Immediate data: bittest #Data GPRy

Operands
h8000 ≤ Data ≤ h7FFF
GPRy: GPR0 – GPR31

Execution
data & GPRy → None

Description
Bitwise and, between the value in registers GPRy and the “data” value. The result is not stored.
The N and Z flags are updated.

Example
bittest GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1010</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0010</td>
<td>h0010</td>
</tr>
</tbody>
</table>
**bra{cond}[r]**

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

**Syntax**
Register direct/relative: \( bra{cond}[r] \) GPRx
Immediate PC address/relative: \( bra{cond}[r] \) #Addr

**Operands**
h8000 \( \leq \) Addr \( \leq \) h7FFF
GPRx: GPR0 - GPR31

**Execution**
if \{cond\} is TRUE
  GPRx → PC
  or Addr → PC
else
  PC+1 → PC

**Description**
A conditional branch jump. Either register or constant based. The jump is delayed two cycles, that is the two instructions following the branch instruction are executed either the branch is taken or not. None of the two following instructions may be bra, call, rts, loop or repeat instructions. Bra may not be used as a repeat instruction or as one of the two last instructions in a hardware loop. To use relative jumps one “r” is added to the instruction word, without space. Then the jump value is added to the present PC address.
No flags are updated.

<table>
<thead>
<tr>
<th>{cond}</th>
<th>Relation</th>
<th>Flag status</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt</td>
<td>GPRx &gt; GPRy</td>
<td>Z=0 and N=0</td>
</tr>
<tr>
<td>ge</td>
<td>GPRx &gt;= GPRy</td>
<td>N=0</td>
</tr>
<tr>
<td>lt</td>
<td>GPRx &lt; GPRy</td>
<td>N=1</td>
</tr>
<tr>
<td>le</td>
<td>GPRx &lt;= GPRy</td>
<td>Z=1 or N=0</td>
</tr>
<tr>
<td>eq</td>
<td>GPRx = GPRy</td>
<td>Z=1</td>
</tr>
<tr>
<td>ne</td>
<td>GPRx /= GPRy</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>carry</td>
<td>C=1</td>
</tr>
<tr>
<td>nc</td>
<td>not carry</td>
<td>C=0</td>
</tr>
<tr>
<td>o</td>
<td>overflow</td>
<td>O=1</td>
</tr>
<tr>
<td>no</td>
<td>not overflow</td>
<td>O=0</td>
</tr>
</tbody>
</table>

**Example**
bragtr #h30

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h0100</td>
<td>h0130</td>
</tr>
</tbody>
</table>
butterfly

Type of instruction
Arithmetic instruction – butterfly addition and subtraction.

Syntax
Register direct without carry: butterfly GPRx GPRy GPRz ACCw

Operands
GPRx: GPR0-GPR31
GPRy: GPR0-GPR16
GPRz: GPR16-GPR23
ACCx: ACC0-ACC1

Execution
GPRx + GPRy \rightarrow ACCw [31:16]
GPRy – GPRx \rightarrow GPRz

Description
The values in register GPRx and GPRy are added and the result is stored in lower part of ACCx. At the same time the values in register GPRx is subtracted from the one in GPRy and the result is stored in GPRz.
The flags N, Z, and C are updated according to GPRx + GPRy. C is set when unsigned addition generates carry. O is set if any of the signed operations generates overflow.

Example
butterfly GPRx GPRy GPRz ACCw

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0010</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h0020</td>
</tr>
<tr>
<td>GPRz</td>
<td>hxxxx</td>
<td>h000E</td>
</tr>
<tr>
<td>ACCw (low)</td>
<td>hxxxx</td>
<td>h0032</td>
</tr>
</tbody>
</table>
call[r]

**Type of instruction**
Program flow instruction - subroutine jump

**Syntax**
Immediate PC address/relative: call[r] #Addr
Register direct/relative: call[r] GPRx

**Operands**
h8000 ≤ Addr ≤ h7FFF
GPRx: GPR0 – GPR31

**Execution**
PC → PC stack
Addr → PC

**Description**
A subroutine jump

The jump is delayed two cycles, that is the two instructions following the call instruction are executed before the jump is taken. None of the two following instructions may be bra, call, rts, loop or repeat instructions. Call may not be used as a repeat instruction or as one of the two last instructions in a hardware loop. To use relative jumps one “r” is added to the instruction word, without space. Then the jump value is added to the present PC address. No flags are updated.

**Example**
call #h3000

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h0012</td>
<td>h3000</td>
</tr>
</tbody>
</table>
**clracc**

**Type of instruction**
MAC instruction - clear accumulator

**Syntax**
Register direct: `clracc ACCx`

**Operands**
ACCx: ACC0, ACC1

**Execution**
0 → ACCx

**Description**
Clear accumulator register.
No flags are updated.

**Example**
clracc ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC0</td>
<td>hxxxxxxxxxx</td>
<td>h000000000</td>
</tr>
<tr>
<td>xx</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**comp**

**Type of instruction**
Arithmetic instruction - compare two values.

**Syntax**
Register direct: \texttt{comp GPRx GPRy}

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

**Execution**
GPRy – GPRx → None

**Description**
The value in register GPRx is subtracted from the value in register GPRy, but the result is not stored.
The flags are updated. C is set when unsigned subtraction does not generate borrow. O is set when signed subtraction generates overflow.

**Example**
\texttt{comp GPRx GPRy}

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0100</td>
</tr>
<tr>
<td>GPRx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>GPRy</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
</tbody>
</table>
count

Type of instruction
Arithmetic instruction – count leading ones or zeros.

