Mapping of a DAB Radio Decoder to Homogeneous Multi-Core SoC

A case study to evaluate a NLP based mapping flow

Master's Thesis
by

Berend Dekens

Committee:
prof.dr.ir. M.J.G. Bekooij (CAES)
dr.ir. A.B.J. Kokkeler (CAES)
J.H. Rutgers, M.Sc (CAES)

University of Twente, Enschede, The Netherlands
March 20, 2011

Abstract

The race for higher performance in computer processors came to a halt when increasing clock speeds was no longer possible due to the increasing importance of at least two factors: the increasing gap between CPU and memory speeds (memory wall) and the trend of exponentially increasing power with each factorial increase of frequency (power wall). Instead, the industry created multi-core processors to increase processing power. The use of multi-core processors is less than trivial as their efficient usage requires new languages for easy parallel programming.

In this thesis we will evaluate the Omphale Input Language (OIL) and its tool kit by means of a case study. OIL is a language to describe a Nested Loop Program (NLP) and is meant to assist with writing parallel programs by providing a 'coordination language'. A coordination language is the 'glue' to combine blocks of side-effect free code (called tasks) written in another language.

OIL has some aspects of a functional language, allowing it to be converted more easily into a task graph. The synchronisation in this task graph can be modelled in a Cyclo-Static Data Flow (CSDF) model. This CSDF model can be used to compute scheduler settings and buffer capacities such that real-time constraints can be met. Furthermore, it can be used to reduce scheduling and synchronisation overhead. The model can be used to calculate communication channel parameters to guarantee throughput and latency and guarantee the absence of deadlock.

OIL provides the means to write or convert sequential programs written in another language to a hybrid program which can run in parallel on a Multi Processor System on Chip (MPSoC) architecture. However, no experiment has been carried out to evaluate how practical such a rewriting step is and whether there are fundamental problems that hamper conversion into an OIL program.

In this thesis, we will convert an existing Digital Audio Broadcasting (DAB) decoder algorithm written in sequential C in order to evaluate OIL and its tool kit, 'Omphale'. The algorithm will be converted to be executed on an MPSoC architecture. The used MPSoC system was developed at CAES at the University of Twente within the Netherlands Streaming (NEST) project. The goal of the NEST project is to research and exploit MPSoC architectures

tailored for streaming applications with means for low power, composability, and reconfigurability.

The existing architecture was extended with a network bridge running at 100 Mb/s to accommodate the need for high speed I/O for the case study. The DAB radio decoder processes 10-bit samples at 8 MS/s requiring 80 Mb/s for real-time signal streaming.

A direct mapping of existing functions to tasks of the DAB decoder resulted in a first (naive) partitioning which ran at 2.3% of the required speed for real-time radio decoding. The bulk of the processing time was spent in a signal filter which could not be split using task level parallelism because of dependencies. Instead, since OIL does not support Data Level Parallelism explicitly, a work-around was implemented to execute the signal filter in parallel (16 threads) to reach real-time throughput constraints. We recommended to add proper Data Level Parallelism (DLP) support to OIL and Omphale as its necessity is demonstrated with the parallel conversion of the signal filter in de DAB decoder.

A problem which could not be solved was that the frame decoder was not reaching real-time throughput constraints. While the original data flow design of the decoder gave the impression that components like frequency and time demultiplexing as well as Viterbi decoding could be split into separate partitions, the actual implementation required control structures. This control prevents the partitioning of the existing frame decoder without fully redesigning and rewriting said decoder.

While the original implementation of the DAB decoder prevents Task Level Parallelism (TLP) because of many data dependencies, these dependencies have different origins. We found four types of dependencies during the case study: originating from the standard or design choices and inherent to the algorithm, dependencies introduced by the compiler and dependencies introduced by data or structure sharing.

The dependencies resulting from design choices are hard to avoid. Most of the possible design choices to avoid dependencies result in a quality trade-off where dependencies are removed by sacrificing quality properties. This trade-off makes the design choices impossible without expert knowledge of the algorithm in question. This means that automatic dependency prevention or removal with design choices is not possible. This type of design choice dependencies also prevented parallelism within the DAB decoder: the decoder had to be modified in order to create potential parallelism.

Note that dependencies do not necessarily stem from the language used to describe the algorithm: both imperative as well as applicative implementations will face the same inherent problem with fundamental dependencies.

To conclude, while OIL provides a coordination language to write parallel programs, using existing implementations for algorithms might not result in the level of parallelism that might be expected when the algorithm itself is considered. Some of the dependencies preventing parallelism are fundamen-

tally from the algorithm itself while others are the result of design choices. These problems are unrelated to the type of language they are expressed in.

Contents

CHAPTER 1

Introduction

Since the dawn of computing, processing speeds and processing power have been increasing. The need to go faster came from the hardware industry pushing new products, as well as software developers thinking of new ways to use available processing power. Over the years, processors started to feel the bottleneck of the speed of their peripherals in a computer system and as a result, processors were designed that no longer shared a common clock with the system. Instead, they used a clock which ran a multitude faster than the system clock.

More advanced instruction sets for processors together with ever increasing clock speeds made sure the modern day computer became more powerful with each generation. But after decades of ever increasing clock speeds, a boundary appeared which prevents us from simply increasing clock speeds to boost performance [6].

Using more advanced instruction sets and higher clocks speeds is the fastest solution to improve performance. Faster execution requires no modification to programs. In order to exploit more advanced instruction sets, compilers have to be modified and programs have to be recompiled. Both can yield a performance gain without having to modify the source of a program.

Because of the size of transistors in modern day Application-Specific Integrated Circuit (ASIC) technology, the common 'wisdoms' started to change [2]. One of the old wisdoms was that transistors are expensive and power is 'free'. Nowadays, this is inverted to the assumption that transistors are 'free' and power is the problem: all those tiny transistors together need a lot of power to operate and the additional problem of heat dissipation makes it nearly impossible to increase their operating speeds.

In order to keep increasing processing power, a new direction was chosen: instead of making one processor go faster, systems were created with multiple processors. Multi-processor systems have been around for years but tradition-

ally consisted of physically separate Central Processing Unit (CPU) packages and required special motherboards to house, feed and control the extra hardware. The next step was to mount multiple processors in the same physical package. The added advantage of this structure is the option to share resources between processors and increase their speeds: the on-chip clock is almost the same for all intermediate components, unlike the 'old' multi-processor systems which used slower system buses to transfer data between components. An added advantage was that power usage dropped in this configuration as well [31]. The observant reader might notice that sharing on-chip resource might provide benefits but the amount of resource sharing is increased as well, which increases undesirable sharing effects.

So after the race for increasing clock speeds, a new contest is emerging: a race to fit as many processors and as much memory on a single chip as possible. While in theory processing power increases linearly with the number of processing cores (neglecting sharing effects from sharing resources), actually harvesting this power proved to be less than trivial.

Because humans tend to express their (written) intentions sequentially, most computer languages follow suit. This natural way of expressing oneself works fine as long as the intended target reads the instructions in the same way. When this is no longer the case, for example when using multi-threaded programs running on multiple processing cores, the familiar way of writing computer programs becomes a problem.

Taking a step back, the way computer programs are written by a programmer might be sequentially but the source code is not executed directly on a CPU. The compiler for the programming language converts the human readable form into something a computer can execute. When this conversion is done without any optimisations, the resulting program will, in essence, be a literal copy of the original source.

In most cases it is inefficient to simply translate the source because of hidden properties of the target system the programmer might be unaware of, unlike the compiler. Therefore, optimisations like loop unrolling (removing jump overhead), or instruction reordering to accelerate program execution, are applied.

The problem is that the compiler alone can only exploit parallelism up to a certain point, something which is heavily tied into the type of programming language.

To explain this further, we first define three properties for programming languages. Based on the property, some aspects prevent parallelism:

1. ***Pure declarative:*** the language describes *what* needs to be done, *not how* it needs to be done. It describes the constraints a solution should satisfy but not the steps that should be performed to obtain a solution. An example of such a language is Linear Programming (LP). In LP a 'program' is defined as a set of mathematical equations, for an example

take a look at the equations below where the maximum and minimum value of $z = 3x + 4y$ is requested with the following constraints:

$$\begin{aligned} x + 2y &\leqq 14 \\ 3x - y &\geqq 0 \\ x - y &\leqq 2 \end{aligned}$$

The mathematical formulae define only the constraints that a solution should satisfy, they do not define how the solution is computed. The solution of this example for $z = 3x + 4y$ is a maximum for $z = 34$ with $x = 6$ and $y = 4$ and a minimum with $z = -15$ with $x = -1$ and $y = -3$.

2. ***Pure imperative:*** the language describes the order in which operations must be executed but not their dependencies. Most 'common' programming languages have mainly aspects of this type. For example, look at the example code below:

```
f( );
g( );
h( );
```

Since it is implicit on which data the functions operate, reordering them might not yield the same result as the original program.

3. ***Pure applicative:*** the language describes dependencies between operations but not the order in which they must be executed. Most Functional Programming (FP) languages have this property. Take a look at the code below for an example where in- and outputs are explicitly labelled.

```
f( x, out a );
g( y, out b );
h( a, b );
```

In this example, specifying the statements in a different order will not change anything as the possible execution of statements depends on in- and outputs. As such, f() and g() can be executed in parallel or in an arbitrary order and h() will be executed after f() and g() have completed.

The current problem with multi-core technology is that its still being programmed much like single-core systems: most programs run a sequential ordering of instructions. As such, adding more CPU cores will not accelerate their execution. Even programs which are modified for multi-core systems use a limited number of threads (browsers, office suites, etc). Intel developed in

2009 a 48-core CPU for research purposes [14, 28]. When considering such a system, it is likely that most existing software is not suitable to exploit such a system: most of it would remain unused.

Programs in most common programming languages contain implicit dependencies. Therefore, most common languages have similarities with pure imperative languages. Most commonly used programming languages, like C, C++ and Java, have mostly imperative aspects. The problem as denoted above is that derivation of data dependencies at design time in such languages is often difficult or even impossible. When these problems are circumvented, it becomes possible to execute independent parts of programs in parallel.

The challenge is to find methods to derive data dependencies in languages with imperative aspects. Since derivation of data dependencies is an difficult issue in pure imperative languages [10], we relax this to the search for detecting data dependencies between pieces of code written in a language with imperative aspects - as long as such a block is side-effect free and as such independent, each block can be executed in parallel.

This raises the question which algorithms, that are expressed in a language with imperative aspects, can be divided in such independent blocks and whether this is enough to reach the amount of parallelism needed to efficiently use a multi-core system. Another element which will likely be relevant is whether a program was written with the prospect of being used in a parallel environment or not.

Another aspect which is just as important as having blocks of program code to run in parallel is the interaction or communication between these blocks. Since pieces run on different threads or cores, issues like cache coherency and data consistency in communication need to be addressed.

One solution is to use communication libraries to provide for example FIFO buffers for communication between threads. Another solution is to use a so-called coordination language. A coordination language is a language in which communication between functions is explicit. Furthermore, a coordination language can be restricted in such a way that a corresponding model can be created that is not Turing complete and therefore amendable for analysis. With this analysis a mapping of tasks can be computed that satisfies real-time constraints. Furthermore, the analysis model enables automatic optimisation of the task graph to reduce the synchronisation, communication and scheduling overhead.

In this thesis the coordination language called ‘OIL’ and the tool kit called ‘Omphale’ has been used to convert a DAB radio decoder from an imperative, single-threaded implementation to a multi-threaded version which will be deployed on a multi-core embedded system. The conversion will provide insight in aspects of writing or converting imperative programs for multi-threaded usage as well as evaluate OIL itself.

We define converting a sequential description to a parallel one as adding synchronisation. When we talk about automatic parallelization, we mean the

situation where a sequential description is converted into a parallel one by adding synchronisation automatically.

1.1 Problem Statement

In this thesis, we will attempt to address the following questions by means of a case study using an existing DAB decoder. The case study will be done by mapping parts of the decoder into smaller parts and connect these parts using OIL. The tool kit for the OIL language will then be used to generate a parallel program which will be benchmarked and analysed. Given the experience obtained with OIL during the case study, we will formulate answers to the following questions.

1. How suitable is the use of OIL and the Omphale tool kit for creating a real-time DAB channel decoder implementation out of a sequential C implementation?
2. What methods, tools, and language extensions could simplify the creation of such a real-time application and which issues prevent automatic parallelization?

1.2 Outline

The outline of this thesis is as follows. We first create a perspective on current technologies using an overview of some languages which are based on imperative and/or applicative principles targeted for parallel programming. Using some source code examples we will demonstrate some of their aspects. We will introduce a new language called OIL and compare it briefly with the presented languages.

Subsequently, we present an overview of the used hardware platform and its internal components, from a high level view to the separate components within the design. Given the target application, the platform will be evaluated and extended in order to accommodate for requirements for the case study. In the next chapter, we continue by explaining what DAB radio is and how it works. We then explain how the DAB radio stream is composed and which radio technologies are used.

Next we explain how the original decoder design relates to the actual implementation and what operations were needed to make the decoder suitable for use with OIL. The mapping of the resulting program will then be explained. We conclude by presenting some bottlenecks which were found and solved during the mapping phase.

The mapping is evaluated in the next chapter where we take a look at the resulting performance of the DAB decoder. Besides evaluating the DAB decoder in terms of execution-time and efficiency, we will take a look at the role

OIL played in the result. While the language provided the 'glue' to compose potential parallel segments, the increased complexity by adding another layer to the program might have adverse influences as well.

We end this thesis by presenting our conclusions and will provide ideas for future work based on the results from the case study and its evaluation.

The related work discussion can be found in chapter 2. In chapter 3 the hardware is presented. The explanation about DAB radio can be found in chapter 4. In chapter 5 the mapping of the algorithm is explained, while in chapter 6 the evaluation can be found. Finally, in chapter 7 the conclusions are presented followed by the future work recommendations in section 7.1.

CHAPTER 2

Related Work

As explained in chapter 1, programming languages can be assigned specific properties. Commonly used languages like C have mainly imperative properties: a simple function call can result in unknown data dependencies. Even though it is possible to write programs in a way where dependencies are clear, making it an applicative example, C remains mainly imperative.

A functional language like Haskell is an applicative language. However, constructs like a do-expression in Haskell (which would be an imperative command as it implies order) make Haskell not a pure applicative language.

As stated, most commonly used programming languages are imperative. Since imperative languages tend to match the imperative mood for commands to take action found in natural languages, it is a likely reason to explain the popularity of imperative programming languages.

The disadvantages are very clear as well: since the order of operations are mostly fixed, converting such programs to be executed on multiple cores simultaneously is very difficult. As such, attempts have been made to augment existing languages to provide the programmer new tools to exploit potential parallelism. Another trend is the design of completely new languages, tailored for parallel execution.

2.1 Parallelism Types

Before considering a (far from exhaustive) selection of parallel languages, lets first present some commonly used types of parallelism.

- **Instruction Level Parallelism** is a form of parallelism where multiple independent instructions are executed simultaneously. Most common processors these days support this form of parallelism, this is the case with super-scalar processing [15] and VLIW. While Instruction Level

Parallelism (ILP) provides some increased performance, it is (near) impossible for a CPU to perform dependency analysis. This limits the amount of parallelism which can potentially be gained in threads [32].

- **Data Level Parallelism** is a form of parallelism where identical operations are performed on multiple data elements in parallel. For example, when a loop is used to sum each pair of elements in 2 arrays, this operation can be executed in parallel. Single Instruction, Multiple Data (SIMD) is a form of DLP taken to the extreme in some systems like GPUs.
- **Task Level Parallelism**, also called 'function level parallelism', is a form of parallelism where tasks (or functions) are running in parallel on multiple processors. Tasks could be simple threads, each running on a different processor. For example, most web-servers use multiple threads to service all requests.

While ILP is mostly in the realm of CPUs and compilers, both DLP and TLP are forms of parallelism which can be expressed in a programming language. Some languages express parallelism explicitly while others use implicit parallelism.

We will now discuss some parallel programming languages, starting with imperative languages followed by applicative languages.

2.2 Imperative Languages

OpenMP is a language based on the imperative language C++ and differs only from the original language by the extensions which are implemented using the 'pragma' compiler commands [5]. Using these commands, sections of code can be marked to be executed in parallel. Usually this is done to normal loop constructs, but iterative calculations can be 'collapsed' as well. It it up to the programmer to clearly mark shared and private variables in such sections and failure to do so can lead to unexpected and incorrect results. OpenMP programs result in DLP but all parallelism needs to be explicitly defined. OpenMP is designed to be used on shared memory multi-core systems [3], as are most other languages discussed below. For an example of OpenMP see code listing 2.1 where a function 'foo()' is executed in 4 parallel threads:

```
#pragma omp parallel num_threads(4)
{ foo( omp_get_thread_num() ); }
```

Code Listing 2.1: Example of OpenMP spawning 4 threads to execute a function concurrently

Similar to OpenMP is Habanero Java, another imperative language, [25] which is based on IBMs X10 [12] language: it fully supports the Java language

but some added keywords to the language add support for parallel programming to 'plain' Java. Habanero Java programs are written with explicit parallel sections. The sections are used for synchronisation to guarantee deadlock free behaviour and to keep parallel programming simple (no dangling threads etc). The language provides support for parallel loops and explicit asynchronously executed sub-sections (which can be part of a loop). These parallel subsections are denoted by a keyword and a scope and end as soon as the operations in the scope complete. If desired, a barrier can be added to the program flow to wait for parallel executed statements to complete. Habanero Java hides all thread management and communication from the programmer allowing programs to execute parts in parallel with ease. A drawback from the Habanero Java language is the fact that the parallelism has to be defined explicitly; in essence parts of a sequential program are executed in parallel and as soon as a parallel section is complete, the original thread continues as a normal sequential program. Another aspect is the job of the programmer to make sure that no data conflicts or even data corruption is possible, something that would not be plainly visible in the sequential execution of the operations. See for an example of Habanero Java the parallel numeric integration implementation in code listing 2.2. The 'async' keyword at the for loop results in parallel execution of the loop content. Note that since 'sum' is a shared variable, the *atomic* keyword is required for correct functioning:

```
public double integrate() {
        double sum = 0, step = 1.0 / NSTEPS;
        finish for(int i = 0; i < NSTEPS; i++) async {
                double x = (i + 0.5) * step;
                atomic sum += 4.0 / (1.0 + x * x);
        }
        return sum;
}
```

Code Listing 2.2: Example of Habanero Java to perform parallel numerical integration

Another parallel language which is fairly successful is OpenCL [17, 30]. OpenCL is an imperative language based on C99 and allows the design of so called 'kernels' which are operators to be used on input data and should generate output where needed. A kernel is not much more than a function with arguments and should return one or more values when it is complete; the kernel itself contains no internal parallelism. OpenCL kernels are used as SIMD operators and are for example very useful for image processing where each pixel or group of pixels are transformed in the same way. Although the operations on inputs can be run in parallel using a GPU for example, the controller which dispatches kernels and controls the application is not executed in parallel. Note that the massively parallel execution of kernels on input data means that data dependencies between kernels are a problem: if kernels have

to defer execution until another kernel has generated (part of) their input data, the amount of possible parallelism would decrease significantly.

