

QDSP ***Domain-Specific Language***

Documentation:

Education and semester:

Robot Systems Engineering - 9th semester

Revision:

1.0 - MMMI SDU 01/2014

Supervisors:

Anders Stengaard Sørensen & Ulrik Pagh Schultz

Developed by:

Jon Henneberg & Anders Krauthammer



*Faculty of Engineering
University of Southern Denmark*

Contents

1	Introduction	2
2	Design	3
2.1	DSP	3
2.2	Run-time programing	4
2.3	Language	6
3	Implementation	9
3.1	Compiler	9
3.2	Validator	15
4	Testing	16
5	Conclusion	17
A	Instruction-set	18
B	Testing result	19

Introduction

1

This document along with the QDSP user manual, is the result of a self study project in domain-specific languages (DSL). This activity will focus on creating a DSL that will allow for the creation of functioning machine code for the QDSP which has been developed at MMMI SDU. This DSP has been created to allow easier transfer of regulations algorithms from the computer to the FPGA, with a minimum of changes to the gateway, in order to ease the development of code on the QDSP. The key challenge is thus in the automatic generation of code, not in programming of FPGAs.

In order to accomplish the desired tasks a compiler will need to be created that is capable of decoding a custom language, and create both a VHDL module containing the QDSP and the code it should run. The compiler should also be able to create the machine code for the DSP allowing it to be programmed onto the DSP at run-time. It would be preferable if the compiler was capable of optimizing the DSP, so only the instructions that are needed gets implemented thereby conserving space on the FPGA, both by not having the logic and allowing the address-busses to operate at a reduced size.

Design

2

The overall goal for the design of the DSP and DSL was to allow for the greatest amount of flexibility, so the resulting DPS generated would require the minimum amount of space on the system.

2.1 DSP

The basis of the design comes from the QDSP designed by Anders Steengaard. This design was maintained in the VHDL as to support interoperability between the old and new design while additional features were added. The overall design of the DSP can be seen in figure 2.1.1. The design was created to allow the DSP to be inserted in applications that use μ TosNet or Unity-Link and only require minimal changes to the existing VHDL. The result of this design is that the DSP has two main memory areas, on to interface with the surrounding VHDL, called the IO memory, and one to either μ TosNet or Unity-Link referred to as the Shared memory.

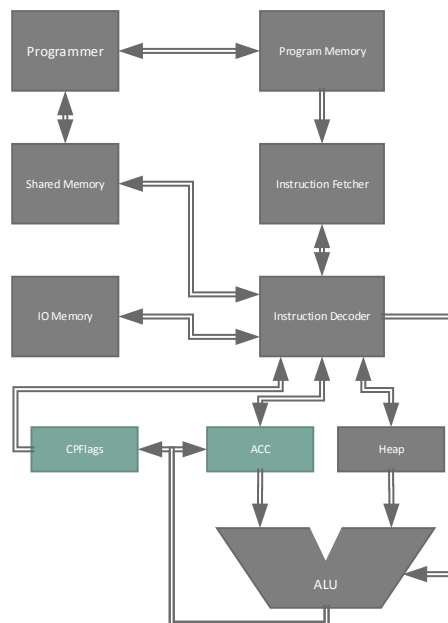


Figure 2.1.1: Block diagram of QDSP

2.1.1 Instruction-set

The structure of the instruction-set from the original QDSP was designed to allow for arguments to be provided along with the instruction. This was done by having a fixed length instruction followed by argument, this structure was maintained in this new design.

The instruction-set was updated to allow for new features such as branching whereby evaluations on values could be done and subsequent jumps in the program code could be achieved, allowing for loops and conditional code execution.

Existing features of the old instruction-set such as the ability to divide or multiply by a fixed base2 number such as 256 using bit-shifting, was removed and an instruction that could do the same but also take an argument for the amount of bits to shift were put in their place, thus reducing the amount of instructions needed, when wanting to f.x. divide by two different numbers.

The full instruction-set can be found in appendix A.

2.2 Run-time programming

Run-time programming functionality, designed to utilize the existing computer interface of the QDSP, was added to allow for the run-time programming the program memory, allowing for faster iterations on the code running on the DSP. To do this an additional component was added to the DSP that has access to the program memory and the shared memory, allowing it to circumvent the main instruction decoder. A file format was created to allow the compiler to create files that could be programmed to the device using a predetermined interface.