Syntax
Register direct: count GPRx GPRy [z]

Operands
GPRx: GPR0 – GPR31
GPRx: GPR0 – GPR31

Execution
#1/0 (GPRx) → GPRy

Description
Counts the number of leading ones/zeros, depending on the z flag, in GPRx and stores the result in GPRy. The flags are not updated.

Example
count GPRx GPRy z

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0F12</td>
<td>h0F12</td>
</tr>
<tr>
<td>GPRy</td>
<td>hxxxx</td>
<td>h0004</td>
</tr>
</tbody>
</table>
**getfield**

**Type of instruction**
Logic instruction – get bit-field from register

**Syntax**
Register direct:  
getfield GPRx GPRy GPRz GPRw

**Operands**
GPRx: GPR0 – GPR31  
GPRy: GPR16 – GPR23  
GPRw: GPR16 – GPR23  
GPRz: GPR0 – GPR31

**Execution**
GPRx(GPRy:GPRz) → GPRw

**Description**
The instruction copies a bitfield from GPRx to GPRw. GPRy gives the fields starting point in GPRx and GPRz its length.
No flags are updated.

**Example**
getfield GPRx GPRy GPRz GPRw

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h2345</td>
<td>h2345</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0004</td>
<td>h0004</td>
</tr>
<tr>
<td>GPRz</td>
<td>h0008</td>
<td>h0008</td>
</tr>
<tr>
<td>GPRw</td>
<td>hxxxx</td>
<td>h0034</td>
</tr>
</tbody>
</table>
**jmp[r]**

**Type of instruction**
Program flow instruction – jump.

**Syntax**
Register direct/relative: \( \text{jmp[r]} \) GPRx
Immediate PC address/relative : \( \text{jmp[r]} \) #Addr

**Operands**
\( h0000 \leq \text{Addr} \leq hFFFF \)

**Execution**
\( \text{Addr} \rightarrow \text{PC} \)

**Description**
Unconditional jump.

The jump is delayed two cycles, that is the two instructions following the jmp instruction are executed before the jump is taken. None of the two following instructions may be bra, call, rts, loop or repeat instructions. Jmp may not be used as a repeat instruction or as one of the two last instructions in a hardware loop. To use relative jumps one “r” is added to the instruction word, without space. Then the jump value is added to the present PC address. No flags are updated.

**Example**
\( \text{jmp \#h3000} \)

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h0012</td>
<td>h3000</td>
</tr>
</tbody>
</table>
**load**

**Type of instruction**
Data move instruction. Load register with immediate data.

**Syntax**
Immediate data: \texttt{load \#Const GPRy}

**Operands**
h8000 ≤ Const ≤ h7FFF  
GPRy: GPR0 - GPR31

**Execution**
Const → GPRy

**Description**
The constant,(Const), is loaded into the register GPRy.  
No flags are updated.

**Example**
\texttt{load \#h2034 GPRy}

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRy</td>
<td>h1210</td>
<td>h2034</td>
</tr>
</tbody>
</table>
**loadacc**

**Type of instruction**
Data move instruction - load acc with sign extension.

**Syntax**
Register direct  
loadacc GPRx ACCx [L]

**Operands**
GPRx: GPR0- GPR31
ACCx: ACC0-ACC1

**Execution**
GPRx → ACCx

**Description**
The value in register GPRx are loaded to ACCx, with sign extension. If the processor is set in integer mode, the value is loaded to the 16 least significant bits. The higher register is extended with sign-bits if the L flag isn’t set. If the L-flag is set the higher register keeps its value. If the processor is in fractional mode the value is loaded to the higher bits [16:31] and the guard-bits are sign-extended. The lower register in this case are set to zero or keeps its value, depending on the L-flag.

**Example**
loadacc GPRx ACCx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0001</td>
<td>h0001</td>
</tr>
<tr>
<td>ACCx (low)</td>
<td>hxxxx</td>
<td>h0000</td>
</tr>
<tr>
<td>ACCx (high)</td>
<td>hxxxxx</td>
<td>h00001</td>
</tr>
</tbody>
</table>
loadtm, loaddm

Type of instruction
Memory instruction- load register from memory

Syntax
Register indirect:       loadXm APy GPRy
Register indirect with post increment: loadXm APx++ GPRy
Register indirect with offset address: loadXm APx #Offset GPRy
Register indirect with variable offset: loadXm APx APz GPRy

Operands
h0 ≤ Offset ≤ hFF
APx: AP0- AP8
GPRy: GPR0-GPR31

Execution
DM(#Addr) → GPRy

Description
The value stored at the address APx or APx + offset in the specified memory is copied to
register GPRy. If post increment addressing is used APx is increased with the step value
corresponding to the register.
The flags N and Z are updated.

Example
loaddm APx++ GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>APx</td>
<td>h0200</td>
<td>h0201</td>
</tr>
<tr>
<td>GPRy</td>
<td>hxxxx</td>
<td>h1234</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
</tbody>
</table>
loaddmi

Type of instruction
Memory instruction - Load register, immediate address.

Syntax
Immediate address:    loaddmi #Addr GPy

Operands
h0 ≤ Addr ≤ hFFFF
GPy: GPR0 – GPR31

Execution
DM(#Addr) → GPy

Description
The value stored at the address #Addr in dm is copied to the register GPrx. The flags are not updated.

Example
loaddmi #hFF00 GPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPy</td>
<td>hxxxx</td>
<td>h1234</td>
</tr>
<tr>
<td>DM(hFF00)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
</tbody>
</table>
**loop[r]**

**Type of instruction**
Program flow instruction - hardware loop.

**Syntax**
- Register direct/relative: `loop[r] GPRx`
- Immediate PC address/relative: `loop[r] #Addr`

**Operands**
- `h0 ≤ Addr ≤ hFFFF`
- `GPRx: GPR0 – GPR31`
- `LOOP register`

**Execution**
- `PC + 1 → Loopstartstack`
- `Addr → Loopendstack`
- `LOOPregister → Loopcounterstack`

**Description**
The instructions between the loop instruction and the PC adress `Addr` (including that address) is repeated a number of times specified by the value in the `LOOP` register. Up to four nested loops are possible, however two loops may never end at the same adress. The loop must have at least two instructions (otherwise repeat is used) and the last two instructions in a loop must not be `jmp`, `bra`, `call` or `repeat`. To use a relative loop address, one “r” is added to the instruction word, without space. Then the length of the loop will be the number of instructions specified.