OpenCL is suitable to be executed on multiple platforms, depending on the platform, it might not be possible to execute multiple type of kernels at the same time. For example, CUDA [19] can only do this for limited number of kernels and only for relatively new hardware [20]. See for a (stripped) example of FFT using OpenCL code listing 2.3, note that only the control functionality is shown; the actual kernel is a (normal) function. In the example below, a OpenCL context is requested for a specific platform and a command queue is created. After the definition of communication buffers and the loading of the algorithm kernel, the kernel is *queued* for execution.

```
context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, ...);
queue = clCreateCommandQueue(context, NULL, 0, NULL);
buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, ...);
program = clCreateProgramWithSource(context, 1, &fft1D_1024_kernel_src, ...);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "fft1D_1024", NULL);
...
clEnqueueNDRangeKernel(queue, kernel, ...);
```

Code Listing 2.3: Example of OpenCL code showing the control functionality

Cilk is an imperative language based on C which relies on the programmer to specify which functions should be run in parallel [13]. This form of TLP requires the programmer to design functions which are side effect free (no modifications to global structures etc). For example, a programmer can start a number of functions in parallel (not necessarily identical functions) and the program flow will simply continue without waiting for the invoked function to complete. As such, the return values for such 'spawned' functions are undefined until an explicit 'synchronisation' is performed. Once the 'sync' is complete, the return values are available for further use by the invoking function.

As a consequence, programmers might reorder their program to 'spawn' function calls as early as possible in a function to benefit from the potential parallelism. Cilk functions are largely identical to their C counterparts and as such, most programmers should be able to modify existing code or write new Cilk functions without detailed knowledge of the internal workings of the language.

Any performance gained from using Cilk implies that functions can actually be executed in parallel and there are not too many dependencies preventing this. In case of programs which are highly data dependant, where for example each function call is performed on the results of a previous function, Cilk can not execute functions in parallel. The resulting program execution will be sequential, just like the original C counterpart. See for an example the Cilk program in code listing 2.4 which calculates a Fibonacci number.

```
cilk int fib (int n) {
        if (n < 2) return n;
        else {
                int x, y;
                x = spawn fib (n-1);
                y = spawn fib (n-2);
                sync;
                return (x+y);
        }
}
```

Code Listing 2.4: Example of Cilk code to calculate Fibonacci numbers

2.3 Applicative Languages

A language that takes a different take is Single assignment C (SaC) [26], a subset of C which is in fact a functional language. Like stated in chapter 1, functional languages are commonly regarded as pure applicative. Because SaC shares a semantic subset, the subset is suitable to automatically determine the dependencies between operations and as such it is applicative (even though C is imperative). SaC was designed to ease the transition from imperative to applicative languages by providing programmers with a familiar language.

The restrictions SaC impose might be a problem for algorithm designers since some programming features which are valid for C are not available in the subset, for example a lack of pointers. This means that re-factoring existing C code to SaC might prove near to impossible without actually rewriting parts. See code listing 2.5 for an example with some simple integer operations.

```
use StdIO: all;
use Array: all;
int main() {
        vect = [ 1,42,3,81,77 ];
        print( sum( vect ) );
        print( maxval( vect ) );
}
```

Code Listing 2.5: Example of SaC performing some simple integer operations

A functional language is much more suitable to use for parallelism since the order of operations follow directly from the data dependencies as expressed in the program. The functional essence of the languages also means there is no implicit note of global state. As a result, all functions are side effect free and can be executed in parallel. The other side of this potential parallelism is that it is possible to keep distributing functions on processors to the level of basic operations such as $A + B$. Since it is reasonable to assume that each 'branching' of a function and synchronisation to combine results incorporates some level of overhead, it might not be optimal for performance to exploit parallelism to the highest degree in a functional program [9].

The balance between fine and coarser granularity for parallelism in FP languages is largely depending on the actual implementation and most likely differs from platform to platform as well as the intended usage. While this holds for all parallel languages we discussed, the applicative aspect allows the derivation of parallelism much more easily. In the end, the balance between cost (synchronisation and communication overhead) and performance gain is a general problem.

See as an example of a functional language the implementation of the Quick-Sort algorithm in the functional language called Haskell [16]. It is possible to write this in a mere 2 lines of code, see code listing 2.6.

```
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

Code Listing 2.6: Example of Quick-Sort in Haskell

Directly related to FP is data flow programming. In data flow programming, a model is constructed which is usually depicted (or conceptually thought of) as a directed graph of data flowing between operations. Data flow machines execute operations as soon as the inputs for said operations are available.

In this model of a program, its data and the operations on them can be seen in most functional languages where each result is computed from inputs when needed and recursively solving these dependencies yields the answer or completion of the program.

Some of the data flow programming languages are largely graphical. An example data flow programming language would be Agilent VEE [1]. In Agilent VEE, input (from both virtual and physical sensors as well as data generators) can be processed by graphically connecting sources to operation blocks and sending output to other operation blocks or actuators. See Figure 2.1 for an example.

The language used in the case study of this thesis is OIL. OIL provides a nested loop program description which is implemented as a mix of applicative and imperative elements. It was designed at NXP to be used in real-time systems and was implemented as one possible target for the NEST MPSoC hardware architecture which was developed at the University of Twente, see also the next chapter. The nature of NLP is a structure where a number of loops iterate over a stream of data. While the loops not necessarily process an unlimited amount of data, making the loop repeat endlessly will result in the processing of an infinite stream of data: stream processing. As such, this makes OIL especially suitable for streaming applications, where the streaming aspect is visible in an infinite loop in the algorithm.

To emphasise the streaming aspect and how it differs from other parallel languages, lets visualize the threading and communication model for some of the languages we just discussed. Cilk for example is using such a model as it explicitly ‘spawns’ off tasks and combines them. OpenCL does something

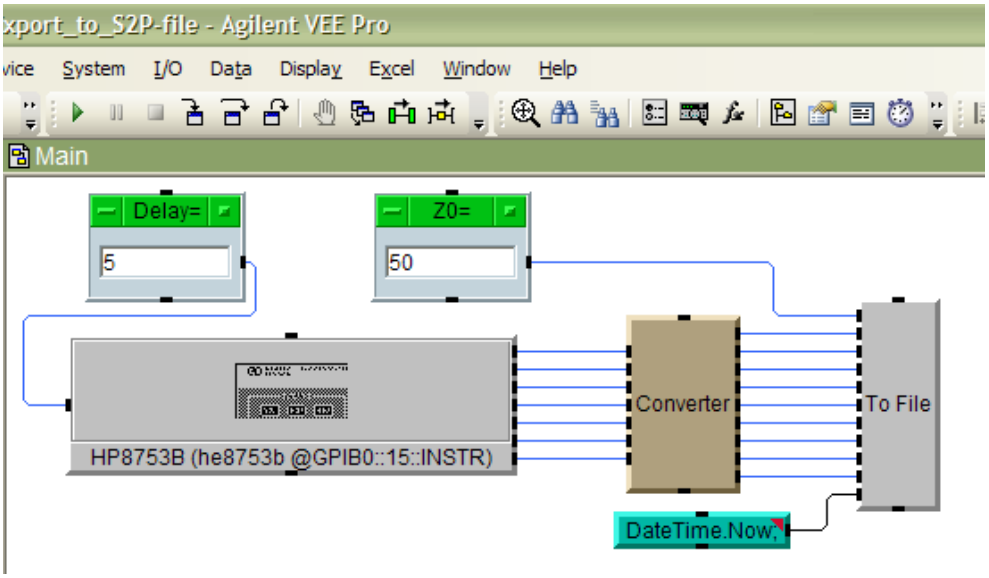


Figure 2.1: Example of a Agilent VEE program

similar and even OpenMP uses something like this. In Figure 2.2 we see one control thread which branches into multiple tasks running in multiple threads. Eventually, the tasks end and the control flow returns to the control thread.

When we consider the model used with OIL, we have a control task to start the program which will then start tasks in parallel. These tasks do not end unless the program is terminated. See Figure 2.3 for an example. This form of 'pipeline' parallelism is clearly different from that of the other languages we just showed. As each task in the pipeline receives input, it becomes active until the entire pipeline is active.

OIL attempts to strike a balance between imperative and applicative or functional languages. The language semantics are similar to C but the actual syntax has some fundamental deviations. The coordination language is used to describe nested loops: function calls (to functions written in another programming language) with explicit inputs and outputs within a (possibly infinite) loop.

The resulting dependencies between functions (based on the in- and output) and conditional program flow are then put in a model which looks mostly like a hybrid solution between one or more data flow models and a state machine. Each function call is considered as an atomic task and as such is run as a task on a specific processor. When we assume a pool of infinite processors, each task will run on a processor and will be activated as soon as data is available.

The strength of OIL is the removal of key problems (like synchronisation and communication) from the grasp of the programmer and provide a tool

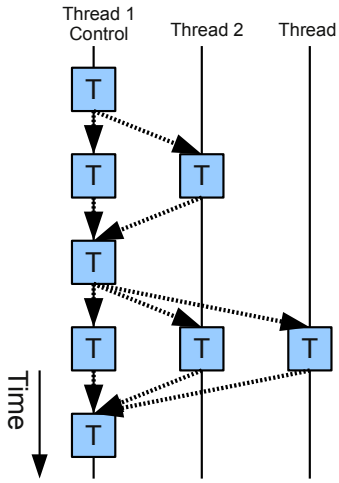


Figure 2.2: Example of the threading and communication model used in various parallel languages

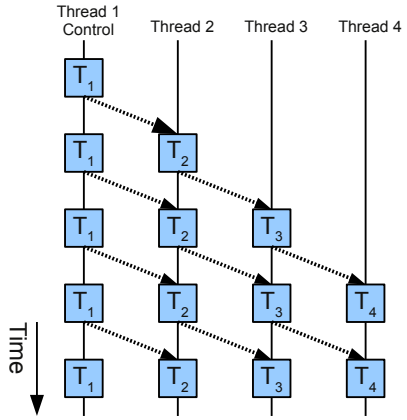


Figure 2.3: Example of the threading and communication model used in OIL

which allows traditional imperative languages to be combined with applicative languages in a way that allows algorithms to be executed on parallel hardware. The internal data flow model is used to derive a task graph with an execution schedule which is guaranteed to be deadlock free.

The main strength of OIL is the prediction of real-time behaviour for the resulting program after compilation. Combined with the worst-case run times of each task, the compiler can devise a possible schedule for the available hardware which is guaranteed to satisfy throughput and latency constraints. Even combinations of OIL programs can be run on the same system with the guarantee that each algorithm will be executed within the boundaries as specified on compile time.

The tool kit used to compile OIL can generate output for multiple target

	Target System	Derived from	Explicit parallelism	Inferred ILP	Inferred DLP	Inferred TLP	Explicit communication	FIFO communication	Mainly Imperative	Mainly Applicative
OpenMP	PC	C++	✓		✓				✓	
Habanero Java	PC	Java, X10	✓			✓	✓[1]	✓[1]	✓	
OpenCL	PC, GPU	C99	✓[2]		✓	✓	✓		✓	
Cilk	PC	C	✓			✓			✓	
FP	PC, others				✓	✓	✓			✓
SaC	PC, GPU	C				✓	✓			✓
OIL	PC, SoC	-[3]			✓[4]	✓	✓	✓		✓

[1] Habanero consists of multiple projects, including the CnC library which handles (amongst other things) communication.
[2] While OpenCL is a SIMD language, the kernels themselves have no parallel structures.
[3] Based on NLP and applicative concepts.
[4] While OIL has no explicit DLP support, this can be inferred by creative usage of the syntax.

Table 2.1: Properties of various parallel languages compared

systems. Among the possible outputs (at the time of this writing) are:

- **plain POSIX:** suitable to be compiled and executed on ordinary Unix systems, normal POSIX threads are combined with a special wrapper for communication between tasks. Note that there is no explicit scheduling involved in this mode: tasks are executed as soon as their input buffers contain data and their output buffers are ready to accept new tokens.

- **SystemC:** The SystemC output allows the algorithm to be executed in a data flow simulator. The simulation provides deterministic temporal behaviour and should exhibit the parallel behaviour which was described in the coordination language.

- **embedded system output:** this output is used on a dedicated multi-core chip with a shared memory architecture. The tasks from the coordination language are mapped onto specific cores in such a way to guarantee their throughput and latency constraints. This target is using a special real-time kernel [34] and uses software FIFO communication between tasks.

For an example of OIL code, see code listing 2.7. In this example the function *f(x, out y)* uses the value of x and produces a value for y. The

function *g(y)* will be executed when a value for y is available. Note that the execution of both functions can be in parallel.