2.2.1 Programming interface

The programming interface was devised so it allowed for the programming of individual parts of the program and heap memory, as well as validation of the functionality of the device before programming in started. The interface uses four registers in on either μ Tosnet or Unity-Link, two to write data and two to receive. The complete command list can be seen in table 2.2.1.

Name	Transmit			Receive		
	Control		Data-Out	Status		Data-In
start loader	0x81	N/A	N/A	0x81	N/A	N/A
stop loader	0x82	N/A	N/A	N/A	N/A	N/A
write program	0x83	Address	program data	0x83	Address	program data
read program	0x84	Address	N/A	0x84	Address	program data
write heap	0x85	Address	heap data	0x85	Address	heap data
read heap	0x86	Address	N/A	0x86	Address	heap data
validate device	0x87	Feature key	Validation key	0x87	Feature result	Validation result

Table 2.2.1: List of commands used in the program loader.

The control and status registers uses the structure defined in table 2.2.2, that allows extra data to be transmitted along side the command, and the Data-in and -out allows for the transmission of data to the DSP. The Start and Stop loaders commands are used to inform the DSP to go into the programming state. when the programming state is entered the is returned in the status register, but as the loader is stopped when a stop loader command is sent, not change on the status register is possible.

Start	Size[Byte]	Description
0	2	command
2	6	data

Table 2.2.2: Structure of control and Status registers.

The read program and read heap commands uses the address provided in the control data area to return the address and data of a given memory area. When writing a new program to the DSP first the validation must be performed, until that is done the loader is in a read only state. This is done by transmitting a validation command along with a feature key, this feature key is a on-hot encoding of all the instructions required by the code, and a validation key, which structure can be seen in table 2.2.3. These keys are validated on the DSP and the results are returned. If the validation was successful the read-only flag is removed allowing for the programming of the DSP.

Start	Size[Bit]	Description
0	10	Heap size
10	22	Program size

Table 2.2.3: Structure of the validation key and result.

The write program and heap commands requires the address that is to be written along side the data, when that is received the data is stored in the memory and the new address and data value is returned, allowing for validation of the written data.

2.2.2 Programming file format

The programming file is designed to allow the programming software to get all parameters needed to successfully program the DSP. The data is stored in a binary format, consisting of a header followed by the heap and program data. The overview of the file structure can be seen in table 2.2.4.

Start	Size[Byte]	Name	Description
0	1	HS	Size of header
1	1	CA	Control Address
2	1	OA	Data-Out Address
3	1	IA	Data-In Address
4	1	SA	Status Address
5	1	PCL	Program Code Length
6	2	HC	Heap Count
8	2	PC	Program Count
10	6	FK	Feature Key
16	8	VK	Validation Key
HS	$5 \cdot HC$	HD	Heap Data
$HS + 5 \cdot HC$	$PCL \cdot PC$	PD	Program Data

Table 2.2.4: Data structure of programming file

Header

The first data is used to determine the size of the header, which allows for changes to the header and still retain backwards compatibility. The next four elements describes what registers should be used for the communication. These are followed by the PCL element, which describes the size of each Program data element in bytes, this ensures that the program data occupies a minimum of space in the file. The heap and

program count are used to calculate the start and end of the data areas, as they have no predefined length, but depends on the program created. Lastly the feature and validation keys that are used to ensure that the DSP supports the functions required of the program.

Heap data

The heap data is structured with an address and value, this means that only the heap elements that has initial values needs to be stored in the file and programmed. The structure can be seen in table 2.2.5.

Start	Size[Byte]	Description
0	1	Address
1	4	Value

Table 2.2.5: Data structure of heap data element

Program data

The program data stores the code for the entire program, stored sequentially this is done as there is no need to assign addresses to the programming code, as it will need to overwrite the entire code every time the DSP is programmed. The structure of a program data element can be seen in table 2.2.6.

Start	Size[Byte]	Description
0	PCL	Value

Table 2.2.6: Data structure of program data element

2.3 Language

There are generally two different ways of making DSLs. They can be either internal, where they are running inside of another language, and external, where a separate program is used and new files are generated. In the current project an external DSL was the most applicable. In order to facilitate the creation of the DSL eclipse with xtext was chosen as the development platform, as this allows for the creation of an eclipse plug-in. By doing this the DSL can later be written and compiled in eclipse all the features of eclipse, such as syntax and error highlighting.