**Example**
- `loadi #30 LOOP`
- `loop #2000`

*Instructions from program addresses PC+1 to 2000 will be looped 30 times*
**Ishl**

**Type of instruction**
MAC instruction - 40-bit shift.

**Syntax**
Register direct:  
Ishl GPRx ACCy [c]

**Operands**
GPRx: GPR0 - GPR31  
ACCy: ACC0, ACC1

**Execution**

Accy << (GPRx & h3F) → ACCy

**Description**
If the value in GPRx is positive the value in accumulator GPRy is shifted GPRx steps to the left. If the value in GPRx is negative the value in ACCy is arithmetically shifted -GPRx steps to the right. The result is stored in register ACCy. The c bit is set to use the carry bit.
The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
Ishl GPRx ACCy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0008</td>
<td>h0008</td>
</tr>
<tr>
<td>ACCy</td>
<td>h000000FF2</td>
<td>h00000FF22</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>00</td>
</tr>
</tbody>
</table>
Ishli

Type of instruction
MAC instruction. 40-bit shift with immediate data

Syntax
Immediate data: Ishli #Steps ACCy [c]

Operands
-32 ≤ Steps ≤ 32
ACCy: ACC0, ACC1

Execution
ACCy << Steps → ACCy

Description
If the value Steps is positive the value in accumulator GPRy is shifted Steps steps to the left. If Steps is negative the value in ACCy is arithmetically shifted –Steps steps to the right. The result is stored in register ACCy. The c bit is set to use the carry bit. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

Example
Ishl #12 ACCy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCy</td>
<td>h000000FF2</td>
<td>h000FF220</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>00</td>
</tr>
</tbody>
</table>
**lsl**

**Type of instruction**
Shift instruction - logical shift.

**Syntax**
Register direct: \( lsl \ GPRx \ GPRy \ [c] \)

**Operands**
GPRx: GPR0 - GPR31  
GPRy: GPR0 - GPR31

**Execution**
GPRy \( \ll (GPRx \ & \ h1F) \rightarrow GPRy \)

**Description**
If the value in GPRx is positive the value in register GPRy is shifted GPRx steps to the left. If the value in GPRx is negative the value in GPRy is logically shifted \(-GPRx\) steps to the right. The result is stored in register GPRy. The c bit is set to use the carry bit. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
lsl GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRx</td>
<td>hFFFE</td>
<td>hFFFE</td>
</tr>
<tr>
<td>GPRy</td>
<td>hFF22</td>
<td>h3FC8</td>
</tr>
</tbody>
</table>
**lsli**

**Type of instruction**
Shift instruction - logical shift with immediate data.

**Syntax**
Register direct: `lsli #Step GPRy [c]`

**Operands**
-15 ≤ Step ≤ 15
GPRy: GPR0 - GPR31

**Execution**
GPRy >> Step → GPRy

**Description**
If the value Step is positive the value in register GPRy is shifted Step steps to the left. If the value in GPRx is negative the value in GPRy is logically shifted –Step steps to the right. The result is stored in register GPRy. The c bit is set to use the carry bit. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
`lsli #4 GPRy`

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b1000</td>
</tr>
<tr>
<td>GPRx</td>
<td>hFF22</td>
<td>hF220</td>
</tr>
</tbody>
</table>
**mac, macu, macus, macs, macsub, macsubu, macsubus, macsubsu**

**Type of instruction**
Mac instruction - multiply and accumulate.

**Syntax**
Register direct: \text{mac[sub][u/su/us]} GPRx GPRy ACCz

[SAT]

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR16 - GPR31
ACCz: ACC0, ACC1

**Execution**
ACCz ± GPRx * GPRy → ACCz

**Description**
The value of register GPRx is multiplied by the value of register GPRy and the product is added to (mac, macu, macus, macsu) or subtracted from (macsub, macsubu, macsubus, macsubsu) the accumulator ACCz. mac and macsub executes a signed multiplication and macu and macsubu an unsigned multiplication. macus/macsu and macsubus/macsubsu considers the first or the second operand to be unsigned respectively. If SAT is added the accumulator will be saturated after the accumulation. The status flags N, Z and O are updated.

**Example**
mac GPRx GPRy ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0003</td>
<td>h0003</td>
</tr>
<tr>
<td>ACC0</td>
<td>h0000001000</td>
<td>h000000100 6</td>
</tr>
</tbody>
</table>
**max**

**Type of instruction**
Arithmetic instruction – maximum and address search with parallel load.

**Syntax**
Register direct with postincrement: \( \text{max} \text{ GPR}x \text{ AP}x++ \text{ GPR}y \text{ AP}y \)

**Operands**
- **GPRx**: GPR0 – GPR31
- **APx**: AP0 – AP7
- **GPRy**: GPR0 – GPR31
- **APy**: AP0 – AP7

**Execution**
if \( \text{GPR}x > \text{GPR}y \) then
- \( \text{GPR}y \leq \text{GPR}x \)
- \( \text{AP}y \leq \text{AP}x \)

**Description**
The instruction compares the values from the two GP registers. If the value in \( \text{GPR}x \) is larger than the \( \text{GPR}y \) value, the \( \text{GPR}x \) values from \( \text{GPR}x \) and \( \text{AP}x \) are copied to \( \text{GPR}y \) and \( \text{AP}y \). There are also a parallel memory load with postincrement of the address pointer.