```
def int x, y; x = 0;
while(1) {
        f(x, out y);
        g(y);
}
```

Code Listing 2.7: Example of OIL

2.4 Summary

In this chapter we discussed some parallel programming languages and introduced OIL. The different approach OIL takes has some advantages over other languages, most noticeably the ability to 'embed' blocks of another language while adding parallelism using an coordination language. This attribute will most likely prove valuable in the re-factoring of an existing algorithm if this algorithm was written in a imperative language like C. As the functionality of the original algorithm will still be compiled just like in the sequential version, the efficiency of the resulting OIL program is directly related to the efficiency of the implementation of the original algorithm.

As the applicative property is preferable to the imperative property when we want parallelism, as clear dependencies help in parallel execution, some of the languages we discussed attempt to bring the applicative aspect to an imperative language. While others like OIL or SaC might share semantics with an imperative language like C, they are actually mainly applicative.

In the next chapter, we will introduce the embedded systems platform that will be used with OIL and its tool kit.

CHAPTER 3

Platform

In the previous chapters, we looked at parallel programming languages and introduced the language called OIL, which will be used in the case study. As mentioned before, the OIL language has multiple target platforms. Besides the Unix targets, the OIL compiler, called Omphale, also supports the MPSoC architecture developed at the CAES group at the University of Twente.

The architecture was developed for the NEST project. Participants in the NEST project are Thales, Philips Medical Systems, Océ and Next eXPerience Semiconductors (NXP). The goal of the project is to research and exploit:

1. MPSoC architectures with means for low power, composability, and reconfigurability.

2. A design flow for MPSoC based systems using high level synthesis.

3. An MPSoC run-time system management. By means of dynamic reconfiguration, the run-time system is capable of dealing with adaptive service requirements and platform variability.

While the NEST architecture and Omphale tool kit are developed separately, the 2 systems have been connected to allow OIL based programs to run on the MPSoC.

In this chapter we will first introduce the architecture used for this case study. After explaining the general structure, we will explain which NoC designs are available for this architecture. Afterwards, we will look into 2 of the more complex tiles: the processing tiles and the ethernet tile. We will then explain why ethernet was chosen as a new tile for high speed I/O and present some results from testing the ethernet tile.

3.1 Overview

Within the NEST project a multi-processor architecture was designed for streaming applications. The architecture consists of processor tiles and peripheral tiles which are all interconnected by means of a Network on Chip (NoC). This MPSoC architecture allows the system to scale to increase processing power and add new peripherals when needed.

One of the key aspects of the architecture is its scalability. The NoC is generated based on a simple specification and each processing tile is an instance of a template. The NEST architecture generator can generate a system description (in the VHSIC Hardware Description Language (VHDL) language) which can be simulated and synthesized for the target hardware platform. In this case study we use the architecture on a Field Programmable Gate Array (FPGA) in combination with a normal desktop computer, see Figure 3.1. Note that the processing tiles run at 100 MHz.

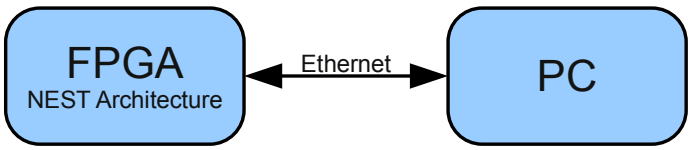


Figure 3.1: Overview of the system set up

Inside the FPGA, we find the NoC which interconnects all tiles in the system. The network provides some guarantees about its behaviour, depending on the type. At least the lower bandwidth limit can be determined, as well as the upper limit on latency.

The processing tiles, labelled *CPU Core #* as shown in Figure 3.2, each contain a single CPU, and are almost identical[1]. Each processing tile has its own peripherals like timers and memory.

While processing tiles have their own private memory, these are very limited in size (mere kilobytes) while most data intensive applications most likely need more storage. The memory tile connects a DDR memory bank to the architecture. Due to the fact that the design is using a distributed shared memory architecture in combination with Memory Mapped I/O (MMIO), the external memory can be easily accessed from each tile. The ML-605 evaluation board comes with a 512 MB memory bank.

To provide feedback and allow input from the outside world, the peripheral tile connects some LED lights and some push buttons to the system. Again, due to the sharing design of the system, these peripherals are available to all tiles.

[1]Each tile and CPU can be configured individually, for this case study all CPUs are all configured identical and some tiles were equipped with profiling hardware.

Taking feedback another step further than a few blinking lights, is the DVI tile. This tile drives the display port on the testing board in order to send a video signal to an external monitor. While connected to the system, this tile uses the memory tile via the NoC to read its pixel data from the DDR memory. In order to produce visual output, other tiles need to write their image data in a reserved section of the DDR memory.

The last tile is an ethernet bridge. The ethernet tile gives an external device, like a computer, access to the entire system. This tile was created for this project and is used to stream radio data from and to the DDR memory.

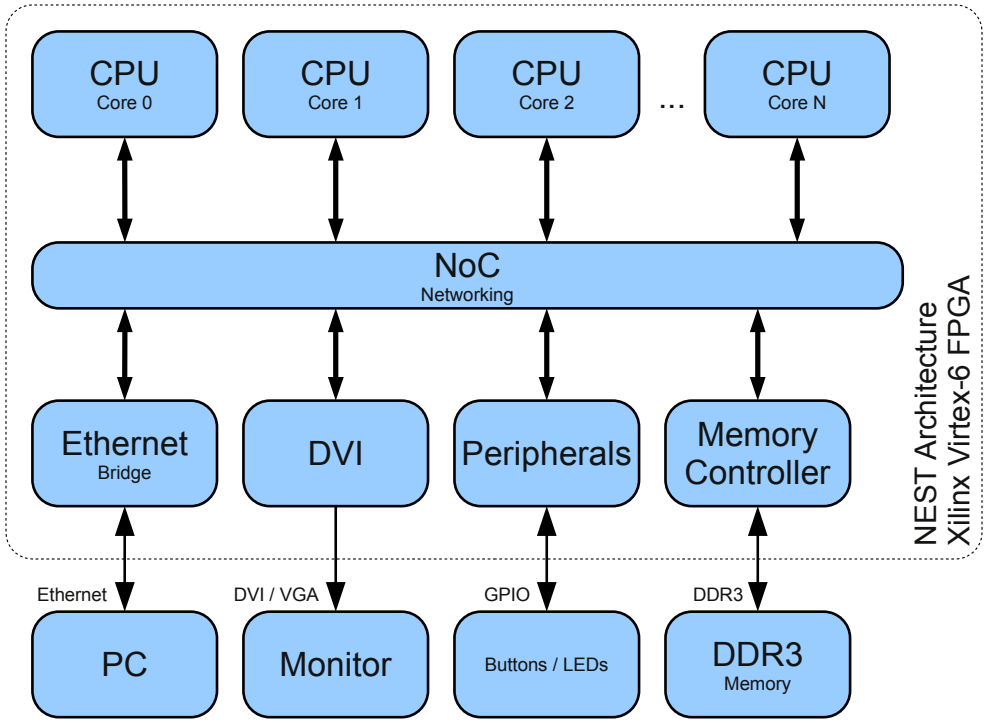


Figure 3.2: Overview of the NEST MPSoC architecture

3.2 Network on Chip

The NEST platform uses a NoC to inter-connect all components or tiles in the system. At the time of writing, the platform generator and the tiles support any NoC which uses Device Transaction Level (DTL) [27] compatible ports to connect the tiles to the network.

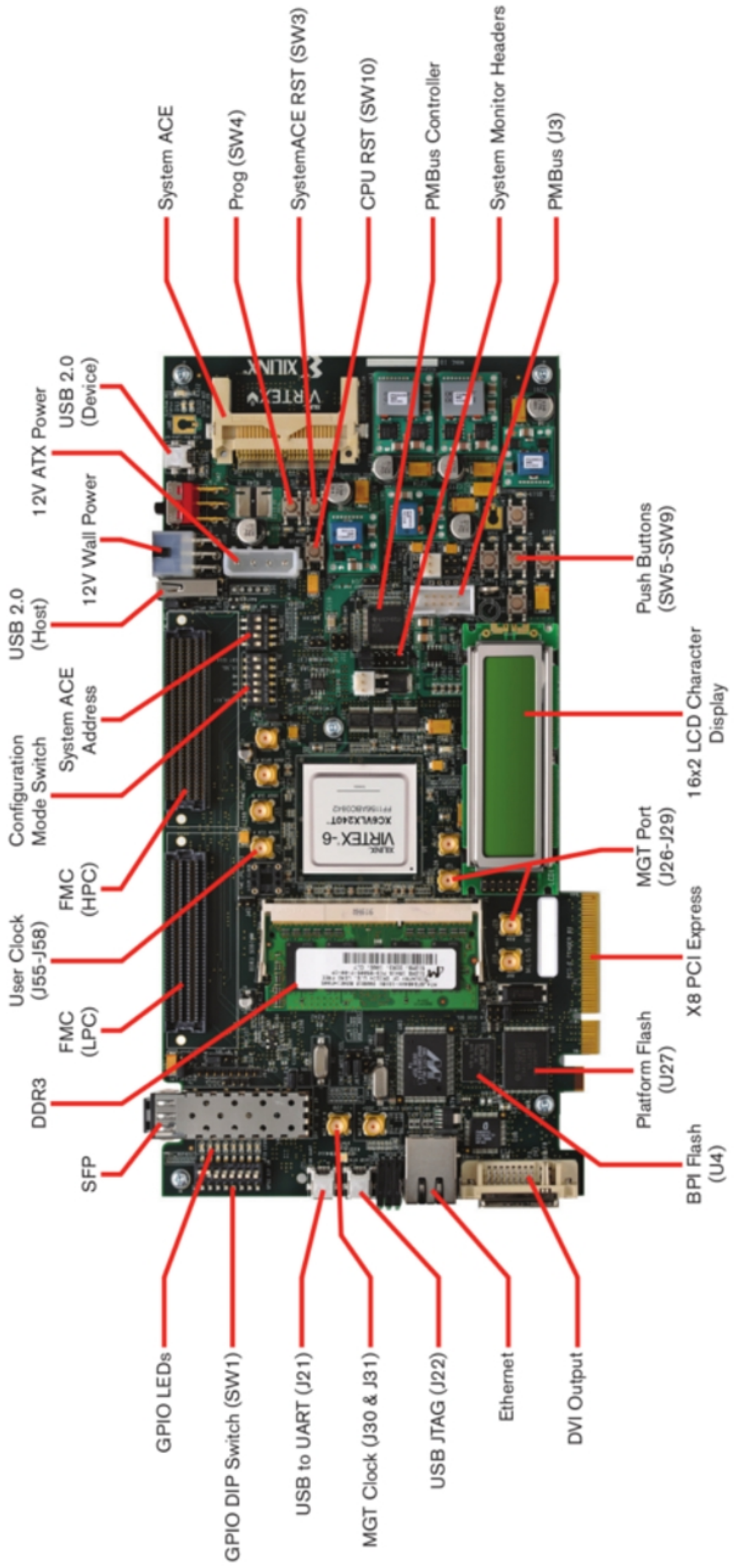


Figure 3.3: Xilinx evaluation kit used with the NEST architecture

3.2.1 Æthereal

Æthereal is a NoC which was developed at NXP as a scalable and highly reconfigurable network to replace traditional bus architectures for MPSoC systems [8]. It consists of routers which are connected to end points and to other routers to form a complete network.

The Æthereal tool kit can generate a network specification based on so called use cases. Depending on the application, more than one use case can be provided to the tool kit. Each use case describes which end point communicates to which end point and specifies the type of service required: for example latency bounds can be set and guaranteed throughput (bandwidth) can be requested.

The resulting NoC is generated as VHDL and is composed by considering the specifications of all use cases and can be simulated as well as synthesized.

A problem with Æthereal is the fact that it is using a connected network. This means that a dedicated point to point connection is provided. As each connection requires buffers and other logic, regardless of the actual implementation of the link itself, each connection is expensive in terms of area. As the number of connections in a multi-core system increases exponentially with the number of cores, the use of a connected network becomes a major issue.

3.2.2 Warpfield

Since in some applications the features and guarantees of Æthereal are not required, a new type of NoC was developed at the University of Twente: Warpfield[2]. While Æthereal uses dedicated point to point connections with configurable bounds for latency and bandwidth, Warpfield uses a connectionless design where data packets are switched and routed as they travel through the network. Also, all peers can communicate with each other. The major advantage of this network is the reduction in area when compared to Æthereal while still providing throughput and latency bounds. To illustrate this: the architecture has been generated for an FPGA for a 32 core system using a Warpfield network, at which point a lack of resources limited the number of CPUs. When using Æthereal, the network itself consumed a significant amount of resources, resulting in a 8 core system.

Warpfield uses fair scheduling and as such the network is starvation free. If a Warpfield NoC is used to connect N devices while the network has a total bandwidth of B, then every device on the network gets *at least* B/N bandwidth, given equally sized transactions. When network peers do not use their bandwidth, unused bandwidth is available to the other peers. This is in contrast to Æthereal, which provides exactly the required amount of

[2]The work on Warpfield is unpublished and therefore only a global description is provided.

bandwidth, but does not allow connections to use more than their requested bandwidth, even if the network is otherwise idle.

The latency bounds are a result from the fair packet scheduling as well. The worst case scenario would be a network under full load where a packet has to wait at each hop. Since the schedule is fair, and the number of hops in the network is limited, for each network configuration the upper latency bound can be determined.

As the entire design works at 100 MHz and the network uses communication channels which are 32-bit wide, the throughput between peers is in the ideal case equal to 400 MB/s but because of protocol overhead throughput ranges from 133 MB/s up to 320 MB/s for bursts.

3.3 Processor Tile

The used processor architecture is the MicroBlaze soft processor core designed for Xilinx FPGAs. As a soft-core processor, the MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs. The instruction-set architecture is based on RISC (Reduced Instruction Set Computing), a CPU design strategy used in many well known CPUs: ARM, SPARC, PowerPC and MIPS. The MicroBlaze has many aspects which can be user configured at design time, for example floating point support.

The processor is supported by a special GCC compiler. In the design used for this case study, each CPU is provided with a Floating Point Unit (FPU) and options like a barrel shifter and a hardware divider. Both the instruction cache as well as the data cache have 4 KB or more memory.

Each CPU has a small dual-port local memory which is used for some of the kernel administration. In general, all data and instructions are stored in the on-board DDR memory (which is accessed through the NoC).

The CPU is connected to its peripherals using a local bus, the Processor Local Bus (PLB). Both caches are connected to the PLB and so is the network bridge which connects the processor tile to the network. Note that a specific memory range is considered 'local' and all other addresses are assigned to the network interface to allow every device which is connected to the PLB access to the other tiles in the network.

A timer is used to fire interrupts to perform multi-threading by means of time slices and to keep track of time.

Finally, each tile has a dual-port scratch memory attached to the PLB which is also connected to the NoC. This memory has an address range on the NoC assigned to it and allows direct data passing between tiles (without using the DDR memory).

Each processing tile runs an embedded kernel which performs memory management and task switching.

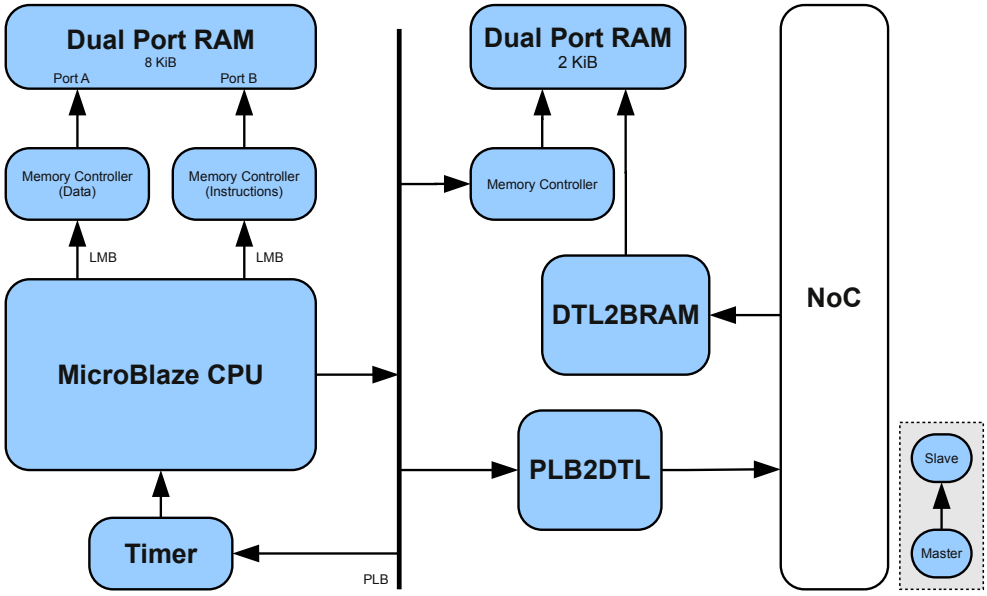


Figure 3.4: Internals of the processing tile

3.4 Ethernet Tile

The case study, which is explained in chapter 4, will process large amounts of data. Even though the platform has DDR memory available, there was no reliable and fast way to perform I/O with the system. In this section we will explain why we chose ethernet to expand the I/O capabilities of the platform and how it was implemented.

Previously, only 2 methods of I/O for the platform were available. Either by means of the debugging interface of one of the MicroBlaze CPUs (optional, the debugging interface can be omitted) and the serial UART port. The latter requires a special initialisation of one of the CPUs in the system with an interpreter program in order to listen and process bytes from the serial port.

While the debugging port is marginally faster, the serial I/O is performed at a 115 kbps. Not only is this transfer speed insufficient when megabytes of data have to be transferred every second, but there is no support for flow control nor extensive error correction either. This makes up- and downloading data using the serial port on the NEST platform an unreliable operation.

The USB debugging interface for the MicroBlaze CPU provides access to a CPU with debugging support enabled but requires to suspend the processor in order to read or write data from the system. In other words: to use a MicroBlaze core for I/O on a streaming platform, the core would essentially be sacrificed to become a dedicated I/O component.

The ML-605 has a programmable Universal Serial Bus (USB) interface

which could also be used for system I/O. The downside to using USB is the fact that special drivers need to be written which makes it an OS specific implementation.

The last problem we encountered was the fact that similar efforts were made to use the USB interface directly with the Æthereal network and only USB Full Speed was supported, meaning the I/O ran at 12 Mb/s.

The ML-605 evaluation board also has support for tri-speed ethernet. The full ethernet driver supports all ethernet modes up to full duplex 1 GBit/s LAN, but also consumes a lot of resources. Xilinx also provided a 'light' driver for the ethernet interface which requires almost no resources on the FPGA but is locked on a fixed network configuration and limited to 100 Mb/s.

3.4.1 Interface Requirements

The new interface tile will be called HSI which stands for High Speed Interface, a relative term based on the bandwidth of serial port it is meant to be replacing (a speed-up or bandwidth increase of roughly a factor 800).

The bridge IP will be designed as a peripheral tile on the NoC with the following requirements:

- The IP is to be designed with speed (throughput) in mind, at least 80 Mb/s is required.
- The IP will use an external interface type available on common computers.
- The IP is stand-alone: it does not rely on other parts of the system (except the NoC).
- The IP is robust: external influences (commands, data) should not interfere with its functioning.
- The IP will be able to access most parts of the MPSoC system, for example by supporting MMIO.

Speed and type

The Xilinx board used for this project is the ML-605 which has numerous I/O connections. Most of these connections are not common on normal computers or would require a lot of effort to connect to a simple computer.

We selected the standard ethernet interface using the 'light' ethernet controller implementation on the ML-605. The controller was set up to use the maximum speed of 100 Mb/s in full duplex mode.

Stand-alone and robustness

Any communication with the outside world needs to be handled by the IP. This means that a form of data handling is required.

The most logical solution is to add a dedicated processor to the IP[3]. This allows data handling and parsing and because the behaviour is defined in software, it should be relatively simple to extend or improve the functionality of the IP.

Robustness can be achieved by keeping the software on the IP minimalistic: no operating system, no multi-tasking. Even stripped down to simple firmware, a network stack should be provided in order to be able to communicate with standard ethernet devices. In order to facilitate this, we created a minimal network stack which provides a partial implementation of a number of protocols, see section 3.5.