2.3.1 General structure

The overall structure of the language resembles C, thou utilizing a limited subset of the features, making it easier for anyone with experience with a C-like programming language to program to the QDSP. As the structure of the DSP is unique there are several differences that bares mentioning.

Data formats

The data formats that the language has are define, var, and constant.

Constants are used when defining a number that can not be changed later in the code.

Defines are used to give meaningful names to the memory addresses in the two main interfaces of the DSP i.e. the IO memory and Shared Memory

Vars are used when dealing with variables on the DSP, these occupy the heap, and the data can be changed anywhere in the program.

Math operators

The mathematical operators implemented thus far can be found in table 2.3.1. These are selected as they are the basic mathematical operations needed to do most math. The order of operations chosen is purely based on the order of the operators, with no significance given to the type of operator used, this was chosen as it would require less time to implement, and allowed more focus on other parts of the design. In order to allow specific operations to be evaluated correct, support for parenthesis was included as this gave a greater freedom when writing the equations.

Operator	Description
+	Add
-	Subtract
*	Multiply ¹
/	Divide ²
«	Left bit shift
»	Right bit shift

Table 2.3.1: Math operators

There are some caveats with some of the operators, namely the multiplication and division. The multiplication only outputs its result into the same signed 32bit values as all data elements, this requires the user to ensure that the result does not exceed this when calculating, as data will be lost. The division used in the design is based on bit-shifting as this can be done on one clock cycle but the downside to this is it only supports division with numbers that are a power of two.

Assignment operators

The assignment operators that are implemented is a natural extension of the supported mathematical operators, as they are dependent on the mathematical operators. A list of the assignment operators can be found in table 2.3.2.

Operator	Description
=	Sets the data to the argument
+=	Adds the argument
-=	Subtracts the argument
*=	Multiplys with the argument
++	Adds 1
--	Subtracts 1

Table 2.3.2: Assignment operators

2.3.2 Custom functions

In addition to the normal c-like options of language several custom functions have been added to enable some of the more specific features of the DSP, they can be seen in table 2.3.4.

¹The multiplier multiplies to values and stores the result in a 32 bit value make sure that the result dos not exceeds a 32 bit value.

²It is only possible to divide by a number that is a power of 2. It is therefore not possible to divide by variables and constants.

Name	Format	Description
Min	Min(x, y)	Finds and returns the lowest value of x and y
Max	Max(x, y)	Finds and returns the largest value of x and y
Sleep	Sleep(x)	Inserts x amount of NOPs in the program code
truncate16	truncate16(x)	Truncates the number to a 16bit value
RunBootLoader	RunBootLoader()	Makes the code test if run-time programming is requested

Table 2.3.3: Custom functions

This is done as most of these map directly to instructions in the instruction set and does not have a natural way to be included with normal mathematical operations. The only function that does not map directly to a instruction is the sleep function, which adds a number of NOP operations to the code.

2.3.3 Configuration

In addition to the functional code the language also has parameters that can be used to adjust how the compiler processes the code. These are included in the language as opposed to having as parameters in the project, as this solution did not require an in-depth analysis into how eclipse stores project settings and allows an entire project to be contained within a single file. The entire list of parameters can be seen in table 2.3.4

Name	Type	Default	Description
ProjectTitle	String	NewQdsp	Title of the project
AutoCommentQDSPCode	Boolean	true	Comments the micro instructions
EnableOptimization	Boolean	true	Assigns the programing transmit address on SH memory for run time programming
RuntimeProgrammable	Boolean	false	Enables the capability of run time program the QDSP
HeapSize ³	Integer	32	The amount of Heap registers.
ProgramMemorySize ³	Integer	128	The amount of program memory available.
ProgramingControlAddress ³	Integer	7	Assigns the programing control address on SH memory for run time programming
ProgramingStatusAddress ³	Integer	0	Assigns the programing status address on SH memory for run time programming
ProgramingDataInAddress ³	Integer	1	Assigns the programing revise address on SH memory for run time programming
ProgramingDataOutAddress ³	Integer	6	
SHInterfaceType ³	String	Unity-Link	Changes the SH interface to match Unity-Link or uTosNet

Table 2.3.4: Configuration paramaters

³Only used if RuntimeProgrammable it true

Implementation

3

3.1 Compiler

As xtext was the chosen development framework, the compiler was written in a combination of Java and xtend, a language that extends java with new functionality but at the same time removes others.