**Example**
\( \text{max} \text{ GPR}x \text{ AP}x++ \text{ GPR}y \text{ AP}y \)

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0020</td>
<td>h0011</td>
</tr>
<tr>
<td>APx</td>
<td>h0200</td>
<td>h0201</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0009</td>
<td>h0020</td>
</tr>
<tr>
<td>APy</td>
<td>h0013</td>
<td>h0200</td>
</tr>
<tr>
<td>MD(h0200)</td>
<td>h0011</td>
<td>h0011</td>
</tr>
</tbody>
</table>
min

Type of instruction
Arithmetic instruction – minimum and address search with parallel load

Syntax
Register direct with postincrement: \[ \text{min GPRx APx++ GPRy APy} \]

Operands
- GPRx: GPR0 – GPR31
- APx: AP0 – AP7
- GPRy: GPR0 – GPR31
- APy: AP0 – AP7

Execution
if GPRx < GPRy then
  GPRy<=GPRx
  APy<=APx

Description
The instruction compare the values from the two GP registers. If the value in GPRx is smaller than the GPRy value, the GPRx values from GPRx and APx are copied to GPRy and APy. There are also a parallel memory load with postincrement of the address pointer.

Example
max GPRx APx++ GPRy APy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0002</td>
<td>h0011</td>
</tr>
<tr>
<td>APx</td>
<td>h0200</td>
<td>h0201</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0009</td>
<td>h0002</td>
</tr>
<tr>
<td>APy</td>
<td>h0013</td>
<td>h0200</td>
</tr>
<tr>
<td>MD(h0200)</td>
<td>h0011</td>
<td>h0011</td>
</tr>
</tbody>
</table>
move

Type of instruction
Data move instruction - move between registers.

Syntax
Register direct: move GPRx GPRy

Operands
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

Execution
GPRx → GPRy

Description
The value of register GPRx is copied to register GPRy. No flags are updated.

Example
move GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>GPRy</td>
<td>h1010</td>
<td>hFF12</td>
</tr>
</tbody>
</table>
**move2tm, move2dm**

**Type of instruction**
Memory instruction - write to memory.

**Syntax**
- Register indirect with postincrement:  `move2{tm/dm} APx++ GPRx`
- Register indirect with offset address:  `move2{tm/dm}APx #Offset GPRx`

**Operands**
- `h0 ≤ Offset ≤ hFF`
- `APx: AP0 - AP7`
- `GPRx: GPR0 - GPR31`

**Execution**
`GPRx → DM(APx + Offset)`

**Description**
The value of register `GPRx` is copied to the specified memory address. No flags are updated.

**Example**
`move2dm APx++ GPRx`

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>APx</td>
<td>h0200</td>
<td>h0201</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>hxxxx</td>
<td>hFF12</td>
</tr>
</tbody>
</table>
move2dmd

Type of instruction
Memory instruction – write to memory immediately data

Syntax
Register indirect: move2dmd #Data GPRx

Operands
h8000 ≤ Data ≤ h7FFF
GPRx: GPR0 – GPR31

Execution
#Data → DM(Addr)

Description
The data value is written to the memory address pointed out by the address pointer.

Example
move2dmd #h1234 GPRx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0200</td>
<td>h0200</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>hxxxx</td>
<td>h1234</td>
</tr>
</tbody>
</table>
move2dmd+

Type of instruction
Memory instruction – write to memory immediately data post increment.

Syntax
Register indirect with post increment: move2dmd+ #Data APx

Operands
h8000 ≤ Data ≤ h7FFF
APx: AP0 – AP7

Execution
#Data → DM(Addr)

Description
The data value is written to the memory address pointed out by the address pointer and it is updated with post increment.

Example
move2dmd+ #h1234 APx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>APx</td>
<td>h0200</td>
<td>h0201</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>hxxxx</td>
<td>h1234</td>
</tr>
</tbody>
</table>
move2tmd

**Type of instruction**
Memory instruction – write to memory immediately data

**Syntax**
Immediate data: move2tmd #Data #Addr

**Operands**
h8000 ≤ Data ≤ h7FFF
h0 ≤ Addr ≤ h7F

**Execution**
Data → DM(CM(Addr))

**Description**
The data value is written to the memory position pointed out by Addr in the Control memory.
The control memory consists of the last 128 addresses in the TM.

**Example**
move2tmd #h4332 #65

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM(hFF00+65)</td>
<td>hxxxx</td>
<td>h4332</td>
</tr>
</tbody>
</table>
move2dmi

Type of instruction
Memory instruction - write to memory, immediate address.

Syntax
Adress direct:                          move2dmi #Addr GPRx

Operands
h0 ≤ Addr ≤ hFFFF
GPRy: GPR0 – GPR31

Execution
GPRx → DM(#Addr)

Description
The value stored in register GPRx is copied to the address Addr in dm.
The flags are not updated.

Example
move2dmi #hFF00 GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>DM(hFF00)</td>
<td>hxxxx</td>
<td>hFF12</td>
</tr>
</tbody>
</table>
**mpy, mpyu, mpyus, mpysu**

**Type of instruction**
Mac instruction – multiplication.

**Syntax**
Register direct: mpy[u/su/us] GPRx GPRy ACCz
[SAT/RND]

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR16 - GPR31
ACCz: ACC0, ACC1

**Execution**
GPRx * GPRy → ACCz

**Description**
The value of register GPRx is multiplied by the value of register GPRy and the product is placed in the accumulator ACCz. mpy executes a signed multiplication and mpyu an unsigned multiplication. mpyus/mpysu considers the first or the second operand to be unsigned respectively. If SAT is added the result will be saturated and if RND is added the result will be rounded.
The status flags N and Z are updated.