System Access

Since most parts of the MPSoC are addressable via the NoC, it stands to reason that the IP should use the NoC as well. This way, most peripherals of the MPSoC are memory mapped for the IP and access to other parts becomes trivial.

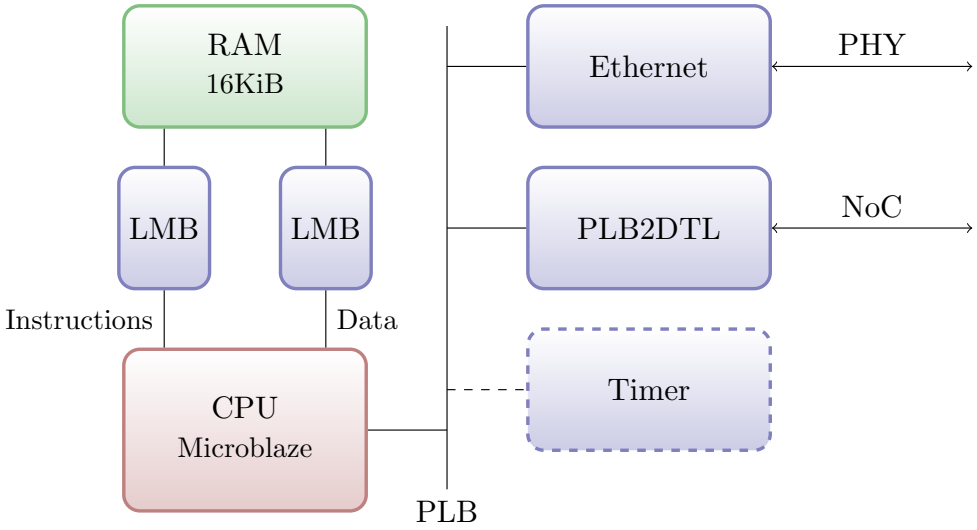


Figure 3.5: Design of HSI bridge

3.4.2 Tile Overview

The original design is created using the Xilinx XPS software. This allowed for rapid prototyping to build the core for the peripheral tile which could be converted to an IP during integration. The result which was integrated with the NEST MPSoC system is shown in Figure 3.5.

[3]Note that unlike with the USB debugging interface, the CPU can be programmed to use spare capacity for other tasks like collecting profiling information.

3.4.3 Throughput

The ethernet network transmits and receives data at 100 Mb/s. This means each bit takes $\frac{1}{100*10^6} = 1*10^{-8}s = 10\ ns$ and each byte takes 80 ns to transmit or receive. Since each packet which is exchanged over an IEEE 802.3 ethernet network contains at most 1538 bytes (see table 3.1), a complete packet is transferred in $1542 * 80\ ns \approx 123\ \mu s = 123 * 10^{-6}s$. Since the MicroBlaze CPU in our system runs at 100 MHz, one ethernet packet is transferred in $123\ \mu s \Rightarrow \frac{1}{100*10^6} = 10\ ns/cycle \Rightarrow \frac{123000}{10} = 12300$ CPU cycles.

In other words: the HSI bridge needs to copy data words (4 bytes) in 32 CPU cycles or less to be able to reach maximum throughput (this includes overhead from loops and protocol handling before the data can be copied).

Field	Bytes (Octets)	Description
Preamble	7	10101010 (0xAA)
Start-of-Frame Delimiter	1	10101011 (0xAB)
MAC Destination	6	MAC address for target NIC
MAC Source	6	MAC address from source NIC
Ethertype/Length	2	< 0x800 = length, otherwise protocol
Payload (Data and padding)	46-1500	Padding appended when < 46
CRC32	4	Checksum, bad frames are dropped
Interframe gap	12	Gap between frames
	84-1538	

Table 3.1: Overview of fields in a ethernet 802.3 frame

3.4.4 Measurements

One of the problems of the bridge design became visible during testing: the maximum speed of ethernet is not reached. To determine more accurately how long data transfers take, multiple performance tests were run to measure the actual times and speeds. To get accurate results, the speed of all data transfers between components is measured individually. The normal data flow is marked in Figure 3.6.

The total throughput is roughly the serial throughput of all components as measured before. Since early tests showed that protocol parsing takes less than 200 cycles and data transfers take thousands of cycles, the overhead of the protocol stack is neglected in the following sections.

Please note that these measurements are performed in a system where other cores on the NoC are practically idle: the processor used for measuring transfer speed is the only active user of the DDR memory. When using applications on the system, resource sharing effects might reduce performance.

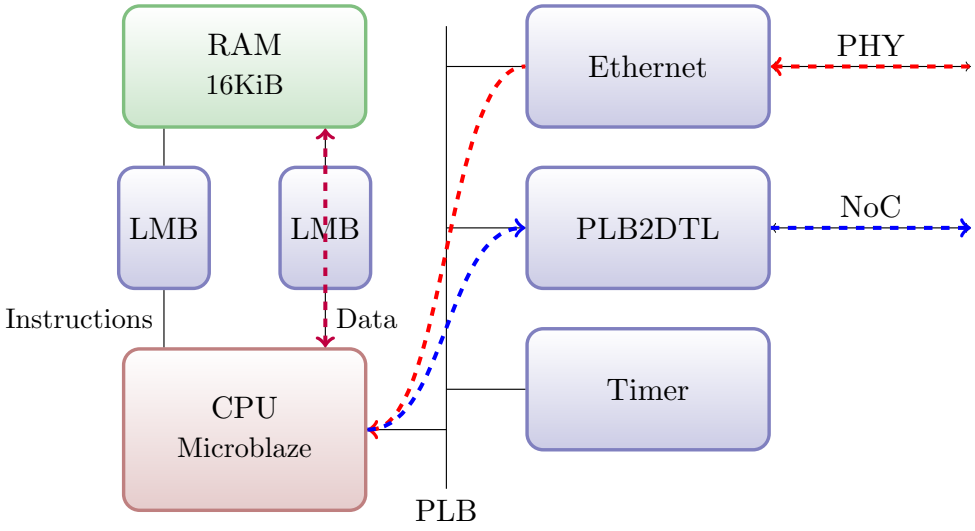


Figure 3.6: HSI bridge with timer and data flow for storing data

3.4.5 CPU to DDR throughput

This test is meant to determine the maximum throughput from the HSI processor to the DDR memory.

A test program is used to generate data (for example, incrementing 32-bit integer values, e.g. the memory address itself) which is stored in DDR memory. This means no data is loaded from the local on-tile BRAM memory since all data is generated internally on the processor. The only overhead for each write is an increment for the memory pointer and the branch for the loop.

The design was altered to include a timer to track CPU cycles, see Figure 3.6 for the system model and the data flow.

Measurements showed roughly $268.4 * 10^6$ cycles for 64 MB of data. Since transfers are done per word (= 4 bytes), exactly 16 cycles per word are required to store data from the CPU into DDR. Since the loop uses a counter which has to be incremented and a branch instruction for the loop itself which takes 3 cycles, actually *storing a single word takes* $\sim$ *12 cycles.*

With a system clock of 100 MHz and 16 cycles per word, this means the maximum throughput or bandwidth is 200 Mb/s from CPU to DDR.

3.4.6 Local RAM to DDR throughput

Extending the performance test from subsection 3.4.5 to include a read from local RAM (the used on-tile RAM is a dual port BRAM) memory. Since BRAM memory has single cycle access times, the expected increase in time is 3 cycles per word: 1 cycle for a word read, a local memory pointer increase

and an 'AND' operation on the local memory pointer to loop (the 'AND' operation provides a mask on the read address since 64 MB is copied and the local memory is only 16 KB in size).

The measurement results confirm this: $318.8 * 10^6$ cycles for 64 MB of data means 19 cycles per word are needed to copy data from local memory to DDR.

With a system clock of 100 MHz and 19 cycles per word, this means the maximum throughput is ∼168 Mb/s from local RAM to DDR.

3.4.7 Ethernet controller to local RAM

One of the final data flow paths is the transfer of data from the ethernet device buffers to the local memory. After data has been moved to local memory, the ethernet device can resume receiving data (note that there are parallel buffers: while at least one buffer is free, data is received from the physical network).

The first measurement was performed while ignoring the actual buffer status and while the network was idle to prevent external influences. The test copied 1500 bytes in 375 word transfers from the first receive buffer to the local memory.

Surprisingly enough, the read of the entire buffer was done in roughly 7,500 cycles. This means a single word is copied in ∼20 cycles, even slower than the storing of words to the external DDR memory!

It seems that the Xilinx implementation of the ethernet device driver did not use 'memcpy' or a similar optimized function but rather implemented a similar algorithm. A few tests showed that memcpy was significantly faster so the driver was modified to perform the data copy using the 'memcpy' function.

The results speak for themselves: the whole buffer of 1,500 bytes was copied in roughly 5,800 cycles. This means a single word is copied in 15 cycles, a speed-up of 25%!

Given the system clock of 100 MHz and 15 cycles per word, this means ∼213 Mb/s can be copied between the local ethernet controller and local memory.

3.4.8 Measurement Summary

Ignoring the protocol overhead, a single word is copied in roughly 34 cycles. This means a theoretical maximum throughput of ∼94 Mb/s can be reached, without taking into account overhead from resource sharing or looping (counters, branch instructions etc). See Figure 3.7 for an overview of the measured performance. The highlighted area is the cross section between the maximum ethernet speed (100 Mb/s) and the minimum transfer speed for the DAB input stream (80 Mb/s). The line at 25 cycles marks the maximum number of cycles for 128 Mb/s for the 16-bit wide samples for the DAB input stream.

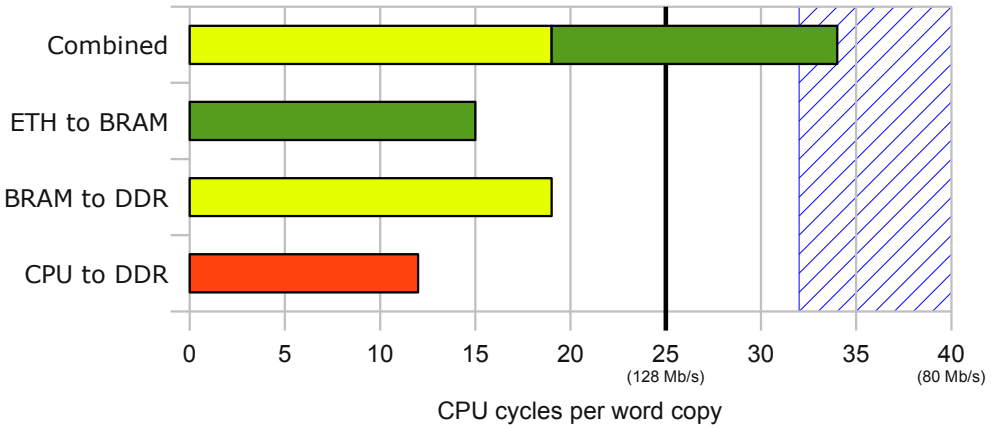


Figure 3.7: Measured cycles required for word transfers (smaller is faster)

Initial testing (without using 'memcpy' to copy data buffers from the ethernet device) showed a throughput of ∼70 Mb/s. If the remote host would attempt to increase throughput beyond that point it would result in packet loss (increasing total used bandwidth but reducing actual throughput as packets would be discarded, resulting in zero gain). After tweaking the software (using 'memcpy' on the initial data transfer for example), throughput was increased to ∼91 Mb/s, without any packet loss.

Since the required bandwidth for our case study is 80 Mb/s in order to be real-time, this performance is acceptable (see also chapter 4).

3.5 Network Stack

One of the problems when building a bridge between a NoC in an architecture and a physical ethernet network is the question which protocol stack should be used. The most simple solution is to implement no protocol stack at all.

The first problem with this approach is the fact that all routing and use of standard IEEE 802 compatible equipment becomes impossible. This means that the most minimalistic viable approach should at least implement the Media Access Control (MAC) layer (IEEE 802.3).

The easiest solution would be to implement a basic protocol stack to be able to use standard network access to communicate with other devices on the network. Since the goal is to use the solution for an embedded and real-time environment, a minimalistic implementation will suffice.

As an example, see the network stack in Figure 3.8. The physical layer handles all of the communication on the electrical (wire) level. Next comes the MAC layer which performs basic addressing and integrity checking by means of CRC32. The MAC layer has a field which determines if the payload is raw data or a protocol like Internet Protocol version 4 (IPv4) or Address

Resolution Protocol (ARP). In case of IPv4 as in Figure 3.8, another protocol is wrapped inside the IPv4 [21] structure. This could be Internet Control Message Protocol (ICMP) [23], Transmission Control Protocol (TCP) [22] or in this case User Datagram Protocol (UDP) [29].

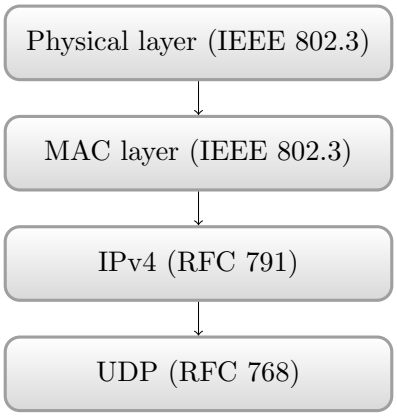


Figure 3.8: Typical network stack for UDP

The protocol stack in the bridge supports a subset of ARP (to communicate its IPv4 address to other devices on the network), a subset of ICMP (to be able to 'ping' the bridge to test connectivity, none of the error handling is implemented or needed) and a subset of IPv4 with basic UDP support. The IPv4 implementation supports non-fragmented frames up to the normal size (1,538 bytes) and the UDP implementation is complete except for checksum checking and generation for payloads (to save CPU cycles).

This minimalistic network stack has a binary footprint of ∼19 KB. As a reference, the complete IPv4 implementation (without 'netfilter') as used in the 2.6.34 kernel of the Linux operating system is ∼1.6 MB as binary. That means the full implementation is over 86 times the size of the embedded stack while the latter is still able to interact with most devices on a network. The hierarchy of the resulting stack can be found in figure 3.9.

3.6 Future Work

In this section we will propose 2 future optimisations which can improve the performance of the HSI tile.

As mentioned before, the ethernet lite core was used to save resources and easily access the network. The full ethernet core has support for transfer speeds up to 1 Gb/s and as such will have much faster I/O buffers compared to the lite core (which currently results in transfer speeds below the maximum ethernet speed). While 1 Gb/s might be a bit much for the rest of

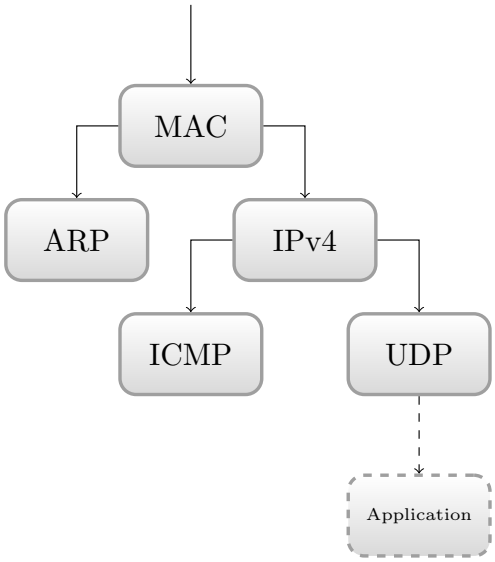


Figure 3.9: Components from the implemented (minimal) network stack

the architecture, the required 16 MB/s for 'uncompressed' DAB signal can be achieved[4].

A second performance boost can come from the use of a DMA controller. Testing showed that a DMA controller only needs 8 cycles per word copy from and to shared memory. This is considerably faster than the 20 cycles it takes the MicroBlaze CPU to write a word to shared memory. Using the DMA controller on local memory as source will mean even lower copy times. This will boost throughput in the HSI tile but will also lower the load on shared memory as the DMA controller will use bursts to write its data, which is faster than separate word writes.

The DMA controller was tested on a (manually) modified architecture. As the DMA controller is not well supported at this time, the actual mapping is performed without it.

3.7 Conclusion

In this chapter, we described the used platform and its components and the extension that was added to the NEST system in order to be able to use the platform for the case study. We explained the choice for implementing a peripheral tile with ethernet support to improve the I/O capabilities of the NEST platform. After evaluating the performance of the new ethernet tile a pair of possible performance improvements are suggested.

[4]The radio signal actually uses 10-bits, packing these bits together results in a signal at 80 Mb/s, while at the full 16-bit per sample 128 Mb/s is required.

A detailed overview of the HSI network protocol and its features can be found in Appendix A.

In the next chapter, we will introduce and explain the DAB radio standard and the decoder algorithm which we will convert to be executed on the NEST architecture.

CHAPTER 4

Digital Audio Broadcasting

In this thesis we will study the mapping of a DAB decoder algorithm onto a multi-core system. In this chapter we will introduce and explain how DAB works.

Digital Audio Broadcasting is a digital radio technology for broadcasting radio stations. Mostly used in various countries in Europe, it was developed in the late eighties in a European project as a replacement for analogue radio broadcasting technologies. One of the key aspects of the standard was to bundle multiple radio streams in one DAB signal and to improve radio broadcasting in terms of spectral efficiency, quality of reception, and in quality of sound.

Both spectral efficiency and sound quality are directly related to the fact that a digital radio technology as DAB uses a digital encoding to transport the sound. The used audio encoding technology is MPEG Audio Layer II, which was a state of the art encoding during the development of DAB and is still widely used in for example DVDs. A problem with using such a digital audio encoding is a result from the fact that the digital DAB carrier has a fixed bandwidth. As such, the number of radio channels available in a single DAB stream is inversely proportional to the bit-rate of the encoded audio, and as such, the audio quality. This resulted in a negative image for DAB broadcasts starting in the 90's in the United Kingdom where the BBC started to reduce audio bit-rates to broadcast more radio channels. This resulted in a reduction in perceived audio quality of DAB which was lower than analogue FM transmissions[1] [4, 11].

Some of the digital technology used in the DAB standard is meant to improve reception. While analogue radio reception quality gradually degrades

[1] DAB supports bit-rates from 64 kb/s for speech, up to 256 kb/s for high fidelity music. Common bit-rates in the UK are 128 kb/s for music stations while at least 192 kb/s is required to improve FM radio audio quality [11].

with decreasing signal, DAB reception remains excellent until the signal falls below a critical threshold. The used transmission technology, called OFDM, is still used today in technologies like WiFi. The use of OFDM results in more robust broadcasts, even in mobile settings, where noise and multi-path distortions are no longer affecting reception. Another benefit is that OFDM transmissions of the same broadcast can be transmitted on the air at the same frequency at multiple, geographically spaced, antennas. In contrast, analogue FM transmissions require that each antenna with an overlap in coverage uses a (slightly) different frequency, forcing mobile listeners to re-tune their receivers as they move from the coverage of one antenna to the next.

The technical base, the physical layer, of DAB is used in newer standards like DMB and DAB+ [36]. DMB adds support for video and multimedia broadcasting, resulting in a digital mobile television platform. DAB+ addresses some of the problems of the old DAB standard: better error correction (Reed-Solomon) to improve reception and a better audio encoding technology, called MPEG-4 audio or AAC+, make DAB+ the successor of DAB.