3.1.1 Structure

With an object oriented development environment, several options on how to decode and handle data presents itself:

Simple structure

A simple structure where the instructions are stored as a list of similar objects containing the instruction name and argument. This has the advantage of being simple and fast to create but it does not utilize all the features of the programming language, and makes it harder to add functionality to the code.

Inherited structure

In an inherited structure a class is created for each instruction and abstract classes are created to represent the various features of instructions. The advantage of this structure is that it allows for easier identification of the instruction type and data specific to that type can be stored on the class. It also allows for easier addition of new instructions as the surrounding structure utilizes the highest level of the code it needs. The disadvantages are that it requires more analysis when starting the design, as it is essential to get meaningful parent classes.

Interface structure

The interface structure is very similar to the inherited structure in that it utilizes the underlying language well but instead of parent classes, interfaces are used. The main difference between the two is that a class would be able to more than one interface, but not have the same data structures on the parent class.

implemented structure

The structure chosen in this case was the inherited structure, as the ability to store data on the higher levels will be useful for linking data, and the grouping of instructions makes the optimizer simpler, more robust, and more general. An overview of the structure of the code can be seen in figure 3.1.1.

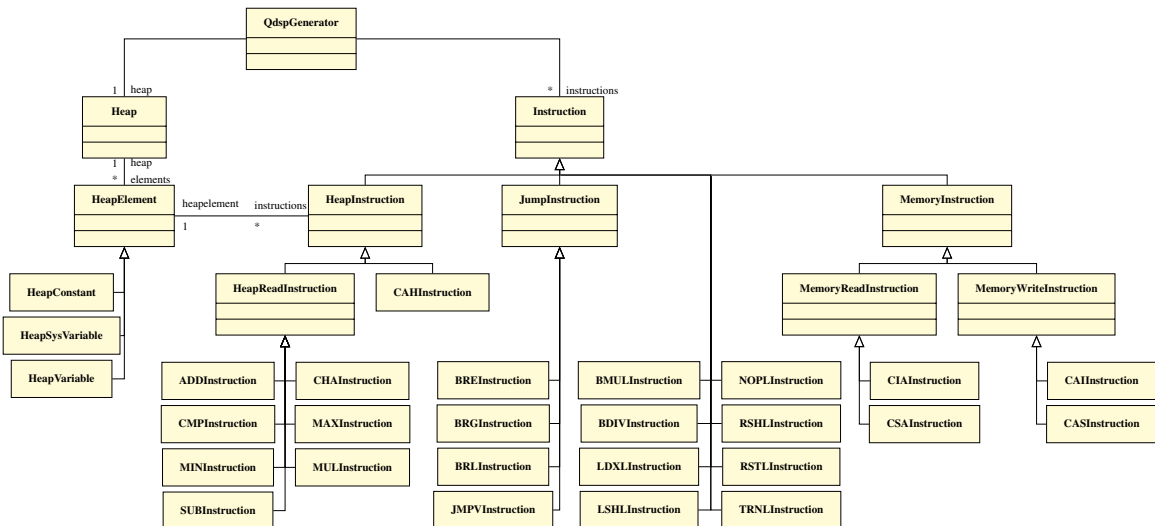


Figure 3.1.1: Diagram showing the structure of the generator

The main grouping of instructions are into heap instructions that are used for instructions that access the heap, and each heap instruction has a link to the heap element that identifies where on the heap the data is stored, and the type of data stored there. The memory instructions that identify the memory type accessed and whether it is a read or write operation. The jump instruction that stores where a jump operation comes from and goes to.

The heap structure allows for a reuse creation of temporary variables, refereed to as system variables, that can be used by several operations, and these can be claimed and released, allowing for an optimization on the use of the variables. The heap elements also keeps track of what instructions are referring to it, allowing for the optimization of the fetches from the heap.

The compilation of the code is separated in three distinct processes. First the language is decoded into the structure of instructions and heap elements, then an optimization is run on the generated instructions, finally the output files are generated.

3.1.2 Decoder

The main functionality for the decoder is to translate the written program into a list of of micro instructions. The list consists of the different instruction objects all inheriting from the instruction class. This allows the different instructions to contain the needed information for the current instruction.