**Example**
mpy GPRx GPRy ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0003</td>
<td>h0003</td>
</tr>
<tr>
<td>ACC0</td>
<td>hxxxxxxxx</td>
<td>h00000000</td>
</tr>
<tr>
<td></td>
<td>xx</td>
<td>6</td>
</tr>
</tbody>
</table>
neg

Type of instruction
Arithmetic instruction - negate value.

Syntax
Register direct: neg GPRx GPRy

Operands
GPRx: GPR0 - GPR31
GPRy: GPR16 - GPR31

Execution
-GPRx → GPRy

Description
The value in register GPRx is negated and stored in register GPRy.
The flags are not updated.

Example
neg GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>hxxxx</td>
<td>hFFEE</td>
</tr>
</tbody>
</table>
**nop**

**Type of instruction**
Program flow instruction - no operation.

**Syntax**
No operands:                      nop

**Operands**

**Execution**
PC + 1 → PC

**Description**
This instruction only affects the PC and is used to create execution delays.

**Example**
Nop

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h8020</td>
<td>h8021</td>
</tr>
</tbody>
</table>
not

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

**Syntax**
Register direct: not GPRx GPRy

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

**Execution**
inv(GPRx) → GPRy

**Description**
The value in register GPRx is inverted bitwise and stored in register GPRy. The flags are not updated.

**Example**
not GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>hxxxx</td>
<td>hFFED</td>
</tr>
</tbody>
</table>
**OR**

**Type of instruction**
Logic instruction - bitwise or.

**Syntax**
Register direct: or GPRx GPRy

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

**Execution**
GPRx | GPRy → GPRy

**Description**
Bitwise or between the values in register GPRx and GPRy. The result is stored in register GPRy.
The flags N and Z are updated.

**Example**
or GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b1000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h8020</td>
<td>h8032</td>
</tr>
</tbody>
</table>
ori

Type of instruction
Logic instruction - bitwise or with immediate data.

Syntax
Immediate data: ori #Data GPRy

Operands
h8000 ≤ Data ≤ h7FFF
GPRy: GPR0 - GPR31

Execution
Data | GPRy → GPRy

Description
Bitwise or between the value in register GPRy and the value Data. The result is stored in register GPRy.
The flags N and Z are updated.

Example
ori #h1111 GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRy</td>
<td>h8020</td>
<td>h9131</td>
</tr>
</tbody>
</table>
repeat

Type of instruction
Program flow instruction – repeat instruction.

Syntax
Immediate data: repeat #Data
Register direct: repeat GPRx

Operands
0 ≤ Data ≤ 255
GPRx: GPR0- GPR31

Execution
data → RepeatReg

Description
The instruction following the repeat instruction is repeated “data” number of times before the PC is incremented again. When register indirect addressing is used the 8 least significant bits are used as data.

Example
repeat #20

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat Reg</td>
<td>h0000</td>
<td>h0014</td>
</tr>
</tbody>
</table>
round

Type of instruction
MAC instruction - round.

Syntax
Register direct: round ACCx [SAT]

Operands
ACCx: ACC0, ACC1

Execution
if ACCx[low] > h8000
   ACCx + h8000 → ACCx

Description
Rounds the accumulator register. If bit 15 of ACCx is a ’1’ h8000 is added to ACCx. If SAT is added the result will be saturated after rounding.
The flags N, Z and O are updated.

Example
rnd ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC0</td>
<td>b000010901</td>
<td>b000011901</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**rsl**

**Type of instruction**
Shift instruction - rotational shift.

**Syntax**
Register direct: \( \text{rsl GPRx GPRy } [c] \)

**Operands**
- GPRx: GPR0 - GPR31
- GPRy: GPR0 - GPR31

**Execution**
GPRy \(<= (\text{GPRx } \& \text{ h1F}) \rightarrow \text{GPRy} \)

**Description**
The value in register GPRy is rotated GPRx steps to the left without carry between msb and lsb. The result is stored in register GPRy. Negative value in GPRx results in right rotation. The c bit is set to use the carry bit. The flags N and Z are updated.

**Example**
rsl GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b1000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GPRy</td>
<td>h2222</td>
<td>h8888</td>
</tr>
</tbody>
</table>
**rsli**

**Type of instruction**
Shift instruction - Rotational shift with immediate data.

**Syntax**
Immediate data: \( \text{rsli} \#\text{Step GPRy} [c] \)

**Operands**
- \(-15 \leq \text{Step} \leq 15\)
- GPRy: GPR0 - GPR31

**Execution**
GPRy \(\ll\) Step → GPRy

**Description**
The value in register GPRy is rotated, Step, steps to the left without carry between msb and lsb. Negative Step gives rotation to the right. The result is stored in register GPRy. The c bit is set to use the carry bit. The flags N and Z are updated.

**Example**
rsli #4, GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRy</td>
<td>hF222</td>
<td>h222F</td>
</tr>
</tbody>
</table>
**rti**

**Type of instruction**
Program flow instruction – return from interrupt.

**Syntax**
no operands: \( \text{rti } [i] \)

**Operands**

**Execution**
PC-stack → PC

**Description**
This instruction jumps back from the interrupt and restores the PC value. The jump is delayed two cycles, that is the two instructions following the rts instruction are executed before the jump is taken. None of the two following instructions may be bra, call, rts, loop or repeat instructions. rts may not be used as a repeat instruction or as one of the two last instructions in a hardware loop. The i flags enables interrupts again. No flags are updated.