At the time of writing, the Netherlands are aiming to reach an 80% national DAB+ coverage by the year 2015 using channel 11C. This excludes regional radio broadcasts which will be transmitted using a different channel. See Figure 4.2 for a map of the expected coverage of the Netherlands by 2015, see Figure 4.1 for the current situation. Clearly, after 25 years, DAB radio is still not widely accepted despite the advantages over FM radio.

In the next section, we will explain the transmission technique used by DAB. In the following section, the internal structure of a DAB transmission is explained. Afterwards, we will relate the presented transmission structure to the decoding of it in a schematic overview of a DAB decoder.

4.1 OFDM

Coded Orthogonal Frequency-Division Multiplexing (COFDM), also known simply as OFDM, is a transmission method where instead of one carrier frequency, data is transmitted in several smaller carrier frequencies. The decimation of data in frequency means the data rate in all these sub-carriers can be much lower compared to when one carrier would need to carry all data. DAB Transmission Mode 1 uses 1536 carriers, spaced 1 kHz apart from each other. See Figure 4.3 for a visualisation of OFDM.

Each sub-carrier in an OFDM transmission carries encoded bits for the digital payload. Depending on the exact implementation, this encoding and the amount of bits it represents can differ. All sub-carriers combined during a small time interval is called a *symbol*. By regarding the complex signal during one symbol, phase information can be extracted. This phase information is then used to extract the 2 bits encoded per sub-carrier. This is called Differential Quadrature Phased Shift Keying (D-QPSK) and in Figure 4.4 a

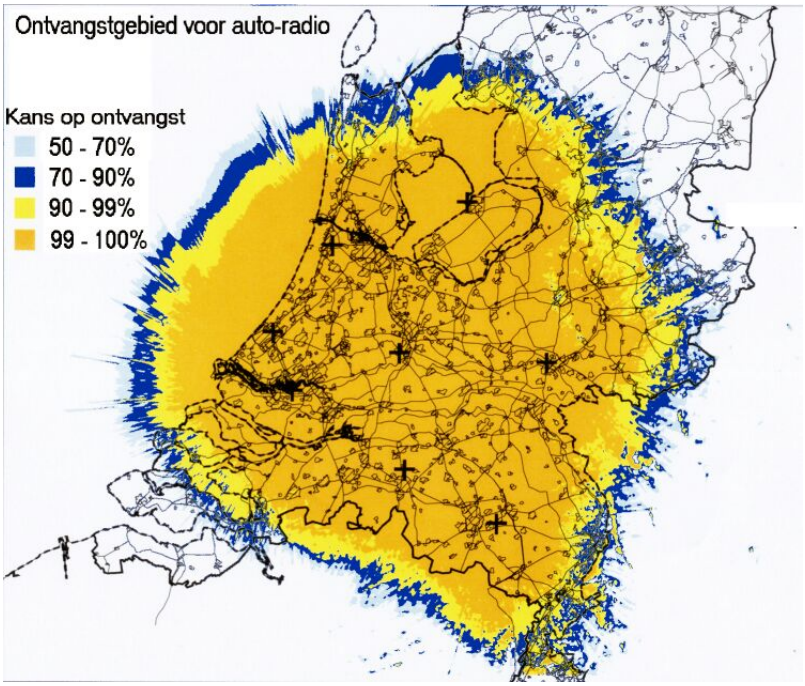


Figure 4.1: DAB coverage area in the Netherlands in 2009 [24]

constellation diagram is shown which can be used to decode the sub-carrier payload based on the complex signal phase[2]. Combining the D-QPSK encoded bits for all sub-carriers yields the data for the entire symbol. When regarding Transmission Mode 1, 1536 carriers with each 2 bits yields a symbol size of 3072 bits, see also Table 4.2.

The advantage of using OFDM for radio transmission is that the length of a single symbol in the DAB stream takes over a millisecond of time. At such speeds, accurately detecting the symbol value is possible. Also, multi-path reception from reflections of buildings and other environmental influences can be easily countered.

Besides data frequency interleaving due to the use of OFDM, time interleaving is also used: this means the data bits are spread over different carriers at different times. The result is that signal loss can be countered for impulse interference or short drop-outs.

4.2 Structure

A DAB radio stream consists of an endless stream of so called frames. Frames are separated by null symbols, where a null symbol is defined as a gap between

[2]The shown constellation diagram is for illustration purposes and might not be the same as used for DAB.

Tabel 6.7 DAB-planning voor kavel 11C

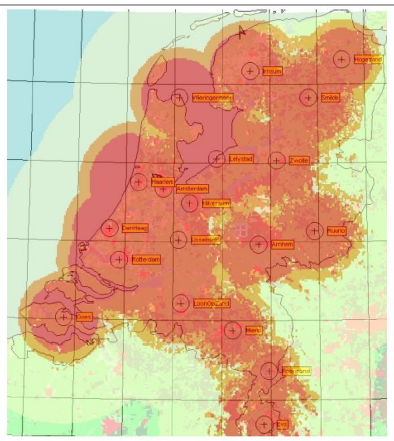
Bedekking	Locatie	EIRP (kW)
	Amsterdam	2.00
	Arnhem	0.63
	Den Haag	0.50
	Eys	1.00
	Goes	0.50
	Haarlem	3.16
	Hilversum	0.10
	Hoogezand	1.00
	IJsselstein	0.50
	Irnsum	1.00
	Lelystad	3.16
	Loon op Zand	2.00
	Mierlo	0.63
	Roermond	0.16
	Rotterdam	1.41
	Ruurlo	1.00
	Smilde	1.00
	Wieringermeer	1.00
	Zwolle	2.00
	Amsterdam	2.00

Figure 4.2: DAB+ coverage area in the Netherlands in 2015 [24]

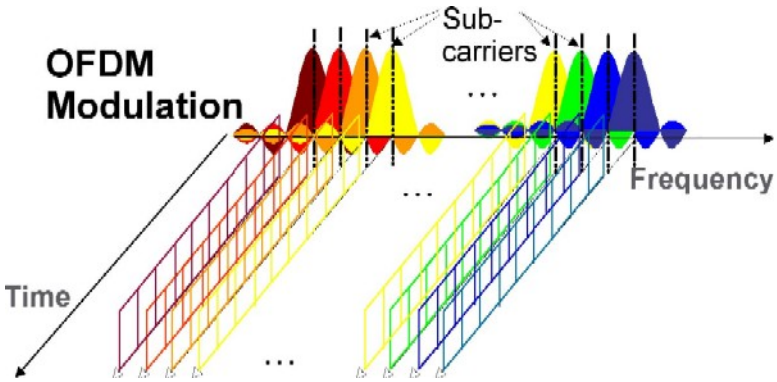


Figure 4.3: Visualisation of OFDM in both time and frequency

frames, in time, with zero energy. A single DAB frame consists of multiple symbols. A null symbol has a different length (in time) compared to a symbol from a frame. The number of symbols per frame is determined by the used Transmission Mode where DAB has 4 different modes. See Table 4.2 for details [35].

A frame in a specific transmission mode consists of a fixed number of symbols. The first symbol in the frame is the so called Transmission Frame Phase Reference (TFPR), this symbol contains no actual data but is used as a reference point to determine the phase information for the remainder of the frame. Following the TFPR, the next three symbols contain Fast Information Channel (FIC) data. This data describes the content of the frame making it possible to selectively decode parts of the frame depending on which audio

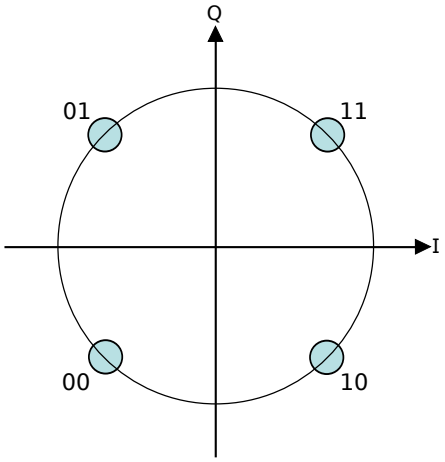


Figure 4.4: Quadrature Phase Shift Keying constellation diagram

channel is requested.

To further protect the data integrity, all the data in the DAB frames are encoded using Viterbi, an encoding scheme to detect and correct transmission errors to reconstruct the original data. Because the FIC is more important than the Main Service Channel (MSC) (a corrupt FIC makes an entire frame useless, a corrupt MSC makes for a small glitch or gap in the audio stream, most likely unnoticeable to human ears), the Viterbi encoding scheme is different for different parts of the frame. As such, de-puncturing is used with the Viterbi encoding to alter the number of encoding bits to encode data bits.

Null	TFPR	FIC 0	FIC 1	FIC 2	MSC 0	MSC 1	...	...	...	MSC 70	MSC 71

Table 4.1: Symbol overview of the structure of a DAB frame

4.3 DAB Decoder

In this section, the structure of the decoder is described at a high level. The decoder actually has 2 states toward producing radio output: tuning and decoding, as shown in Figure 4.5.

The first state is the tuning state. When the decoder begins, it is unknown which DAB mode is present in the IF radio signal. The used DAB decoder can decode all transmission modes and TM1 has the highest bandwidth: 1.5 MHz. To be able to accurately sample this signal, an IF sample rate of 8 MS/s

Parameter	Mode I	Mode IV	Mode II	Mode III
Sub-carriers				
Number of sub-carriers: K	1536	768	384	192
Sub-carrier spacing: Δf	1 kHz	2 kHz	4 kHz	8 kHz
Time relations				
Transmission frame duration: T_{Frame}	96 ms	48 ms	24 ms	24 ms
Symbol duration: $T_{symOFDM} = T_{guard} + T_u$	$1246\mu s$	$623\mu s$	$312\mu s$	$156\mu s$
Guard interval duration: T_{guard}	$246\mu s$	$123\mu s$	$62\mu s$	$31\mu s$
Symbol duration without T_{guard}: $T_u = 1/\Delta f$	$1000\mu s$	$500\mu s$	$250\mu s$	$125\mu s$
Null-symbol duration: T_{null}	$1297\mu s$	$648\mu s$	$324\mu s$	$168\mu s$
OFDM symbols				
OFDM symbols per frame (without null): L	76	76	76	153
OFDM symbols with FIC data:	3	3	3	8
OFDM symbols with MSC data:	72	72	72	144
Transmission frame				
Bits per OFDM symbol:	3072 b	1536 b	768 b	384 b
Bits per transmission frame (without TFPR):	230.4 kb	115.2 kb	57.6 kb	58.368 kb
Network specific parameters				
Maximum f_{RF}:	375 MHz		1.5 GHz	3.0 GHz

Table 4.2: Parameters of DAB Transmission Modes [35]

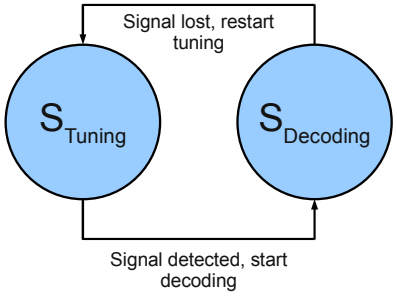


Figure 4.5: Two states within the DAB decoder: tuning and decoding of signal.

is used with 10-bit samples[3]. After detecting a signal and verifying the DAB mode, the decoder goes to the next state.

The next state is the actual decoding state where a previously detected signal is filtered and processed to produce the radio output. Note that the radio output is not audio in a raw form: since DAB uses MPEG audio, the decoder output is a raw MPEG stream and requires further processing before it can be sent to an amplifier or speakers.

Decoding is done in a number of steps and is shown in Figure 4.6. After reading data from the input, which could be buffered data from an ADC, the data stream is split into OFDM symbols, partially filtering the signal, decoding filtered symbols, and processing multiple symbols forming the frame. As the FIC information in the beginning of the frame is required to determine the exact content of the frame, all symbols up to and including the FIC are

[3] 8 MS/s with 10-bit samples results in a data stream of 80 Mb/s, when the 10-bit samples are stored in half words for alignment purposes, the required IF signal uses 128 Mb/s.

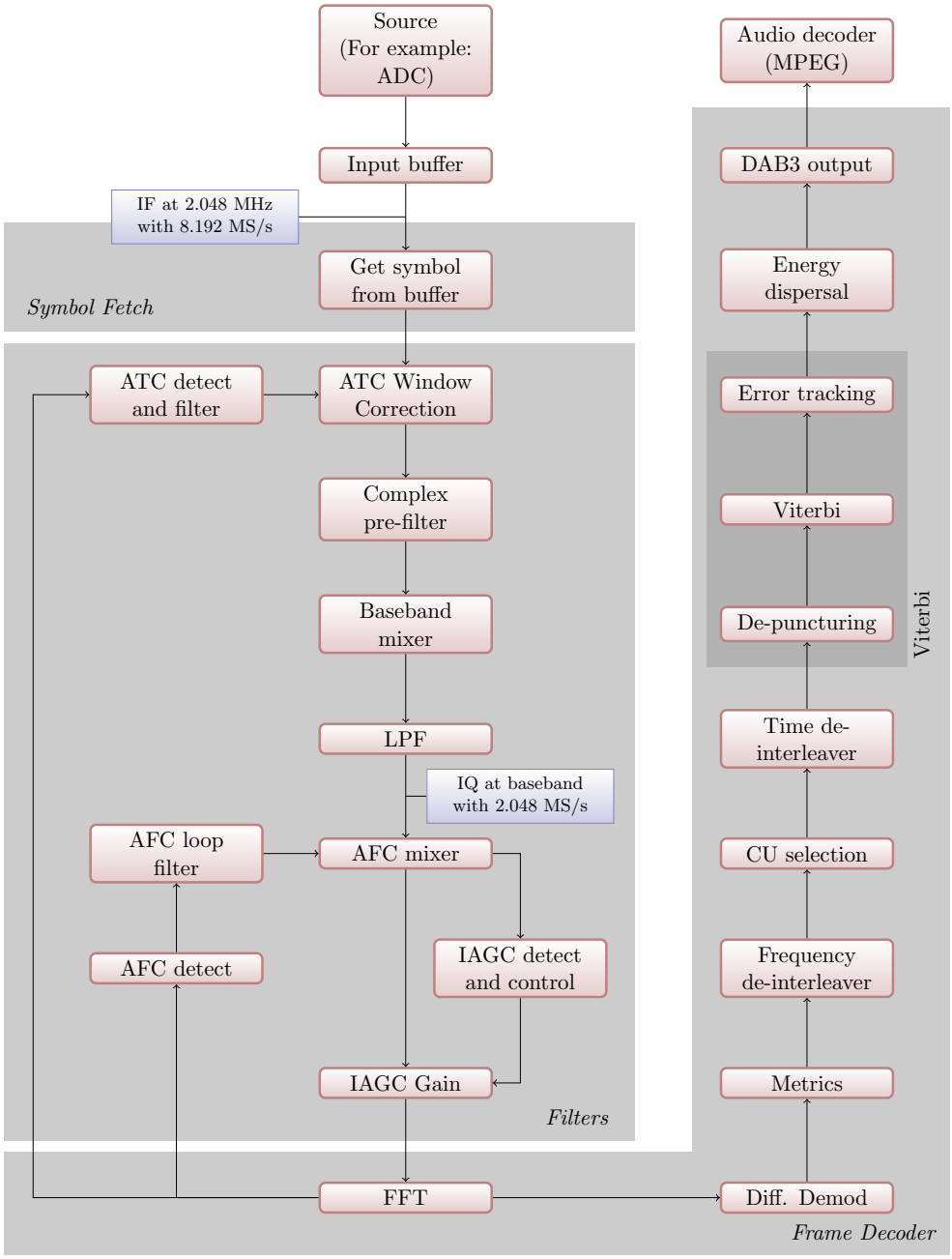


Figure 4.6: High level overview of the decoding of a DAB radio signal

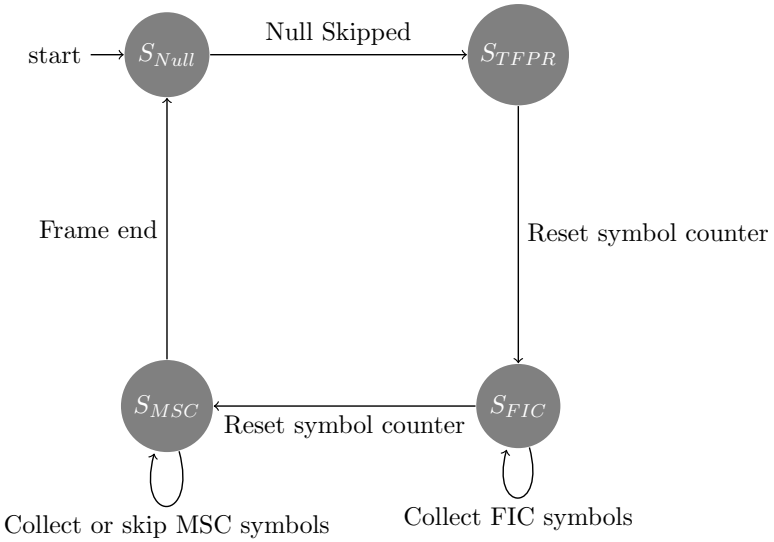


Figure 4.7: Internal state machine used in the frame decoder to handle all symbols

processed and decoded. However, after the FIC information is decoded, the rest of the frame can selectively be decoded depending on which channels are selected. As such, it is not needed to filter and process all symbols in a frame, only the symbols belonging to actively decoded channels will be processed.

The filtering on the radio signal, or rather on the radio signal per symbol, consists of a frequency filter and a conversion from a real valued signal to a complex radio signal. After filtering, the data signal is at the baseband frequency and the bit-rate is reduced by a factor 4 (from 8 MS/s IF to 2 MS/s QI). The filtering step is shown in the signal processing overview in Figure 4.6. Note that while the filtering has multiple components, the 'Complex Pre-Filter', 'Baseband mixer' and the 'Low Pass Filter' are the most computational intensive components in the filters.

After the filters, the symbols are decoded and processed in the frame decoder. The internal components of the frame decoder are shown in Figure 4.6 and in Figure 4.7 the internal state machine for the frame decoding is shown. After skipping the null symbol, the TFPR symbol is processed. As described before, this symbol is used for gain correction, frequency correction and timing correction. Afterwards the symbols belonging to the FIC are processed and decoded. After the FIC, the symbols for the MSC are transmitted. Based on the active channel and FIC information, only required symbols are decoded, the rest is discarded. When a frame ends (the fixed number of symbols per frame is reached), the state machine returns to its starting position. See for an overview of the symbol types Table 4.3.

The DAB decoding itself has more signal processing components which are beyond the scope of this text and as such are shown in a schematic overview

Name	Full Name	Description
TFPR	Time Frequency Phase Reference	Used to perform phase and timing corrections
FIC	Fast Information Channel	Describes the content of the frame (Ensemble / multiplex, see [7], clause 5.2.1
MSC	Main Service Channel	Contents for a specific channel within a frame

Table 4.3: Frame Decoder State Machine information

in Figure 4.6. As an observation, the decoder overview only shows all signal processing in a schematic decoding pipeline but has no notion of control. As such, the addition of control results in source code which is more complex than the rather simple pipeline from Figure 4.6. This control is one of the reasons that the actual conversion of the decoder into tasks is not as fine grained as might be suggested by the overview.

In the next chapter, chapter 5 we will explain how the decoder structure was partitioned for use in Omphale and in chapter 6 we evaluate the experiment.

CHAPTER 5

Mapping

In this chapter we explain the partitioning and the mapping of the DAB decoder, which was explained in the previous chapter. We will explain in more detail about some of the major components within the decoder and how they relate in the actual implementation. During the conversion, we modified some components in order to improve their performance and remove some dependencies in order to obtain the independent behaviour we would like to have for use with Omphale.

In the next section we will define how we ran measurements on our system in order to obtain insight into runtime behaviour. In the following section we will use this measurement approach to determine runtime performance of the decoder and we will explain how the existing decoder was partitioned. After the separation in partitions, we will present the OIL program used to combine the segments into a DAB decoder which can be executed on the multi-core architecture. We will summarize this chapter and highlight some observations from the mapping process in the last section.

5.1 Benchmarking

From this chapter on we will present benchmarking or measurement results. Benchmarking is done by means of the local time source (on-tile timer): when a function begins, the current time stamp is stored and the function is executed. When the function ends, the new current time is retrieved from the local time source. The difference between those two time stamps is calculated and the result is the runtime of a function. When we mention total run times, the summation of all benchmark times of a specific function is presented. Note that the logic to determine the difference in time stamps and summation of run times is not part of the benchmarked times.

Because the embedded kernel performs task switching and measurements are done from within a task, we modified the mapped DAB decoder for testing to execute a single task per CPU. In this manner, we only measure elapsed time for the execution of a single task, even if a context switch occurs. We assume that tasks have large execution times and as such, the time spent in the kernel context switch is insignificant in our measurements.

In order to obtain temporal information about the embedded system and the program execution, we developed a tracing library which allowed automatic benchmarking and provided an overview of individual run times of functions as well as total execution times. In order to do this, the start and stop times of specific functions are stored. At a certain point (for example, when the test data was processed), the collected time stamps are converted into a Value Change Dump (VCD) trace which can be downloaded to a computer for analysis.

5.2 Partitioning

As explained in the previous chapter, the DAB decoder consists of numerous processing components which are merged together in the DAB decoder. In Figure 4.6, components are grouped together in three partitions. These 3 partitions will become the tasks that we will be using in the OIL conversion of the DAB algorithm. These tasks will be used in Omphale as nodes in a CSDF graph. Each node or task communicates via tokens[1]. We will now describe the implementation of the partitions.

5.2.1 Symbol Fetch

The symbol fetcher processes a radio signal received from a data read function and buffers a partial symbol when the buffer does not contain a complete symbol. The data is read from a buffer in the external memory. The symbol fetch will return a symbol when a partial symbol from the input data combined with the partially buffered symbol form a new symbol. When no data is buffered and the input data contains at least one complete symbol, the symbol fetch will copy the symbol into its output. To optimise this logic, there is feedback to the function providing the data. By balancing the input rate, the function can effectively fetch a single token each time it is started, without having to buffer (and thus duplicate) data.

The symbol fetch consists of relatively simple logic which operates on a signal stream with a high bit-rate. The input stream is a sampled radio signal at 8 MS/s with 10-bit samples. Since 10-bits values are not byte aligned,

[1]While Omphale works with tokens for communication, the tokens themselves are language variables with a fixed size. This means a token can be a number (integer) or a complete structure.

they are stored in 16-bit half words. The resulting input stream requires a bandwidth of 128 Mb/s for real-time processing.

The DAB standard uses a signal gap between frames, which is larger than a single symbol (see also the previous chapter and Table 4.2). Since adjusting signal drift can be corrected best when no data is being processed, the null symbol transmission is used to this end. The required information about time, gain and frequency is determined in the frame decoder. This means that the frame decoder, the end of the decoding pipeline, is feeding control information about the next symbol to the symbol fetch. This cyclic dependency inhibits any parallelism and is visible in Figure 5.2.