An example of a program are seen in code sample 3.1.1. This small program drives four LEDs switching them on and of so that the light runs up and down. The diodes are mapped the IO addres 4 pin 19 to 22.

Code Sample 3.1.1: Example of code

```

1  define LED IO[4]
2  const sleeptime = 250000
3  var v1 = 1 << 16
4  var v2 = 55
5  var dir = 1
6  LED = 0
7
8  while(true){
9      if (dir == 1) {
10         v1 = v1 << 1
11     } else {

```

```

12     v1 = v1 >> 1
13 }
14 LED = v1
15
16 for(v2 = 0; v2 < sleeptime; v2++){
17     sleep(1)
18 }
19
20 if (v1 >= 1 << 22) {
21     dir = 0
22 } else if (v1 <= 1 << 19) {
23     dir = 1
24 }
25 }

```

The decoder loops through the code lines, and uses the multiple dispatch principle made possible in `xtend`, to separate the various types of lines, and thereby call the right function handling the specific line of code. The handling of the various types are further described in the subsections below. The decoder is written in a recursive way, that allows all the function handlers to call each other and themselves.

The way that the decoder looks at the code, code sample 3.1.1 consists of 7 lines of code in the first iteration. Namely the first 6 lines, and the `while` at line 8. The code inside the `while` is handed by the `while` function handler in the second iteration. The first 5 lines creates variables, constants, and defines. Line 6 is a instruction setting the LED defined in line one to 0.

Math decoder

The math handler, is not used directly by the handler in first iteration, but is used by the other handlers to handle a math operation. An example of a math operation can be seen in code sample 3.1.1 line 10 where the instruction has a bit shift operation.

The math contains both functions and mathematical operation and is analyzed from behind to find the last function, address, or instance of define. A parenthesis is implemented as a function that can be used inside math operations like the `truncate16()`, `max()`, and `min()`. When the last function or parenthesis is found, the function is handled, and all the skipped operations are executed and the micro instructions are added to the instruction list. This makes it possible to guarantee that functions and parenthesis are executed before the mathematical operations, and the values from the addresses to be loaded. The partial result is then stored in a variable declared by the decoder, that now contains the last part of the math. The process continues until all elements in the math have been executed.

Instruction decoder

The main part of the code will often be instructions, which allows the use of assignment operators to assign values and other data formats to variables, addresses, and defines.

If the instruction is a increment (`++`) or decrement (`--`) instruction, the instruction decoder adds the micro instruction to load the variable to the accumulator, and the creates a math operation to add or subtract 1. The math decoder is then used to decode it to the right micro instructions.

If the instruction is an assignment instruction the math decoder is called to decode the math operation, that afterwards are assigned to the given variable by adding the micro instruction to store the accumulator to the give address on the heap. At last the assignment instruction can have a argument. In that case the given result variable is added in the front of the math with a math operator according the argument.

Variable decoder

The variable decoder is used when a variable is declared. It calls the add element on the heap class, that creates a new heap element. During declaration of a variable, it is possible to assign a value or a math operation directly to the variable. If there is assigned a value it is set in the heap element. If a math operation is assigned the math decoder is used to add the micro instructions to the instruction list so the QDSP executes the math and the micro instruction to store the accumulator to given heap address is added.

Constant decoder

The constant decoder adds a heap element on the heap class with the value of the constant, then a constant is declared.

Function decoder

The function decoder is once again using the multiple dispatch principle for breaking the code handling the different functions into different parts. The most of the functions maps directly to a single micro instruction, the decoding of these are therefore relatively simple, the micro instruction is added to the instruction list, with the parameters given in the used function. The functions creating jumps such as `if()` or the various loop functions uses branch and jump instructions to control the flow of the code on the QDSP. These are more complicated to decode, and are explained beneath.

If Switch

The *if* and *switch* functions uses the same principles, and are almost similar. Therefore only the *if* functions is described in details.

The if function decoder again uses the math decoder to decode the to math expressions in the brackets, storing the first a variable, and keeps the last on the accumulator. Afterwards the micro instructions to compare and branch and jump to end of the code inside the `if()` are added. It constantly keeps track of the positions in the micro instructions for the code blocks, so its possible to jump over the code if the branch tells it to. All the *else if* statements are handled the same way.