**Example**

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC stack top</td>
<td>h0008</td>
<td>hxxxx</td>
</tr>
<tr>
<td>PC</td>
<td>h0200</td>
<td>h0008</td>
</tr>
</tbody>
</table>
**rts**

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

**Syntax**
No operands: rts

**Operands**

**Execution**
PC-stack \(\rightarrow\) PC

**Description**
This instruction jumps back from the subroutine and restores the PC value. The jump is delayed two cycles, that is the two instructions following the rts instruction are executed before the jump is taken. None of the two following instructions may be bra, call, rts, loop or repeat instructions. rts may not be used as a repeat instruction or as one of the two last instructions in a hardware loop.

No flags are updated.

**Example**
Rts

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC stack top</td>
<td>h0008</td>
<td>hxxxx</td>
</tr>
<tr>
<td>PC</td>
<td>h0200</td>
<td>h0008</td>
</tr>
</tbody>
</table>
sat

Type of instruction
MAC instruction - saturate.

Syntax
register direct: 

sat ACCx

Operands
ACCx: ACC0, ACC1

Execution
sat(ACCx) → ACCx

Description
Saturate accumulator register

If the value of ACCx cannot be represented with 32 bits, ACCx will be set too h00007FFFFFFFF or hFFFF80000000 depending on the sign of ACCx. Otherwise the value will be kept.
Flag O is set if ACCx was larger than 32-bits.

Example
sat ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC0</td>
<td>h03xxxxx</td>
<td>h007FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>xx</td>
<td>F</td>
</tr>
</tbody>
</table>


setfield

Type of instruction
Logic instruction – set bit-field in register

Syntax
Register direct: setfield GPRx GPRy GPRz GPRw

Operands
GPRx: GPR0 – GPR31 from
GPRy: GPR16 – GPR23 start
GPRz: GPR16 – GPR23 length
GPRw: GPR0 – GPR31 to

Execution
GPRx → GPRz(GPRy:GPRz)

Description
The instruction copies the number of bits pointed out by GPRz, from GPRx to GPRw. GPRy decides the start position for the GPRw register. No flags are updated.

Example
setfield GPRx GPRy GPRz GPRw

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRx</td>
<td>h1321</td>
<td>h1321</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0004</td>
<td>h0004</td>
</tr>
<tr>
<td>GPRz</td>
<td>h0008</td>
<td>h0008</td>
</tr>
<tr>
<td>GPRw</td>
<td>h6450</td>
<td>h6210</td>
</tr>
</tbody>
</table>
**setstep**

**Type of instruction**
Data move instruction – sets the step length for step 1 to 4, immediate data.

**Syntax**
Immediate data: setstep #Data #Step

**Operands**
0 ≤ Data ≤ 15
1 ≤ Step ≤ 4

**Execution**
Data → GPR9((Step-1)*4)

**Description**
The instruction sets the step size of the corresponding address pointer in GPR9.
No flags are updated

**Example**
setstep #1 #2

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR9</td>
<td>h2222</td>
<td>h2212</td>
</tr>
</tbody>
</table>
### sub

**Type of instruction**  
Arithmetic instruction - subtraction.

**Syntax**  
Register direct: \( \text{sub } \text{GPRx GPRy [c]} \)

**Operands**  
GPRx: GPR0 - GPR31  
GPRy: GPR0 - GPR31

**Execution**  
GPRy – GPRx → GPRy

**Description**  
The value in register GPRx is subtracted from the value in register GPRy and the result is stored in register GPRy. The c bit is set to use subtraction with carry in 
The flags N, Z, C and O are updated. C is set when unsigned subtraction does not generate borrow. O is set when signed subtraction generates overflow.

**Example**  
sub GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h000E</td>
</tr>
</tbody>
</table>
sub3

Type of instruction
Arithmetic instruction - subtraction

Syntax
Register direct: \text{sub3} \ GPRx \ GPRy \ GPRz \ [c]

Operands
GPRx: GPR0- GPR31
GPRy: GPR0- GPR31
GPRz: GPR16- GPR23

Execution
GPRy - GPRx \rightarrow GPRz

Description
The values in register GPRx is subtracted from the value in register GPRy and the result is
stored in register GPRz. The c bit is set to use subtraction with carry in.
The flags N, Z, C and O are updated. C is set when unsigned subtraction does not generate
borrow. O is set when signed subtraction generates overflow.

Example
\text{sub3} \ GPRx \ GPRy \ GPRz

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h0020</td>
</tr>
<tr>
<td>GPRz</td>
<td>hxxxx</td>
<td>h000E</td>
</tr>
</tbody>
</table>
subi

**Type of instruction**
Arithmetic instruction - subtraction with immediate data.

**Syntax**
Immediate data without carry: subi #Data GPRy [c]

**Operands**
h0 ≤ Data ≤ hFFFF
GPRy: GPR0 - GPR31

**Execution**
GPRy − Data → GPRy

**Description**
The value Data is subtracted from the value in register GPRy and the result is stored in register GPRy. The c bit is set to use subtraction with carry in.
The flags N, Z, C and O are updated. C is set when unsigned subtraction does not generate borrow. O is set when signed subtraction generates overflow.

**Example**
subi #h5 GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h001B</td>
</tr>
</tbody>
</table>
subsub

Type of instruction
Arithmetic instruction – two subtractions

Syntax
Register direct without carry: subsub GPRx GPRy GPRz GPRw ACCx

Operands
GPRx: GPR0-GPR31
GPRy: GPR16-GPR23
GPRz: GPR16-GPR31
GPRw: GPR16-GPR23
ACCx: ACC0-ACC1

Execution
GPRy - GPRx \rightarrow ACCx [31:16]
GPRw - GPRz \rightarrow ACCx [15:0]

Description
The values in register GPRx is subtracted from the value in register GPRy and the result is stored in the higher part of ACCx. At the same time the values in register GPRz is subtracted from the value in register GPRw and the result is stored in the lower ACCx register. The flags N, Z, and C are updated according to GPRy - GPRx. C is set when unsigned subtraction does not generate borrow. O is set when any of the signed operations generates overflow.