To solve this problem, we used information produced in several iterations before the current one, rather than in the previous iteration. This effectively delays the arrival of information between tasks and allows parallel execution with appropriate sized delays. When considering time, frequency, and gain correction, we assumed the signal quality would be slightly reduced when delaying this feedback information. The information to correct small deviations in the signal reception should be available for the current frame after the first symbol, the TFPR, is decoded. The information about the next symbol type in the stream (fixed size or variable size) can not be delayed. However, since a Transmission Mode uses a fixed number of symbols per frame, we assumed it would be possible to predict all required information about symbols in advance.

Instead of producing information for the next iteration, the function would produce information (for the symbol to be processed) for the future iteration that will consume the information. Of course, this requires the delay between production and consumption to be known and fixed. When we look at the example in Figure 5.1, we see multiple tokens to allow tasks to be executed in parallel. When we consider the n^{th} execution of a task the n^{th} iteration of the graph, we see that every token produced by T_3 in the n^{th} iteration will be consumed 3 iterations later, in iteration $n + 3$.

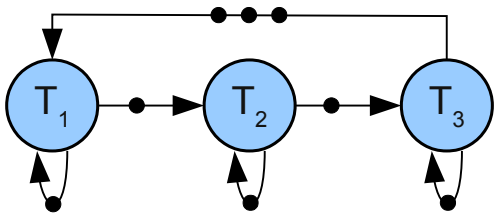


Figure 5.1: HDF example with 3 tasks and a token delay between T_3 and T_1

We observed that the 'fetching' of a symbol could result in a lot of data duplication and copying: from external buffer to input token, input token to internal buffer, internal buffer to output token. This worst case scenario meant that data would be present in the shared memory at four locations.

While this can be reduced, data duplication is a major issue.

Instead, data should be stored once and pointers should be passed. Processing should be performed in-place, if possible. Currently, the Omphale flow will not exploit in-place processing.

Measuring throughput in our embedded system using a MicroBlaze showed that we could not reach I/O speeds required for real-time DAB signal streaming, see Table 5.1. Note that the theoretical maximum is based only on the measured cycles to load and store a single word while the measured maximum is a complete software implementation. To improve throughput, a DMA controller was added to each processing tile in the architecture to handle the remaining data duplication.

DAB input signal:	16 MB/s
Theoretical maximum MicroBlaze (50 cycles load and store):	8 MB/s
Measured maximum MicroBlaze (memcpy):	6.4 MB/s
Measured maximum DMA controller:	48.8 MB/s

Table 5.1: Transfer speeds for copy operations from and to the external memory

5.2.2 Filtering

The filters of the decoder largely consist of a complex pre-filter (CPF), baseband mixer and a low pass filter. The CPF is an image-reject mixer [18], also known as a quadrature mixer. The CPF generates a complex valued signal from a real valued signal and reduces the sample rate by a factor 2. The baseband mixer shifts the signal from the Intermediate Frequency (IF) to the baseband frequency. Since the previous filters introduced unwanted high frequency components (aliasing), the low pass filter removes these high frequency components from the signal. The LPF does another decimation of a factor 2 on the sample rate. The total reduction of sample rate is a factor 4: at 8 MS/s from a real valued signal to 2 MS/s for a complex valued signal.

The filters are implemented as a large number of multiplications and additions. The original filters had support for floating point arithmetic and supported 'skipping' through the stream for an arbitrary number of samples. Because of the skipping functionality and the fact that any number of samples could be processed, internal state was preserved and reused on each filter run. This effectively meant that the filters could not operate in parallel.

While the use of floating point arithmetic results in very accurate signal processing, the DAB decoder does not require floating point precision. As the input samples effectively held 10-bit information, we modified the filters to use 16-bit fixed point arithmetic instead. The use of fixed point arithmetic resulted in a 37.5% runtime reduction in the filters from Figure 4.6.

Floating point:	$11 * 10^6$ cycles
16-bit fixed point:	$8 * 10^6$ cycles
Speed-up:	37.5%

Table 5.2: Measured cycles of filter execution

While the improved performance was a step towards real-time DAB decoding, time measurements (benchmarking) showed that the filters required over 13 seconds of processing time for a second of radio signal. The resume functionality, which was used to keep track of phase information for the carrier signal, was used solely for filtering and skipping symbols using an arbitrary number of samples on each run. As the conversion to a complex signal was performed by mixing with a 1 MHz carrier signal (sine wave), the period of this signal is 1 μs or 8 samples. The internal state, which kept track of the (phase) offset of the 1 MHz carrier signal, was no longer needed when the filter would be used on an offset (and data size) of a multiple of 1 μs or 8 samples from the input signal. This is because each start of the filter will match the beginning of the period of the carrier wave. For a more detailed description see [35, p.334].

After removing the skipping logic and local filter state, parallel execution of the filters became possible. As filter processing required a speed up in time of a factor of at least 14, 14 dedicated CPUs are required for real-time processing of the filters in the DAB decoder. At the time of writing, the Omphale tool kit does not have proper support for DLP, as such, the filters will not be executed on 14 cores in parallel without modifying the OIL program.

To create parallel filter execution in OIL, we defined 14 tasks, each running its own instance of the filters. We used a switch statement with an iterator to switch between filter tasks. By manually increasing buffer sizes in the Omphale output, we managed to execute all filters in parallel and effectively create DLP.

Note that while the (FIR) filters have an internal state, this internal state is generated from the data it is processing. Since the filling of the filter state is part of the computation time, using the filters on very small blocks of signal data is not very efficient. As the filters are currently used on complete symbols (which contain thousands of samples per symbol), this overhead is currently not significant.

5.2.3 Frame Decoding

The frame decoder holds the components which decode and reassemble the DAB bit stream. Internally, the frame decoder (see Figure 4.7) uses a state machine to match each incoming symbol to a specific part of the DAB audio frame. Depending on the symbol type (TFPR, FIC and MSC), different de-

coding properties are applied. The frame decoder contains the logic for the symbol decoding (D-QPSK), time deinterleaving and Viterbi decoding with de-puncturing. Viterbi is an error correcting technique where multiple bits are used to encode a data bit. Because there are a limited number of possible Viterbi codes for for encoding data bits, bit errors can be corrected by correcting to the most likely correct code.

After symbol decoding, the data bits are passed to the appropriate decoder depending on the symbol type. The TFPR is used to determine corrections for timing, frequency and gain. The FIC is decoded to determine the content of the frame. The active MSC is decoded and raw MPEG data is passed to the output of the decoder.

The control flow of the frame decoder (including the state machine) prevents us from partitioning it into smaller partitions. As a result, all the processing components from the pipeline in Figure 4.6 are executed in the same task. As this task has internal state to keep track of which symbol it is decoding, it is impossible to be executed in parallel without making this state machine explicit in OIL such that it can be pipelined. Without redesigning the frame decoder, it is not possible to perform real-time radio processing.

The resulting task graph from the partitioning is shown in Figure 5.2. The run times in the tasks are measured times from the decoding of a sampled DAB radio stream containing 1.6 seconds of signal. The decoder is decoding one channel. Almost all partitions are executed within 1.6 seconds, resulting in real-time DAB radio decoding. The only partition which is not processed in real-time is the frame decoder, which is 5 times slower than real-time. Note that the DDC filter is executed in 16 tasks, so while the filters require 21 seconds of processing time, they are executed in $21/16 = 1.3$ s.

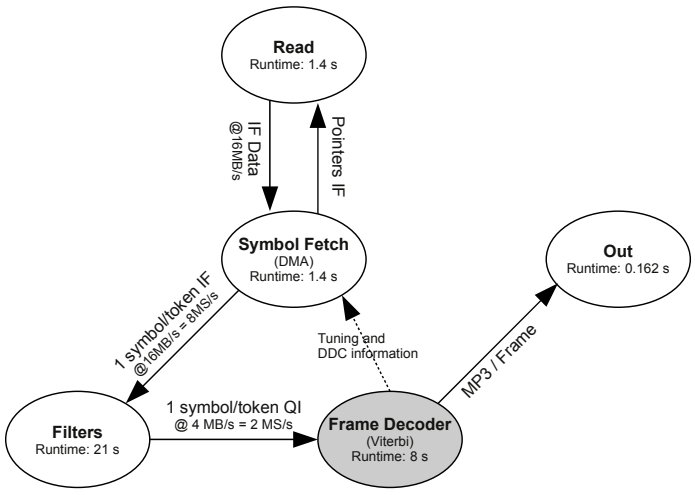


Figure 5.2: Task model of the decoder after modifications for OIL

Sample rate:	8 MS/s
Bit rate:	16 MB/s
Signal found after (processing time):	8.3 s
Data processed until detection:	4.8 MB
Time processed until detection:	0.3 s (4.8 MB / 16 MB/s)
Performance:	28 times slower than real-time

Table 5.3: Measurements of filters while tuning into a DAB signal

5.2.4 Tuning

The tuning state of the decoder, as shown in Figure 4.5, requires the same signal filters as the decoding state. The null symbol has zero energy and as such can be detected without actually decoding the signal. By applying the filtered signal to a Finite Impulse Response (FIR) filter and a Infinite Impulse Response (IIR) filter, an averaged energy value can be established. Because the IIR filter will respond not as much to the null symbol in the signal while the FIR filter will follow the energy curve much faster, this rising and falling between the 2 filters can be used to accurately detect frame transitions. See Figure 5.3 for a visual example.

Using this edge detection technique, the beginning and ending of DAB frames can be detected. Because each mode has distinct frame lengths (see Table 4.2), the length between null symbols can be used to detect the DAB mode present in the signal [35, p.351].

Note that this method of detection requires the *entire signal* to be filtered, whereas the DAB decoding state only partly filters the radio stream. For real-time tuning, 28 dedicated CPUs would be required, see also Table 5.3.

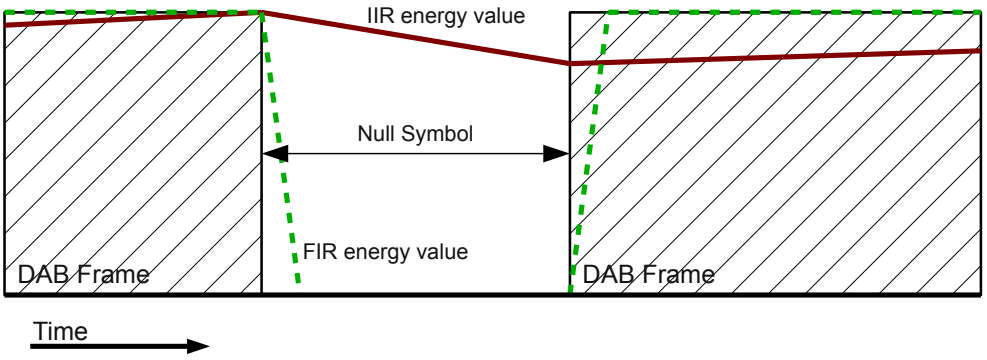


Figure 5.3: DAB null detection during tuning by means of comparing a FIR and IIR filter.

5.2.5 Optimising I/O

As mentioned before, I/O to an external memory in a shared memory system should be kept to a minimum in order to prevent decreased system performance because of a memory bottleneck. At the symbol fetch, we identified a worse case scenario where data would be present in 4 locations in the external memory. We worked around this by only duplicating data when no other solution was possible, and we used a DMA controller to speed up transfers.

As the filters process large amounts of signal data, we measured the number of CPU cycles required per symbol and compared that with the number of reads and writes to the external memory. Our test signal contained a Transmission Mode 1 stream. Using the average measured number of cycles needed to execute the filters for a single symbol, we can derive that the filters spend 3.8% (see Table 5.4) of their cycles for read and write operations. Since the filters needed a 1400% execution time reduction, the 3.8% from the I/O was negligible and the DLP solution was implemented.

Note that we attempted to split the filter components into separate tasks to create TLP. However, the internal bandwidth between components is so high (a multiple of the IF bandwidth) that the external memory became a bottleneck. This resulted in execution times of over 10 times longer than before partitioning of the filters. Another issue was the computational complexity of the 3 filter components: because of the difference in computational complexity, the execution times of the tasks were not proportional to each other. This fact, combined with the fact that the required bandwidth for real-time processing was not available, made TLP in the filters unattractive.

Another problem was found with the use of the GNU C Compiler, or GCC. The so called 'optimisation level' uses an optimisation strategy where instructions are expanded to increase execution speeds. For example, loop unrolling is such an optimisation where a loop is implemented as a sequential set of instructions, without the loop itself. Overhead from branching instructions and keeping track of iteration counters is removed. In our embedded system, the external memory will become a bottle-neck before the processing power runs out. As such, performance of the DAB decoder is a lot better when optimising for binary program size instead of fast execution. Directly related is the fact that the instruction caches will be more effective when functions are more compact.

Finally, as the decoder can skip symbols during frames (for inactive MSC), the decoder was modified to use almost no I/O between tasks by using a marker on tokens to set symbols as invalid (as such, they should be skipped). Tasks will discard these input tokens and mark their own output as invalid as well.

Transmission Mode:	I, see Table 4.2
IF sample rate:	8 MS/s
IF samples per symbol:	8192
Complex samples per symbol:	2048
Total CPU cycles per symbol: (measured)	$8 * 10^6$
Cycles single word read: (measured)	32 cycles
Cycles single word writes (measured)	20 cycles
Total time reading:	262144 cycles
Total time writing:	40960 cycles
Total time spent on I/O:	303104 cycles
Total time spent on I/O of total:	3.8%

Table 5.4: I/O measurements from filters during Transmission Mode I decoding

5.3 DAB Decoder in OIL

After describing the partitions of the DAB decoder, we will now explain what the result will look like in OIL. Note that the initialisation of internal state, tuning and decoding are all part of this semi-complete description.

We tested the proposed solution where we delay the feedback information several iterations between the symbol decoder and the symbol fetch. The decoder as shown below does not have this delayed feedback: instead, feedback information is hard-coded into the decoder for a specific test signal. This effectively relaxes the constraints that result from the dependency as indicated by the edge between the frame decoder and the symbol fetch from Figure 5.2.

5.3.1 Program

The DAB decoder is shown in code listing 5.1 on page 51. After initialisation (state 0), the decoder attempts to tune into the signal (state 1). After a signal was detected, decoding is started (state 2). In this program, there are 16 parallel tasks for the real-time filtering during decoding.

Note that OIL has a different concept of scope than most languages do: after the end of a loop iteration, all variables become undefined. If a value should be preserved, it should be set for the next iteration. For example, $a\{1\} = 1$ would assign '1' to a in the next iteration.

Each function has arguments which are explicitly defined as inputs (no keyword) or outputs (keyword *out*). This allows Omphale to determine data dependencies for function arguments.

The result from the OIL program from code listing 5.1 is shown in Figure 5.4. Note how the control flow results in a task graph which is a lot complexer than the one from Figure 5.2. The 16 parallel filters are clearly visible as the 'column' of tasks in the graph.