A jump instruction is created in the start and added to the instruction list in the end of every *if* or *else if* statement, ensuring that the *else if* only are valuate if the first statement is false. The same count for a *else*. At the end them all the micro instruction is added, and the end position for the whole *if*, *else if*, *else* statement is known, the value for the initially created jump instruction is set.

Loop functions

The three loop functions implemented are all decoded by the same decoder. If the Boolean value is true, the decoder just calls the instruction decoder with the instructions inside the loop, and adds a jump to the start of the loop. If the Boolean value is false the instructions inside the loop is skipped.

If the loop function contains a Boolean expression, if the function is a *while* or a *for* a jump instruction is added followed by the instructions for the code inside the loop. The value for the jump instruction is then set to jump over the code. If the function is a *do* the jump instruction is ignored. This allows the Boolean expression to be evaluated in the end of the loop, for all three types. The micro instructions for the evaluation of the expression is the added, and a branch and jump instruction is used to determine if the code loops.

3.1.3 Optimizer

When starting the optimization of the instructions generated by the decoder, the code is separate into blocks based on jumps created in the code. Each blocks keep track what blocks lead jump to it, and what block it jumps to. This allows the optimizer ensure that removing an instruction does not make unexpected changes to other code paths. This is achieved by iterating over the entire list of instructions and determining if they are instances of jump instructions, and splitting the existing blocks based on the jump destinations. An example of the code blocks generated and the jumps associated can be seen in code sample 3.1.2 this is generate from the program found in code sample 3.1.1. When ever an instruction is optimized away the groups are updated to reflect the change.

The optimization is run i several stages and parts to try and get a structure that is easily maintained and robust. The first step that is done is to try and optimize the blocks that have been found, this means taking all the blocks that have more that two or more blocks jumping to it, and determining if they all end with the same instruction. If this is the case the instructions are removed from all the parent blocks and added to the start of the child block, this is repeated until there is a difference in one of the blocks. This is the first and simplest level of the optimization, the next part runs through all the instructions and evaluate whether or not they can be removed. Based on the instruction type it is processed through one of three methods Optimize Memory Read, Accumulator or Heap Loads, which will be described in greater detail below.

Optimize Memory Read

Tries to consolidate the use of memory read instructions by setting the memory address of the read instruction in an unused previous memory read, as memory reads requires takes more than one clock cycle.

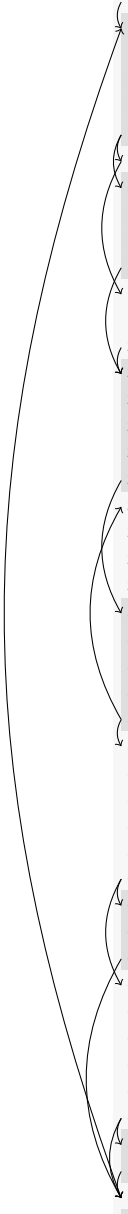
The process iterates back from the instruction to determine if any previous instructions allow for the the removal of the initial memory read of the instruction. If a memory write to the same memory type or a memory read that requires a different address is found before a free memory read, the iteration path is blocked, and the instruction is not optimized away. In the case that the reverse iteration hits the start of the block before finding any valid memory reads, each block is evaluated using the same rules as for the first block and the result for the block is stored. This allows the optimization to iterate back through all execution paths and ensure that they all allow for the optimization of the instruction before it is done.

Optimize Accumulator

Tries to remove unnecessary loads of data from the memory to the accumulator, which occurs when a value from f.x. a calculation is stored to a variable and used in the next calculation as well.

The process iterates back from the instruction to determine if the load from memory is needed or if the accumulator will have the correct value. Just like the memory read optimization it iterates back through the blocks parent if no instructions that affect the accumulator is found in the current block.