Example
subsub GPRx GPRy GPRz GPRw ACCx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0010</td>
</tr>
<tr>
<td>GPRx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GPRy</td>
<td>h0020</td>
<td>h0020</td>
</tr>
<tr>
<td>GPRz</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GPRw</td>
<td>h0006</td>
<td>h0006</td>
</tr>
<tr>
<td>ACCx (low)</td>
<td>hxxxx</td>
<td>h0004</td>
</tr>
<tr>
<td>ACCx (high)</td>
<td>hxxxxx</td>
<td>h00000E</td>
</tr>
</tbody>
</table>
**XOR**

**Type of instruction**
Logic instruction - bitwise xor.

**Syntax**
Register direct: xor GPRx GPRy

**Operands**
GPRx: GPR0 - GPR31
GPRy: GPR0 - GPR31

**Execution**
GPRx ⊕ GPRy → GPRy

**Description**
Bitwise xor between the values in register GPRx and GPRy. The result is stored in register GPRy.
The flags N and Z are updated.

**Example**
xor GPRx GPRy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRx</td>
<td>h8000</td>
<td>h8000</td>
</tr>
<tr>
<td>GPRy</td>
<td>h8012</td>
<td>h0012</td>
</tr>
</tbody>
</table>
**xori**

**Type of instruction**
Logic instruction - bitwise xor with immediate data.

**Syntax**
Immediate data: \( \text{xori} \ #\text{Data GPRy} \)

**Operands**
h8000 ≤ Data ≤ h7FFF
GPRy: GPR0 - GPR31

**Execution**
Data \( \oplus \) GPRy \( \rightarrow \) GPRy

**Description**
Bitwise xor between the value in register GPRy and the value Data. The result is stored in register GPRy.
The flags N and Z are updated.

**Example**
\( \text{xori} \ #\text{h1111 GPRy} \)

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0000</td>
</tr>
<tr>
<td>GPRy</td>
<td>h1234</td>
<td>h0325</td>
</tr>
</tbody>
</table>
Conditional execution

Whit conditional execution instructions can be performed or discarded, depending on the condition. If the condition is not fulfilled the instruction will be “behandlad” like a nop instruction.

The conditional execution can’t be used on memory instructions and program flow instructions.

The condition is written after the instruction with a separating space.

Below is the conditions that can be used shown.

<table>
<thead>
<tr>
<th>{cond}</th>
<th>Relation</th>
<th>Flag status</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt</td>
<td>GPRx &gt; GPRy</td>
<td>Z=0 and N=0</td>
</tr>
<tr>
<td>ge</td>
<td>GPRx &gt;= GPRy</td>
<td>N=0</td>
</tr>
<tr>
<td>lt</td>
<td>GPRx &lt; GPRy</td>
<td>N=1</td>
</tr>
<tr>
<td>le</td>
<td>GPRx &lt;= GPRy</td>
<td>Z=1 or N=0</td>
</tr>
<tr>
<td>eq</td>
<td>GPRx = GPRy</td>
<td>Z=1</td>
</tr>
<tr>
<td>ne</td>
<td>GPRx /= GPRy</td>
<td>Z=0</td>
</tr>
<tr>
<td>c</td>
<td>carry</td>
<td>C=1</td>
</tr>
<tr>
<td>nc</td>
<td>not carry</td>
<td>C=0</td>
</tr>
<tr>
<td>o</td>
<td>overflow</td>
<td>O=1</td>
</tr>
<tr>
<td>no</td>
<td>not overflow</td>
<td>O=0</td>
</tr>
</tbody>
</table>

Example:
move GPR1 GPR2 gt
Labels

Labels are used to simplify jump and call operations.

The label is distinguish by a “!” and a following label, letter or digit, on a new line. The label will be translated to the next code line number. To use the label in the call, write the instruction followed by a “?” and the label.

Example:

....
!a
....
Jmp ?a

Write see labels in jmp and call functions.
## Appendix B

### Vendor Model/Family Introduced Data/Arithmetic Comments

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model/Family</th>
<th>Introduced</th>
<th>Data/Arithmetic</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog Devices</td>
<td>ADSP-21xx</td>
<td>1986</td>
<td>16/fixed</td>
<td>24 – bit</td>
</tr>
<tr>
<td>Analog Devices</td>
<td>ADSP-2106x</td>
<td>1994</td>
<td>32/floating</td>
<td>48-bit instructions</td>
</tr>
<tr>
<td>Analog Devices</td>
<td>ADSP-2116x</td>
<td>1998</td>
<td>32/floating</td>
<td></td>
</tr>
<tr>
<td>Lucent Technologies</td>
<td>DSP16xx</td>
<td>1987</td>
<td>16/fixed</td>
<td>16 - and 32 bit instructions</td>
</tr>
<tr>
<td>Lucent Technologies</td>
<td>DSP16xxx</td>
<td>1997</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Motorola</td>
<td>DSP560xx</td>
<td>1987</td>
<td>24/fixed</td>
<td>24 – bit</td>
</tr>
<tr>
<td>Motorola</td>
<td>DSP563xx</td>
<td>1995</td>
<td>24/fixed</td>
<td>instruction</td>
</tr>
<tr>
<td>Motorola</td>
<td>DSP566xx</td>
<td>1995</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Motorola</td>
<td>DSP568xx</td>
<td>1990</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C2xx</td>
<td>1995</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C27xx</td>
<td>1998</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C3x</td>
<td>1988</td>
<td>32/floating</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C4x</td>
<td>1991</td>
<td>32/floating</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C5x</td>
<td>1990</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C54x</td>
<td>1994</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C62xx</td>
<td>1997</td>
<td>16/fixed</td>
<td></td>
</tr>
<tr>
<td>Texas Instrument</td>
<td>TMS320C67xx</td>
<td>1998</td>
<td>32/floating</td>
<td>VLIW</td>
</tr>
<tr>
<td>ZSP</td>
<td>ZSP164xx</td>
<td>1998</td>
<td>16/fixed</td>
<td>Superscalar</td>
</tr>
</tbody>
</table>