```
task {
  def io_read_t read;          // Pointers to the input data
  def io_unread_t unread;      // Unused samples are fed back into the read
                               // manager to be recycled
  def output_buffer_chunk obc; // Output data from the decoder (when tuned
                               // in properly)
  def Afc_t afc;               // Automatic Frequency Correction
  def Atc_t atc;               // Automatic Timing Correction
  def Agc_t agc;               // Automatic Gain Correction
  def SmbPrc_t smbprc;         // Status of frame decoder (Symbol Processor)
  def Sync_t sync;             // Keep track of synchronization status (whether
                               // a valid radio signal is found)
  def NullDet_t nulldet;       // Status of null detection
  def ddc_buffer_t ddc;        // Data buffer for filters (Digital Down
                               // Converter)
  def smb_buffer_t smb;        // Data buffer for frame decoder
  def Sync_t tmp_sync;
  def int filter;
  // State codes:
  // 0: Initialisation phase
  // 1: No signal is found, use the tuning functions to find the signal
  // 2: Signal is found, use the decoder
  def int status;
  status = 0;
  filter = 0;
  while(1) {
    switch(status) {
      case 0: {
        // State 0: Init I/O and structs
        DAB_Start(out status{1}, out afc{1}, out atc{1}, out agc{1},
                  out smbprc{1}, out sync{1}, out nulldet{1});
      }
      case 1: {
        // State 1: Tuning
        IO_ReadBytes(out read, unread); // Read bytes from the input buffer
        DAB_Tune(out status{1}, read, out unread{1}, sync, out sync{1}, afc,
                 out afc{1}, atc, out atc{1}, nulldet, out nulldet{1});
        loopback_agc(agc, out agc{1});          // Refresh unused variables
        loopback_smbprc(smbprc, out smbprc{1}); // Refresh
      }
      case 2: {
        // State 2: Decoding
        IO_ReadBytes(out read, unread); // Read bytes from the input buffer
        DAB_GetSymbol(out status{1}, read, out unread{1}, out ddc, 0, 0, 0,
                      out tmp_sync, nulldet, out nulldet{1});
        // Run the DDC filter in parallel, since its by far the biggest part
        // in the decoder (even when skipping symbols), this should put us
        // in the ball park of real-time processing
        switch(filter) {
          case  0: {  DAB_DDC_0(ddc, out smb, 0, 0) }
          ...
          case 15: { DAB_DDC_15(ddc, out smb, 0, 0) }
        }
        DAB_DDC_NextFilter(filter, ddc, out filter{1}); // Select the next
                                                        // filter task to be activated
        DAB_SMBDecoder(smb, out obc, afc, out afc{1}, atc, out atc{1}, agc,
                       out agc{1}, smbprc, out smbprc{1}, tmp_sync, 0);
        WriteBlock(obc); // Store the generated MP3 stream
} } } }
```

Code Listing 5.1: Semi-complete DAB decoder in OIL

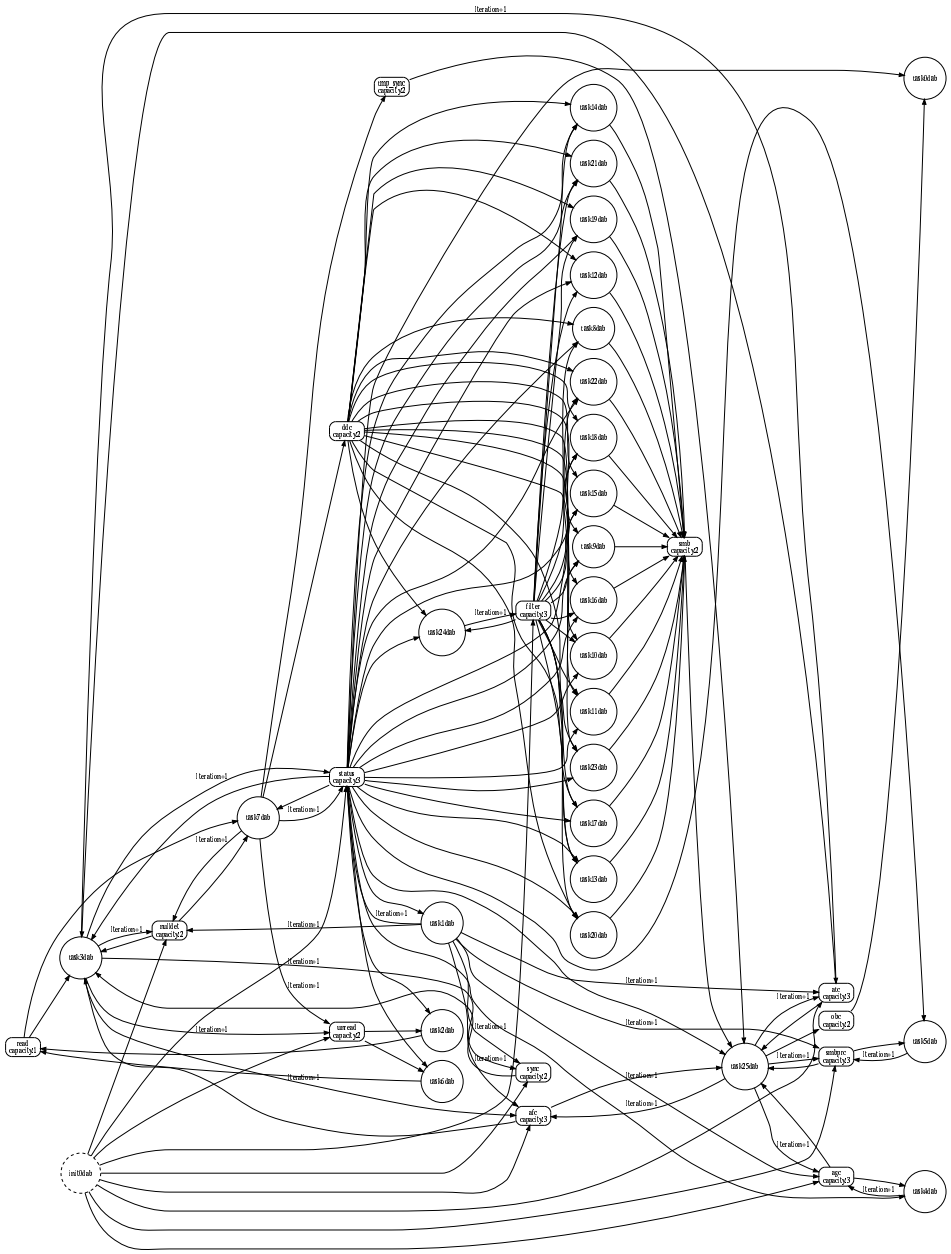


Figure 5.4: DAB decoder with only 3 partitions in a pipeline with filtering using 16 parallel tasks. Note the generated complexity as a result from a small Omphale program.

5.4 Summary

In this chapter we explained the implementation of the DAB decoder in more detail and showed the results of the mapping. The actual task graph from Omphale is more complex than the task graph suggested by the DAB specification. While the basic structure matches, the control flow from the OIL program increases the complexity while the different states, like tuning and decoding, in the decoder add more tasks to the task graph.

We presented some measurement results and some optimisations which were performed to reduce execution times. The only task which is not executed within the real-time time constraints is the frame decoder. Control structures result in data dependencies which are not removable without rewriting the frame decoder.

In the next chapter we will present the evaluation of the DAB radio decoder mapping using the Omphale tool flow.

CHAPTER 6

Evaluation

In the previous chapters, the structure and mapping of the DAB decoder was explained. In the following sections, the experiment will be evaluated. The evaluation is split in two sections: the first one dealing with specific problems with the DAB decoder. The second section deals with the evaluation of OIL in regard to the case study.

6.1 Quantitative Evaluation

The DAB decoder we used for this case study is implemented for single threaded execution. This leads to assumptions and choices which are appropriate for sequential processing, but the resulting dependencies inhibit any possible parallelism. During the conversion of the algorithm, we found four types of data dependencies:

1. Dependencies which are inherently present in the algorithm:
    a) Dependencies which are inherently present in the algorithm definition, e.g. the standard. These dependencies are fundamental and are unavoidable.
    b) Dependencies which are design choices which affect functional behaviour, e.g. quality of reception. At design time, certain choices will introduce new dependencies which will inhibit potential parallelism later on during a conversion. The only ways to avoid these is to rewrite parts of an algorithm or make parallelism a design time goal. These affect the quality of the end result.
2. Dependencies introduced by the compiler. This type can be analysed and can be largely removed or mostly prevented by automated analysis: these dependencies should not be introduced by the tool kit.

3. Dependencies introduced by sharing data structures or memory. While sharing can simplify the implementation of an algorithm or prevent data duplication, it prevents parallelism as data sharing is not possible without introducing new problems.

The first type of dependency is clearly present in the DAB decoder implementation: the signal correction at the beginning of the decoder path is performed using information which is produced by the frame decoder, which is at the end of the decoder path. This feedback mechanism inhibited any parallelism and during the mapping of the DAB decoder we introduced a work-around, as described in subsection 5.2.1. Most dependencies of this type are fundamental and are impossible to remove.

The second type, dependencies resulting from design choices, is hard to avoid. Most of the possible design choices to avoid dependencies result in a quality trade-off where dependencies are removed by sacrificing quality properties like: efficiency, signal quality, etc. This trade-off makes the design choices impossible without expert knowledge of the algorithm in question. This means that automatic dependency prevention or removal with design choices is not possible. This type of design choice dependencies also prevented parallelism within the DAB decoder: the decoder had to be modified in order to create potential parallelism. Note that dependencies do not necessarily stem from the language used to describe the algorithm: both imperative as well as applicative implementations will face the same inherent problem with fundamental dependencies.

The third type is beyond the scope of the algorithm programmer: the compiler should not introduce new dependencies which are not present in the original source code. This type of dependency should be prevented in the compiler itself.

The last type of dependencies are present in the DAB decoder as it is implemented as a sequential algorithm. The sharing of structures allows various components access to data from other components (for example, to trigger a reset in the decoder). The sharing of memory allows multiple functions to perform in-place processing of data.

In order to relax dependencies or to 'fit' an existing algorithm to requirements imposed by the tool kit, modifications of the source code of the algorithm are required. When we move variables and pieces of functionality around without rewriting entire functions, we talk about 're-factoring'. When re-factoring is not possible, it might be possible to rewrite parts of the algorithm. To minimize effort, re-factoring is favourable to rewriting.

6.1.1 In-place Processing

The original design of the decoder used in-place processing for the entire decoder path. While some components have their own internal state, all data

and processing results are stored in the same memory buffer. While this provides advantages for the single threaded execution, like low memory usage and no data duplication overhead, this is unusable for parallel execution. Since each function can be executed in parallel, sharing a memory buffer will have unpredictable results.

As a consequence, we had to replace in-place processing logic with separate processing buffers. This increased memory usage and, in some cases, required the duplication of data. By using data pointers, where possible within tasks, instead of actual data duplication, we managed to keep this new overhead low.

The model of partitioning and spreading of data over multiple tasks, as introduced by OIL, has the potential to prevent in-place processing and introduce overhead when compared to the sequential implementation as data has to be copied. This low-level memory management is an aspect which should not be a problem for the algorithm designer; rather it should be solved by the tool kit.

6.1.2 Structure Sharing

The sharing of data structures within an algorithm allows functionality to cross between components. The difference with in-place processing is that with structure sharing state is shared between functions. The most simple example of this is the reset within the frame decoder of the DAB decoder. When the frame decoder enters an invalid state, it erases all shared data structures, including those from other components within the decoder. The result is a full reset of the decoder.

While structure sharing has advantages for a sequential implementation, it effectively prevents parallelism as multiple tasks updating the same structure will result in unpredictable behaviour. When data from structures can be divided in proper subsets, the division of the data structure will allow parallel execution. When this is impossible, the sharing of structures prevents parallel execution.

6.1.3 Granularity

An important aspect of the conversion is the granularity of tasks. As synchronisation and communication are not 'free', there is a trade-off between granularity and performance. This aspect was visible in a test where the components of the filters from Figure 4.6 were split into several tasks to introduce TLP, as described in 5.2.5.

Besides the increased bandwidth requirements, the components were not equal in computational complexity. This resulted in a couple of time consuming tasks while other tasks were done in a fraction of the time. As the execution times approached those of the overhead from the communication and synchronisation, this approach was deemed to be too resource expensive.

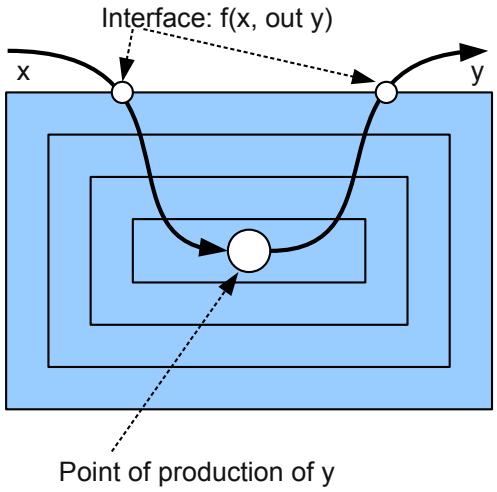


Figure 6.1: Visualisation of multiple layers of control in a program

This trade-off between finer and coarse granularity is a decision which can not be made without defining a cost function for the problem.

6.2 Qualitative Evaluation

As explained before, OIL provides the programmer with tools needed to convert a sequential description into a parallel description. While the language helps exploiting potential parallelism, the fact that we are using a sequential implementation will not help in the search for parallelism within the algorithm. Aside from implementation related problems, the use of OIL introduces new limitations and problems, as we will discuss next.

6.2.1 Interface Limitations

OIL uses an interface model with functions and arguments. To be able to analyse the described program model, the language has to explicitly describe all communication between functions. As a result, there is no flexibility in the in- and outputs of functions. See Figure 6.1 for a visualisation.

For example, if a function operates on input tokens containing radio samples, the function is expected to process all data from the token: as the function ends, the data is no longer available. If not all input data is fully consumed, functions will need to implement internal buffering mechanisms to duplicate data for future use. On the other side, if a function can produce multiple output tokens given its input tokens (and internal buffering), it can not produce more tokens than specified in the OIL function interface. The same holds when there is not enough input data for an output token: the

function is still obligated to produce output tokens as defined in the function interface.

This fixed rate, synchronous data flow program model as described by the OIL code, may require additional logic to overcome the problems with token rates. The additional logic is required to modify control flow in order to abort and resume processing depending on the situation. In some situations, the underlying control logic might be too complex to modify and a full re-write of logic is required.

An example of this problem can be found in the symbol fetch at the beginning of the decoder path. The original symbol fetch would find and then process multiple symbols by simply calling the processing functions as it looped over the input data. As a result, the loop in the symbol fetch would process an arbitrary number of symbols. By altering the structure, the symbol fetch was modified to abort after a single symbol was found. The unused signal data was then buffered internally. To prevent buffer overflow, the number of buffered samples is fed back into the 'read' function. The next input token would then be partially filled based on the number of previously buffered samples. This balances or limits the input bit-rate and prevents buffer over- and under-runs (see Figure 5.2).

6.2.2 Communication Limitations

To be able to properly analyse communication in the program model, OIL requires explicit definitions of communication parameters. This also means tokens have a fixed size.

When dealing with structures which are formed using pointers, for example tree-like structures, the exact size is unknown at compile time. While in some cases a maximum structure size can be introduced by the programmer, this would require application knowledge.

We worked around this limitation by making the tree structures part of the internal state of the function. Combined with the knowledge that a single task is scheduled only on a single CPU, we avoided the passing of pointer structures between tasks. This 'solution' does violate the assumption that partitions do not have implicit state[1]: executing the function in multiple tasks will result in unpredictable behaviour.

6.2.3 'Hidden' Dependencies

Another problem which was encountered when adding more cores running the filter function is the fact that the syntax of OIL, when using conditional constructs, implies that 'parts' of the OIL program can be disabled during a loop iteration. When attempting to introduce DLP in the filtering phase,

[1] Note that this differs from 'side-effect free' which means that partitions can not influence each other.

the amount of parallelism was limited to only two concurrent filter instances. Analysis showed that the limitations were not introduced by the buffer sizes but rather because the complete data flow graph contains all tasks, even when these tasks are 'disabled': inactive tasks loop idly while flagging the input and output buffers. The buffers are constructed in such a way that at compile time for each buffer the number of readers and writers are known; a token is consumed when all readers have read the token, this includes the inactive tasks.