Code Sample 3.1.2: Example of code blocks and associated jumps



```

1  CHA & A(4),      -- Copy dir2 (Heap4) to accumulator
2  CAH & A(4),      -- Copy accumulator to dir2 (Heap4)
3  CHA & A(5),      -- Copy Constant = 0 (Heap5) to accumulator
4  CAI & A(4),      -- Copy accumulator to IO memory address 4
5  CHA & A(6),      -- Copy Constant = 1 (Heap6) to accumulator
6  LSH & A(16),     -- Left shifting accumulator 16 times
7  CAH & A(0),      -- Copy accumulator to v1 (Heap0)
8  CHA & A(6),      -- Copy Constant = 1 (Heap6) to accumulator
9  CAH & A(7),      -- Copy accumulator to sys0 (Heap7)
10 CHA & A(3),      -- Copy dir (Heap3) to accumulator
11 CMP & A(7),      -- Compares accumulator with sys0 (Heap7), sets flags for branch instruction
12 BRE & A(13),     -- Absolute jump over next line, if equal flag is set. Codeline 13
13 JMP & A(17),     -- Absolute jump to start of next Else. Codeline 17
14 CHA & A(0),      -- Copy v1 (Heap0) to accumulator
15 LSH & A(1),      -- Left shifting accumulator 1 times
16 CAH & A(0),      -- Copy accumulator to v1 (Heap0)
17 JMP & A(20),     -- Absolute jump to end of If. Codeline 20
18 CHA & A(0),      -- Copy v1 (Heap0) to accumulator
19 RSH & A(1),      -- Right shifting accumulator 1 times
20 CAH & A(0),      -- Copy accumulator to v1 (Heap0)
21 CHA & A(0),      -- Copy v1 (Heap0) to accumulator
22 CAI & A(4),      -- Copy accumulator to IO memory address 4
23 CHA & A(5),      -- Copy Constant = 0 (Heap5) to accumulator
24 CAH & A(2),      -- Copy accumulator to v2 (Heap2)
25 JMP & A(29),     -- Absolute jump to end of Loop, for compare. Codeline 29
26 NOP & A(0),      -- Do Nothing
27 CHA & A(2),      -- Copy v2 (Heap2) to accumulator
28 ADD & A(6),      -- Adding Constant = 1 (Heap6) to accumulator
29 CAH & A(2),      -- Copy accumulator to v2 (Heap2)
30 CHA & A(1),      -- Copy sleeptime (Heap1) to accumulator
31 CAH & A(7),      -- Copy accumulator to sys0 (Heap7)
32 CHA & A(2),      -- Copy v2 (Heap2) to accumulator
33 CMP & A(7),      -- Compares accumulator with sys0 (Heap7), sets flags for branch instruction
34 BRL & A(25),     -- Absolute jump to start of Loop, if less than flag is set. Codeline 25
35 CHA & A(6),      -- Copy Constant = 1 (Heap6) to accumulator
36 LSH & A(22),     -- Left shifting accumulator 22 times
37 CAH & A(7),      -- Copy accumulator to sys0 (Heap7)
38 CHA & A(0),      -- Copy v1 (Heap0) to accumulator
39 CMP & A(7),      -- Compares accumulator with sys0 (Heap7), sets flags for branch instruction
40 BRL & A(43),     -- Absolute jump to start of next ElseIf, if less than flag is set. Codeline 43
41 CHA & A(5),      -- Copy Constant = 0 (Heap5) to accumulator
42 CAH & A(3),      -- Copy accumulator to dir (Heap3)
43 JMP & A(51),     -- Absolute jump to end of If. Codeline 51
44 CHA & A(6),      -- Copy Constant = 1 (Heap6) to accumulator
45 LSH & A(19),     -- Left shifting accumulator 19 times
46 CAH & A(7),      -- Copy accumulator to sys0 (Heap7)
47 CHA & A(0),      -- Copy v1 (Heap0) to accumulator
48 CMP & A(7),      -- Compares accumulator with sys0 (Heap7), sets flags for branch instruction
49 BRG & A(51),     -- Absolute jump to start of next Else, if greater than flag is set. Codeline 51
50 CHA & A(6),      -- Copy Constant = 1 (Heap6) to accumulator
51 CAH & A(3),      -- Copy accumulator to dir (Heap3)
52 JMP & A(7),      -- Absolute jump to start of Loop. Codeline 7
53 RST & A(0),      -- Resets program counter to 0
54

```

Optimize Heap

Tries to remove unnecessary writes and reservations of heap elements, this is a twofold process one that is run during the incremental processing of the instructions, which removes unnecessary system variables by determining if they are used before being overwritten by another instruction. And a process that runs after

all other optimization has completed to ensure that no data is placed on the heap if it is never read.

The first of the two parts iterates forwards from the instruction and sees if the heap element is used by before it gets over written, this like the two previous operations traverses the blocks that follows the to ensure that no instruction is removed unintentionally.

The second part of this optimization runs through all the instructions from an end, and for every write to the heap, it ensures that the heap address is read again at any point in the code, this is achieved by iteration over all of the code a second time from the start to ensure that even if the code loops the instruction can be read.