### Function

<table>
<thead>
<tr>
<th>Function</th>
<th>ADI 218x</th>
<th>ADI 2106x</th>
<th>ADI 2116x</th>
<th>Lucent 16xx</th>
<th>Lucent 16xxx</th>
<th>Motorola 560xx</th>
<th>Motorola 563xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Block FIR</td>
<td>841</td>
<td>807</td>
<td>572</td>
<td>1240</td>
<td>719</td>
<td>806</td>
<td>928</td>
</tr>
<tr>
<td>Single-Sample FIR</td>
<td>88</td>
<td>24</td>
<td>18</td>
<td>26</td>
<td>16</td>
<td>21</td>
<td>24</td>
</tr>
<tr>
<td>Complex Block FIR</td>
<td>3122</td>
<td>2810</td>
<td>1526</td>
<td>3123</td>
<td>1721</td>
<td>2847</td>
<td>2849</td>
</tr>
<tr>
<td>LMS Adaptive FIR</td>
<td>60</td>
<td>53</td>
<td>53</td>
<td>101</td>
<td>41</td>
<td>58</td>
<td>64</td>
</tr>
<tr>
<td>IIR</td>
<td>43</td>
<td>41</td>
<td>41</td>
<td>65</td>
<td>33</td>
<td>37</td>
<td>46</td>
</tr>
<tr>
<td>Vector Dot Product</td>
<td>43</td>
<td>48</td>
<td>41</td>
<td>41</td>
<td>65</td>
<td>33</td>
<td>46</td>
</tr>
<tr>
<td>Vector Add</td>
<td>83</td>
<td>84</td>
<td>44</td>
<td>123</td>
<td>46</td>
<td>84</td>
<td>87</td>
</tr>
<tr>
<td>Vector Maximum</td>
<td>128</td>
<td>123</td>
<td>47</td>
<td>120</td>
<td>43</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>256-Point FFT</td>
<td>10633</td>
<td>6286</td>
<td>4614</td>
<td>21035</td>
<td>8485</td>
<td>8323</td>
<td>10907</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Motorola 566xx</th>
<th>Motorola 568xx</th>
<th>TI 320C2xx</th>
<th>TI 320C27xx</th>
<th>TI 320C3x</th>
<th>TI 320C4x</th>
<th>TI 320C5x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Block FIR</td>
<td>928</td>
<td>925</td>
<td>1038</td>
<td>1118</td>
<td>1040</td>
<td>960</td>
<td>842</td>
</tr>
<tr>
<td>Single-Sample FIR</td>
<td>24</td>
<td>22</td>
<td>21</td>
<td>22</td>
<td>20</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>Complex Block FIR</td>
<td>2849</td>
<td>3089</td>
<td>3998</td>
<td>3798</td>
<td>3080</td>
<td>2960</td>
<td>3602</td>
</tr>
<tr>
<td>LMS Adaptive FIR</td>
<td>64</td>
<td>63</td>
<td>104</td>
<td>87</td>
<td>65</td>
<td>57</td>
<td>83</td>
</tr>
<tr>
<td>IIR</td>
<td>46</td>
<td>52</td>
<td>110</td>
<td>128</td>
<td>55</td>
<td>52</td>
<td>83</td>
</tr>
<tr>
<td>Vector Dot Product</td>
<td>47</td>
<td>45</td>
<td>45</td>
<td>44</td>
<td>46</td>
<td>41</td>
<td>45</td>
</tr>
<tr>
<td>Vector Add</td>
<td>87</td>
<td>85</td>
<td>158</td>
<td>158</td>
<td>85</td>
<td>85</td>
<td>122</td>
</tr>
<tr>
<td>Vector Maximum</td>
<td>85</td>
<td>85</td>
<td>300</td>
<td>468</td>
<td>122</td>
<td>117</td>
<td>164</td>
</tr>
<tr>
<td>256-Point FFT</td>
<td>10907</td>
<td>15636</td>
<td>25035</td>
<td>27045</td>
<td>12405</td>
<td>11879</td>
<td>20891</td>
</tr>
<tr>
<td>Function</td>
<td>TI 320C54x</td>
<td>TI 320C62xx</td>
<td>TI 320C67xx</td>
<td>ZSP 164xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real Block FIR</td>
<td>684</td>
<td>334</td>
<td>479</td>
<td>551</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-Sample FIR</td>
<td>10</td>
<td>18</td>
<td>34</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex Block FIR</td>
<td>2922</td>
<td>1294</td>
<td>1806</td>
<td>1586</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS Adaptive FIR</td>
<td>58</td>
<td>32</td>
<td>46</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IIR</td>
<td>44</td>
<td>30</td>
<td>48</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Dot Product</td>
<td>41</td>
<td>29</td>
<td>44</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add</td>
<td>61</td>
<td>36</td>
<td>48</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Maximum</td>
<td>11</td>
<td>39</td>
<td>42</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256-Point FFT</td>
<td>13233</td>
<td>4225</td>
<td>4419</td>
<td>7026</td>
<td></td>
<td></td>
<td></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 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/](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/](http://www.ep.liu.se/)

© [Per Lennartsson, Lars Nordlander]