```
int x = 1;
while(1) {
        switch(x) {
                case 1: { f(out x{1}); }
                case 2: { g(out x{1}); }
        }
}
```

Code Listing 6.1: OIL code to illustrate 'hidden' dependencies

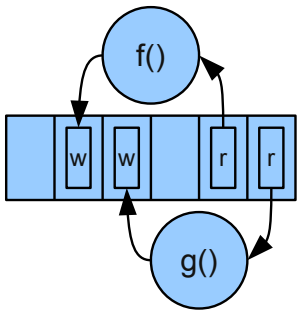


Figure 6.2: Buffer for x for code example from code listing 6.1

Lets visualize the problem: in code listing 6.1 we have a switch statement where 2 functions can be executed based on the value of x. If *f()* would always return the value 1 for $x\{1\}$, the programmer might have the indication that the execution of *f()* and *g()* are unrelated as function *g()* will never be executed. However, when we look at the resulting graph in Figure 6.2, we can see that both functions access the same buffer[2] for variable x. In figure Figure 6.2 we see a round robin FIFO buffer where the read and write windows for both tasks are visible[3]. Since both functions have a write window within this buffer, if one of the functions would not move its write window (for example because it was not executed), at some point the read windows of both functions will not

[2]The switch statement in OIL requires all tasks to examine the value of x to determine whether it should execute its function.

[3]Read windows can be moved 'forward' but only when data is available. In other words: as long as the next buffer position is not part of a write window.

be able to progress. As a result, the progression of both functions is directly related whereas the original program does not clearly show this dependency.

The problem exists because variables are shared between conditional blocks and while only one conditional block is active at one time in a iteration, tasks from other blocks still have to update their read and write windows in order to allow other functions to progress as well. This means that even though the OIL program has no dependencies except the ones that are clearly a result from connecting inputs and outputs from functions together, the resulting executable might exhibit unexpected restrictions from these 'hidden' dependencies.

6.2.4 Data Level Parallelism

OIL has no direct support for DLP. As we explained in the previous chapter, it can be created by means of a work-around. However, this work-around requires the modification of the Omphale results to force the creation of communication channels with larger capacities than the original sizes.

The reason that larger communication buffers are required is quite simple: when for example 16 tasks are to be executed in parallel, the input and output buffers for these tasks should have space enough to serve all tasks at the same time. When communication channels are large enough, the execution of the filters will become pipelined.

See for an example Figure 6.3. In this image, the top execution schedule is shown for a single filter task where 2 symbols are processed. In the bottom schedule, the addition of three extra filtering tasks makes it possible for both the symbol fetch and the processing tasks to be executed without having to wait for the longer filtering task.

6.2.5 Communication Channel Capacity

As mentioned before, in some cases increasing communication buffer sizes yields performance improvements. This is especially true for the DAB decoder where functions have varying execution times, depending on whether symbols can be skipped.

By increasing buffer sizes it is possible to process the skipped symbols[4] while processing on normal symbols is performed. In this fashion, the complete execution schedule per frame becomes more compact and thus faster.

Omphale uses a program called Hebe [33] to analyse the program model and provide guarantees about the execution of the data flow model. While it is possible to improve the mapping by providing latency and throughput constraints, Omphale currently does not use this functionality.

[4]Rather than actual processing of input data, input data for such symbols is discarded and each resulting token in the graph is marked as such. As a result, all signal processing can be avoided on such tokens.

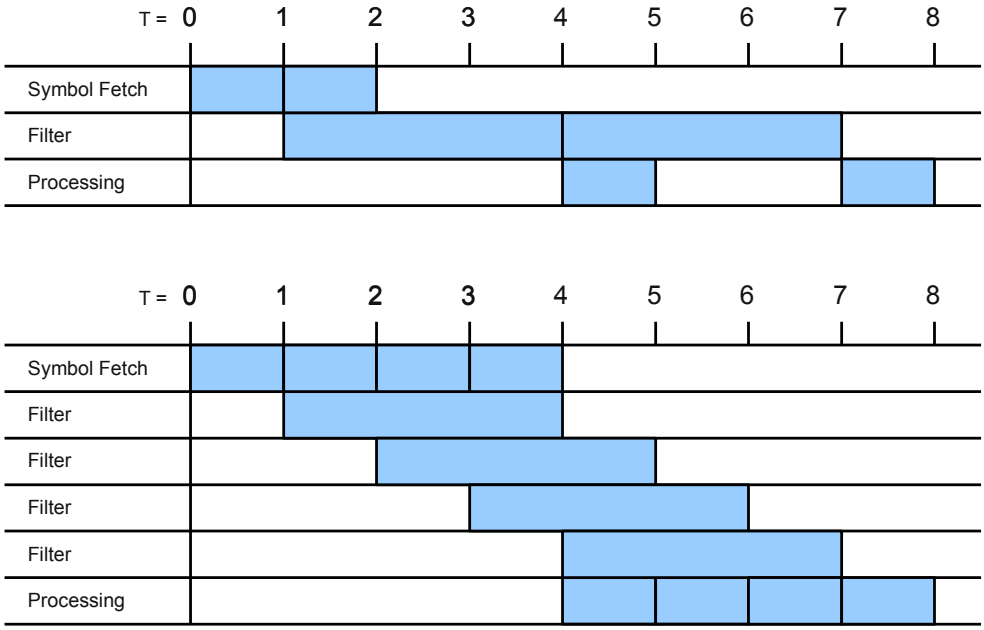


Figure 6.3: In the top schedule, the normal decoder structure is shown. In the lower schedule, 3 additional cores are used for filtering to increase performance.

In fact, since OIL currently has no support for adding execution times of tasks, most of the real-time aspects are not correctly processed by Hebe and as such the resulting tool kit is limited in the guarantees it provides. Without proper execution times, the only guarantee that can be made is the absence of deadlocks. A form of annotations could be added to the OIL syntax to support adding execution times and Omphale should be extended to use this information to guarantee throughput or latency bounds.

An alternative (and probably preferred) method could be automatic Worse Case Execution Time (WCET) measurements: by providing a representative set of test data, Omphale can produce a CSDF model using fixed WCET values for all tasks. The test set is used on the resulting program to measure actual WCET values which can be fed back into the data flow model. The new CSDF model has WCET information and as such can be modified to meet latency and/or throughput bounds.

6.3 Summary

When modifying existing algorithms, the dimensions of the modifications are quite relevant to the usability of the tool kit. When re-factoring to exploit parallelism is not possible, a rewrite of the functionality might suffice. At this point, the question arises whether re-factoring of an existing algorithm was the most efficient course of action.

The mapping of the decoder provided insight into the kind of problems that can be expected when converting a sequential algorithm into a parallel equivalent. While some problems with dependencies can be removed by re-factoring the algorithm, others are fundamental and originate from the specification; as such they are impossible to remove. The only way to introduce parallelism in such a case is the use of methods which relax the constraints. In this project we spent a lot of time in the removal of data structure sharing in order to create parallelism.

The limitations of the function interface which Omphale imposes introduce new problems in existing algorithms. Depending on the algorithm, minor modifications suffice while in other cases a full rewrite would be required.

In the next chapter we will present our conclusions for the case study and present some suggestions for future work.

CHAPTER 7

Conclusion

In this report we evaluated the mapping of an existing DAB decoder algorithm using OIL for use on a homogeneous multi-core System on Chip (SoC). We evaluated the mapping process itself as well as the parallel language and presented the mixed results.

After determining the requirements needed to run a DAB radio decoder, the available I/O on the NEST platform appeared unsuitable. The ML-605 testing board supported ethernet and after designing a minimalistic network stack, a real-time I/O port was created. The use of a dedicated CPU to handle all ethernet network traffic guarantees that none of the other CPUs in the platform are influenced − or can influence − the bandwidth and latency of traffic over the ethernet. The only factor affecting I/O behaviour is sharing effects of the shared memory on the platform; something that could be minimized or even prevented based on the NoC used.

During the mapping process it became clear that re-using existing code has obvious advantages but also causes problems which were not an issue in the original, single threaded algorithm. Data dependencies can sometimes be removed by moving pieces of code around to other partitions. Most of these changes can be done without affecting functional behaviour. However, some of the data dependencies are the result of design time decisions rather than fundamental dependencies and should be avoided as they inhibit potential parallelism.

The best results should be achieved when use on parallel hardware was a design time consideration: keeping data dependencies a local as possible and creating a processing pipeline (sequential) rather than a nested processing structure will greatly ease the conversion and splitting of the algorithm.

While the use of standard 'blocks' of compiled code in parallel in OIL provides numerous advantages over alternative (fine grained) parallel languages like easier re-factoring or simpler design, it also proved to be a barrier to

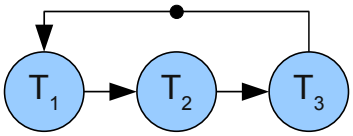


Figure 7.1: HDF example with 3 tasks and no parallelism

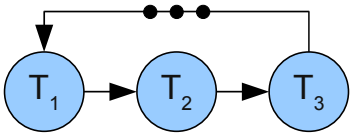


Figure 7.2: HDF example with 3 tasks and parallelism

effectively split large algorithms (like 'large' filters with internal cyclic dependencies) in segments small enough to fully utilise the hardware.

This is reflected in the fact that OIL works best on algorithms which can be split in independent segments or partitions without any dependencies except inputs and outputs. Also, these inputs and outputs should preferably be without cyclic dependencies. If for whatever reason, cyclic loops in the data path can not be avoided, at least pipelining should be possible to be able to run parts in parallel. For example take the example in Figure 7.1 with 3 independent tasks. Because the output of T_3 is required for the execution of T_1 and each consecutive task needs the output of the previous task, effectively only one task will be executing at every point in time. By pipelining the result of T_3 before passing it back to T_1, and assuming all tasks have the same execution time, all tasks can be executed in parallel. See Figure 7.2. This does mean however that it should be possible for T_1 to process data from T_3 with a delay.

Because the OIL tool kit is generating a data flow model from the OIL code which is then used for buffer size calculations, it is important that tokens have a predefined, fixed size. Since tokens are communicated through those buffers, any token which contains pointers to anything but static global data will be unsuitable to be used with OIL, see subsection 6.2.2.

The advantage of using complete functions as parallel units is that it enforces a strict I/O pattern: all communication is done by processing arguments and returning results. This I/O model has 2 major drawbacks.

The first problem is the fact that processing in a function needs to be aborted in order to return a result, even if not all input data is processed, as the number of results is defined in the OIL program. Since OIL is mainly suitable for streaming applications, it is not always feasible to expect input tokens to contain exactly enough data for one result. Since buffering of the input data is a bad idea (if every input token has data for multiple output tokens, infinite memory is needed to buffer the surplus results), the entire

control flow needs to be aborted and somehow resumed on the next invocation. Also, to prevent data loss in the stream of input tokens, (partly) unused input tokens should be managed, requiring additional logic.

To visualize this, refer to Figure 6.1 for an example. The program is used by invoking a function with an input and a single output. If from the input data x at the designated point in the control flow (multiple scopes/functions/loops into the control flow) the output y is created and all off x is consumed, there is no problem. However if this is not the case, the function needs to return y and terminate. The remainder of x should be handled and in the next invocation, processing will have to resume at the exact same point in the control flow - something which is impossible without rewriting the control flow (which might result in a full rewrite of the program, voiding the point of the conversion in the first place).

The next issue is an extension of the previous problem: if the number of output tokens is not fixed (so $N = 0$ or $N > 1$) a form of dynamic data flow is required, something the current OIL tool kit (and syntax, for that matter) can not handle. This means it is not possible to return multiple results, as stated before, and it is not possible to have no output at all at the termination of a function. The latter can be solved by implementing a work-around in the underlying algorithms (e.g. a valid bit on each token).

Note that providing access to the internal communication channels used by Omphale to solve the problems with variable numbers of tokens would void the point of using it in the first place: if communication is unpredictable, the tool kit can no longer determine buffers sizes or provide any guarantees like freedom of deadlock. While solving the communication interface problems, new problems would be introduced. Note that the communication interface imposes generic restrictions which are unrelated to the underlying algorithm.

Although most of the OIL language describes a form of data flow model, the conditional support in the language allows for more than simple data flow algorithms. The resulting hybrid solution, which in essence is a mix between data flow and state machines, allows more complex algorithms than conventional data flow solutions.

The used DAB algorithm has a dependency in the data structure which prevents parallelism regardless of the used mapping language. The only 'solution' to get parallel processing involved changes which altered functional behaviour. Note that a lot of these dependencies originate from the original DAB specification and as such are fundamental: whether the algorithm would be implemented in an imperative or an applicative language would not change this.

7.1 Future Work

As described, the current implementation of the decoder in OIL has parallelism in specific sections. This could be improved by redesigning for example the frame decoder to be executed in parallel. It is likely that dependencies within such parts prevent completely independent execution of components but finer grained partitioning should be possible, allowing the OIL tool kit to distribute the work load better.

Besides improving the mapping of the DAB decoder, OIL itself can be improved as well. As described in subsection 6.2.1, the interface used to describe tasks enforces a fixed bound on the number of input and output tokens. This fixed number of tokens is used to provide guarantees about throughput, latency and absence of deadlock. In some situations, re-factoring an existing algorithm can prove to be impossible because the control flow is too complex to consume and produce the correct amount of tokens. The task interface definition could be expanded to incorporate a more flexible scheme where the number of input and output tokens is flexible (within set bounds) while retaining the fixed token type specification.

Such an extension will provide upper and lower bounds on in- and output tokens, allowing the same models and calculations as before while improving flexibility. For the matter of connecting in- and outputs, the most straightforward method would be to provide the same output (multiple tokens) to the designated inputs. This will require special handling of such tokens in the algorithm itself.

APPENDIX A

HSI Protocol

Communication with the device itself is run on top of UDP, as explained in section 3.5. A protocol has been designed to cater to the basic needs for the functioning of the network bridge: it allows the storing and loading of data to and from the MPSoC by means of the NoC. Since the protocol is controlled by a program on a computer, the following explanation is from the point of view of the computer.

A.1 Commands

In this section, we will explain the commands used in the HSI protocol. For each command we will provide a brief explanation as well as a communication diagram to clearly show in which order packets are exchanged between the platform and the remote host.

A.1.1 Diagnostic

The 'noop' command (No-Operation) sends a packet to the High Speed Interface (HSI) which is simply replied to with a status message. In this way, it is possible to effectively 'ping' the HSI bridge to test connectivity and make sure the bridge is 'alive', see Figure A.1.

A.1.2 Data loading

The 'load' command is used to load up to ∼1400 bytes in one packet. The load request gets a reply packet with the requested data as soon as the data has been fetched from the NoC.

Note that since all resources are memory mapped, the load is performed on a specific address. Due to the nature of the used hardware, even reading

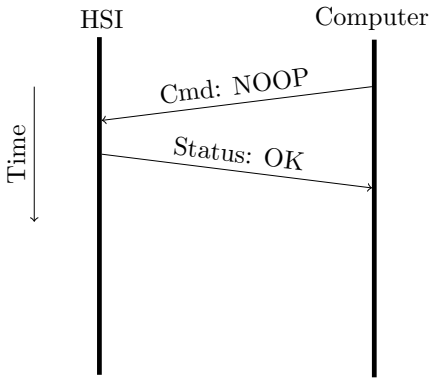


Figure A.1: NoOp command

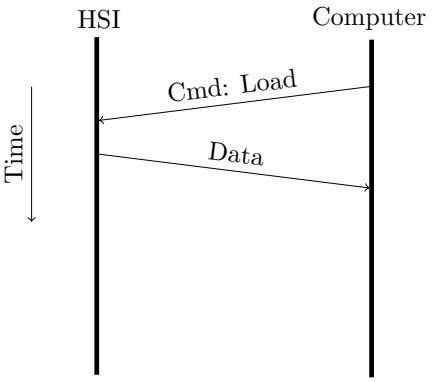


Figure A.2: Load command

from unused address ranges will yield a result (all zeroes for example). See Figure A.2 for an example.

A.1.3 Data storing - Single packet

Since the bridge was designed with a specific application in mind, the performance constraints on data storing are much higher than on data loading.

The first store mode is similar to the 'load' command: a packet is sent containing the target memory address and payload. As soon as the bridge is done storing the data to the specified address, a confirmation packet is sent back to the host to confirm the store, see Figure A.3.

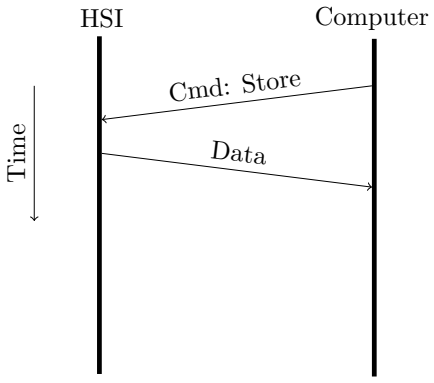


Figure A.3: Store command

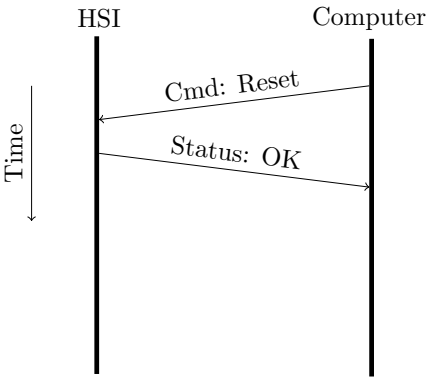


Figure A.4: Reset command

A.1.4 Data storing - Burst mode

The second mode takes advantage of the fact that the network connection is pretty reliable and assumes most (if not all) packets are received properly. A

large amount of data which requires multiple packets to transmit, is sent in a burst of packets: the host sends all data up to a certain amount without needing a reply. When all data is sent (or a certain threshold is reached), the host requests a list of received packages from the bridge.

Because every packet in a single burst had a unique ID, the so called 'manifest' contains exactly which packets were received and which packets were lost. If packets were lost, those specific packets will be sent again in a burst and the manifest is requested again. This process repeats until all data is received or a certain threshold is reached, for example when 500% of the data size has been sent without completing the transfer.

The only thing left is to be able to 'wipe' the manifest between bursts: this is done via a 'reset' command. It works similar to the 'noop' command but before sending a reply, the manifest is cleared. see Figure A.4 for an example of a single 'reset' command.

See for an example with all aspects Figure A.5.

This method of sending data removes the overhead of waiting on acknowledgements for each packet and frees up CPU time on the bridge since it only has to construct replies every 1,400 packets or so. The benchmark results confirm the performance boost and show the achieving of the maximum throughput for the hardware.

Warning: the burst transfer mode is the only command which has 'state' as it keeps track of all received packets. This means that if the 'burst' command would be started a second time while a previous transfer was still in progress, most likely both transfers would fail and the actual transferred data between host and platform is unknown. It is up to the remote host (or hosts) to implement the mutual exclusion that would prevent this.

During a burst transfer, the following 2 commands are not allowed: 'reset' and a new burst store. All other commands do not influence the burst transfer, the only effect being the sharing of bandwidth. This means that all other commands can be used during a burst transfer.

A.1.5 Summary

The HSI protocol was designed with simplicity in mind: simple C structures are used to 'pack' all protocol information and allow for easy parsing and generating packets. The two store modes allow users to switch between a fast store for small data fragments or an accelerated mode for large stores.

The current 'load' command is suitable for small amounts of data and has a limited bandwidth but if more bandwidth is needed for retrieving data, a burst approach could be implemented for the 'load' command as well.

To allow for easy access to the system, all of these functions are implemented in a library which has wrappers for both data storing and loading, allowing the whole inner workings to be hidden to users who only need access to the board and are not interested in tweaking communication.

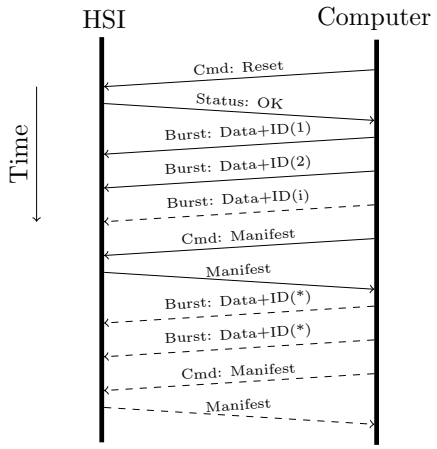


Figure A.5: Burst store

All commands except the 'burst' store are thread safe: any interleaving of packets will result in predictable behaviour since all commands and their reply fit in a single packet (and packets are transmitted atomically).

The exception here comes from the fact that the 'burst' commands keeps state during the transfer. The interleaving of 'noop', single loads and single stores during a burst transfer is allowed as it will not interfere with the burst (in a worst case scenario burst transfer speeds are reduced).

List of Figures

Acronyms

ARP Address Resolution Protocol. 29, 30

ASIC Application-Specific Integrated Circuit. 1

COFDM Coded Orthogonal Frequency-Division Multiplexing. 34

CPU Central Processing Unit. 2, 18, 21–23, 26, 27

CSDF Cyclo-Static Data-Flow. i, 43, 61

D-QPSK Differential Quadrature Phased Shift Keying. 34, 35, 47

DAB Digital Audio Broadcasting. i, ii, 4, 5, 33–35, 63, 65

DLP Data Level Parallelism. ii, 8, 46, 49, 60

DTL Device Transaction Level. 19

FIC Fast Information Channel. 36–38, 40, 46, 47

FIR Finite Impulse Response. 48

FP Functional Programming. 3, 12

FPGA Field Programmable Gate Array. 18, 21, 24

FPU Floating Point Unit. 22

HSI High Speed Interface. 67

ICMP Internet Control Message Protocol. 30

IF Intermediate Frequency. 45

IIR Infinite Impulse Response. 48

ILP Instruction Level Parallelism. 7, 8

IPv4 Internet Protocol version 4. 29, 30

LP Linear Programming. 2

MAC Media Access Control. 29

MMIO Memory Mapped I/O. 18, 24

MPSoC Multi Processor System on Chip. i, 17, 18, 21, 24, 25

MSC Main Service Channel. 37, 38, 40, 46, 47, 49

NEST Netherlands Streaming. i, 12, 17–19, 23, 25, 31, 63

NLP Nested Loop Program. i, 12, 15

NoC Network on Chip. 18, 19, 21, 22, 24–26, 63, 67

NXP Next eXPerience Semiconductors. 12, 17

OIL Omphale Input Language. i, ii, 4, 5, 12–14, 16, 17, 63–65

PLB Processor Local Bus. 22

RISC Reduced Instruction Set Computing. 22

SaC Single assignment C. 11

SIMD Single Instruction, Multiple Data. 8

SoC System on Chip. 63

TCP Transmission Control Protocol. 30

TFPR Transmission Frame Phase Reference. 36, 38, 40, 44, 46, 47

TLP Task Level Parallelism. ii, 8, 10, 49

UDP User Datagram Protocol. 30, 67

USB Universal Serial Bus. 23, 24

VCD Value Change Dump. 43

VHDL VHSIC Hardware Description Language. 18, 21

VLIW Very Long Instruction Word. 7

WCET Worse Case Execution Time. 60, 61

Bibliography

[1] Agilent Technologies. Agilent vee. `http://www.agilent.com/find/vee`.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Ruud van der Pas Barbara Chapman, Gabriele Jost. *Using OpenMP: Portable shared memory parallel programming.* Massachusetts Institute of Technology, 2008.

[4] BBC. The long, slow birth of dab radio. `http://www.bbc.co.uk/news/10569231`.

[5] OpenMP Architecture Review Board. Openmp application program interface. `http://www.openmp.org/mp-documents/spec30.pdf`.

[6] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4, Jul.-Aug.):23–29, 1999.

[7] European Broadcasting Union. *ETSI EN 300 401 - Radio Broadcasting Systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers Specification 1.3.3*, 05 2001.

[8] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414 – 421, sept.-oct. 2005.

[9] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. *SIGPLAN Not.*, 42:251–264, October 2007.

[10] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[11] Sverre Holm. Audio quality on the air in dab digital radio in norway. In *Audio Engineering Society Conference: 31st International Conference: New Directions in High Resolution Audio*, 6 2007.

[12] IBM. The x10 programming language. `http://x10.codehaus.org/`.

[13] Cilk Arts Inc. Cilk 5.4.6 reference manual. `http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf`.

[14] Intel. Single-chip cloud computer. `http://www.intel.com/info/scc`.

[15] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1991.

[16] Simon Peyton Jones, Paul Hudak, Philip Wadler, and et al. Haskell programming language. `http://www.haskell.org/haskellwiki/Introduction`.

[17] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[18] K.W. Martin. Complex signal processing is not complex. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 51(9):1823 – 1836, 2004.

[19] NVIDIA. Cuda. `http://www.nvidia.com/cuda/`.

[20] NVIDIA. Nvidia cuda programming guide, version 3.0.

[21] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[22] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[23] John Postel. Internet control message protocol. RFC 792, Internet Engineering Task Force, September 1981.

[24] Radio en tv zender in nederland. `http://radio-tv-nederland.nl/dab/dab-index.html`.

[25] RICE. Habanero java (hj) project. `http://habanero.rice.edu/hj`.

[26] SAC-Research Team. Single assignment c – functional array programming for high-performance computing. `http://www.sac-home.org`.

[27] Philips Semiconductors. Device transaction level (dtl) protocol specification, version 2.2, 2002.

[28] Agam Shah. Intel to ship samples of experimental 48-core processor. `http://www.itworld.com/hardware/103833/intel-ship-samples-experimental-48-core-processor`.

[29] S. Sluizer and J. Postel. Mail transfer protocol: ISI TOPS20 MTP-NIMAIL interface. RFC 786, Internet Engineering Task Force, July 1981.

[30] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66 –73, may-june 2010.

[31] IBM Systems and Technology Group. Benefits of dual-core computing on ibm system x servers using intel processors. `ftp://dispsd-40-www3.boulder.ibm.com/eserver/benchmarks/Benefits_of_Dual_Core--Intel.pdf`.

[32] David W. Wall. Limits of instruction-level parallelism. *SIGARCH Comput. Archit. News*, 19(2):176–188, 1991.

[33] M. Wiggers. *Aperiodic Multiprocessor Scheduling*. PhD thesis, University of Twente, 2009.

[34] Maarten H. Wiggers, Marco J.G. Bekooij, and Gerard J.M. Smit. Monotonicity and run-time scheduling. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 177–186, New York, NY, USA, 2009. ACM.

[35] Thomas Lauterbach Wolfgang Hoeg. *Digital Audio Broadcasting: Principles and Applications of DAB, DAB+ and DMB*. John Wiley and Sons, Ltd, 2009.

[36] WorldDMB. Introduction to dab/dab+/dmb. `http://www.worlddab.org/introduction_to_digital_broadcasting`.