3.1.4 Generator

When the instructions have been decoded and if need be optimized, the output files are generated. These consist of a VHDL file containing the entire QDSP with instructions, and a file containing only the decoded instruction list. In the case that run-time programming has been enabled two more files are created, a binary file containing the program data to be written, and a python script that can program the DSP.

VHDL file

The VHDL file generated contains the entire DSP, with a fully populated program memory and heap. The instruction-set of the DSP is optimized to only contain the needed instructions and only with the heap and program memory sizes needed to contain the program. If the DSP is configured to be run-time programmable the entire instruction-set is implemented and the size of the heap and program memory can be configured.

Binary file

The binary file is generated based on the file structure defined in section 2.2.2 and the configuration parameters set in the program.

Python script

The Python script decodes the binary file generated and sends the heap and program data to the DSP via μ TosNet or Unity-Link. The support for either μ TosNet or Unity-Link depends on the configuration of the program as it adjusts the generated script.

3.2 Validator

In order to help users avoid making common errors validation code has been written that leverages Eclipse's syntax highlighting, that runs while the user types, in order to inform where errors or warnings are found. The validation return errors if the user tries to divide with a number that is not a power of two, uses the same name on several elements, or tries to write to a read only memory address. Less critical errors are highlighted with warnings these are, unused variables, variables used before they are assigned a value and if configuration parameters that are only used when run-time programmable are assigned when run-time programmable is not enabled.

Testing

4

In order to determine if the redesigned QDSP and compiler remained compatible to the previous version and generated correct results a test-bench setup was done in Xilinx. In order to do this an existing QDSP program was ported to the new language, and the two were provided the same inputs and the resulting outputs were compared. The two tested setups almost the same results, though a small discrepancy between the calculated PWM signals did present itself. This was tracked down to be a rounding error in the old design, as a manual calculation of the result agreed with the new design. The data of the test can be found in appendix B.

Conclusion

5

During the project a DSL, with accompany compiler have been designed and implemented. This compiler is capable of creating both a VHDL module containing the QDSP and external programming files. This QDSP can be optimized to only include the instructions required to execute a given program. The language that has been created resembles C thus making it familiar to anyone with experience in C-like programing languages. The instruction-set of the QDSP has been update to allow for jumps in the program code, allowing for a greater flexibility in the user created code. Further more the QDSP has updated to allow run-time programming whereby users can experiment with the code on the DSP without having to rebuild the entire VHDL project.

During the development of the DSL several features have been rejected due to the scope of implementing them, among these are the ability for the run-time programming to be persistent so the program does not revert when restarting the FPGA; division using a IP core, allowing for arbitrary divisors instead of only power of 2 divisors; and creation of variables within loops of if sentences, that gets freed when the section ends.

Instruction-set

A

Name	Argument	Description
NOP	N/A	No Operation
CAH	Heap address	Copy accumulator to heap
CHA	Heap address	Copy heap value to accumulator
RST	N/A	Reset program counter to 0
LDX	N/A	Test if program loader is requested on memory interface
CAS	Memory address	Copy accumulator to shared memory
CSA	Memory address	Copy shared memory value to accumulator
CAI	Memory address	Copy accumulator to IO memory
CIA	Memory address	Copy IO memory value to accumulator
ADD	Heap address	Add Heap to accumulator
SUB	Heap address	Subtract heap from Accumulator
BDIV	Shift count	Signed binary division using bit shifting
BMUL	Shift count	Signed binary multiplication using bit shifting
MUL	Heap address	Multiply Heap with accumulator
TRN	N/A	Truncate accumulator to 16bit value
MAX	Heap address	Compares accumulator to heap value and set accumulator to the highest of the two
MIN	Heap address	Compares accumulator to heap value and set accumulator to the smallest of the two
LSH	Shift count	Left shifts accumulator the given amount
RSH	Shift count	Right shifts accumulator the given amount
JMP	Program memory address	Changes program counter to specified program memory address
BRE	Program memory address	Changes program counter to specified program memory address, based on compare flags
BRG	Program memory address	Changes program counter to specified program memory address, based on compare flags
BRL	Program memory address	Changes program counter to specified program memory address, based on compare flags
CMP	Heap address	Compares heap value with accumulator and set compare flags based on the results

Testing result

B

