Synthesising Heterogeneously Encoded Systems

W. B. Toms, D. A. Edwards[1]
School of Computer Science,
The University of Manchester, Oxford Road,
Manchester, M13 9PL, UK
{wtoms,doug}@cs.man.ac.uk

A Bardsley
Silistix Ltd, Armstrong House, Oxford Road,
Manchester, M1 7ED, UK
ab@silistix.com

Abstract

Self-timed datapaths require their data to be encoded in a delay-insensitive manner. The dual-rail encoding is commonly used, but more complex codes offer the possibility of better energy efficiency or fewer wires-per-bit. However, these advantages are often negated by datapath manipulations within large systems that require code-groups to be split and reformed. These overheads may be reduced by heterogeneously encoding circuits based on the datapath requirements within the circuits. In this paper, such an approach is evaluated within the Balsa asynchronous synthesis system.

*Techniques for synthesising arbitrary m-*of*-n encodings for datapath components are presented. These implementations allow each channel within a handshake circuit to be assigned an individual encoding. An automated encoding mechanism is described which analyses the datapath requirements of Balsa circuits and assigns codes to channels based on the interaction between sections of datapaths. The performance of the heterogeneous approach is evaluated on two microprocessor implementations.*

1. Introduction

Process variation is becoming a key concern of VLSI designers. Geometry and gate threshold parameters may vary by as much as 45% and 15% respectively in 70nm processes [10]. To cope with these variations, timing overheads are added to synchronous designs resulting in designs that are often clocked at half their actual maximum potential operational speed [6].

The use of *self-timed* [14] design paradigms can overcome the problems of process variation. The execution of self-timed operations is initiated by the arrival of their operands. The validity of data is encoded into the operands, avoiding the need for an external timing reference, making self-timed operations tolerant to variations in propagation delay. Data is encoded within *DI codes* [17]: codes in which no code word is contained within any other, allowing each code word to be detected unambiguously.

There are many different types of DI code. The most simple is the dual-rail code. Here, each bit of data is represented by two wires, one for each binary value. A transition on either wire represents the arrival of valid data. A dual-rail code can be generalised to the 1-of-*n* codes, where datapaths are constructed from *groups* of *n* wires, each of which represents *n* symbols – again the arrival of a single transition in a code group represents the arrival of valid data. In general only codes where $n = 2^m$ are used so as to simplify the construction of binary width datapaths.

The dual-rail code is the most commonly used 1-of-*n* code as logical functions are simple to construct. However, a 1-of-4 code has greater *communication efficiency* [17] than a dual-rail code. In the 1-of-4 scheme, two binary-bits are encoded within a single code group. Half the number of transitions are therefore required to transmit data compared to a dual rail code. Communication efficiency is especially important in self-timed paradigms because most systems employ a 4-phase RTZ (Return-To-Zero) signalling protocol. In a RTZ protocol, all wires in a code group return to a "spacer" (invalid) state in between transmissions of valid datawords, therefore doubling the number of transitions and thus the energy required to transmit data.

1-of-*n* codes are part of a family of codes known as *m*-of-*n* codes: each code group contains *n* wires and valid data is signified by the arrival of *m* transitions upon these wires. The number of symbols represented by an *m*-of-*n* code group, its *size* [17], is given by *n* **choose** *m*. Therefore, the size of *m*-of-*n* codes, where m > 1, is greater than that of the equivalent 1-of-*n* code and hence fewer wires are required to carry a datapath. In codes where *m* is relatively small ($m \leq \frac{n}{2}$) the communication efficiency may also be better than that of 1-of-*n* codes. However, employing complex *m*-of-*n* codes is difficult because if codes are not chosen correctly, the cost of implementing logic functions can be high. For example, because each code group of the 1-of-4 code represents two

1. This work was supported by the EPSRC Advanced Processor Technologies Portfolio Partnership Grant GR/S61270/01 and an EPSRC studentship

binary-bits, odd width datapaths are difficult to represent, and usually one bit (either the most or least significant) is implemented with a dual-rail code. This may incur problems when manipulating odd-width datapaths, as shown in figure 1.1. The figure shows the results of combining a 3-bit

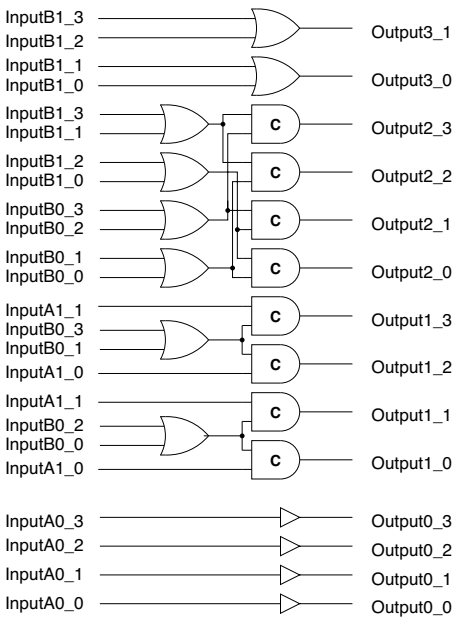

Figure 1.1: Combining 1-of-4 datapaths

(InputA) and a 4-bit (InputB) datapath to form a 7-bit datapath using a 1-of-4 code. Input A consists of two code-groups: one 1-of-4 (InputA0) and one dual-rail (InputA1). InputB consists of two 1-of-4 code-groups (InputB0 and InputB1). The combined output consists of four code-groups: three 1-of-4 groups (Output0, Output1 and Output2) and a dual-rail code group (Output3).

In conventional binary and dual-rail encodings, InputA and InputB may be combined by simply renaming the wires of the two datapaths. In the example above the first code group of InputA requires no conversion. However, because the most significant bit of the InputA is encoded in a dual-rail code, it must be incorporated with the least signficant bit of InputB into a single 1-of-4 code group in the output. The remaining code-groups of inputB must then be recoded into the code-groups of the output which requires circuitry to extract the values of individual bits from each code group. Clearly this operation could be avoided if the output was encoded in a manner that reflected the width of the inputs.

The problem of recoding datapaths applies to all datapath operations, such as bit-extraction, combining and splitting datapaths, when using 1-of-4 encodings. A measure of the overhead of these operations can be gained from a 1-of-4 implementation of the SPA, an ARM Compatible CPU developed at the University of Manchester [13]. SPA contained 623 odd-width datapath manipulations, resulting in a total extra cost of 85186 transistors, around 8% of the entire CPU.

The solution to these difficulties is to allow the encodings of function outputs to be determined by the encodings of their inputs. Thus reducing any unnecessary overhead translating between the two. As in the example shown in figure 1.1, this often requires datapaths to be encoded with multiple different codes. Hence, a system is required that can synthesis *heterogeneously-encoded* circuits and provide a method of specifying new codes.

As the outputs of one function often form the inputs to other functions, conflicts can occur between the requirements of each operation. Therefore the encoding of each datapath must take into account, not only its own operation, but also the operation of the datapaths adjacent to it. For this reason, a *gradated encoding scheme* has been developed, where the encoding of each datapath is determined not only by its own inputs and outputs, but also by the operation of datapaths in its vicinity. The effect of other datapaths on the encoding deminishes with distance from the original datapath, allowing a gradual change in the encoding across the circuit, thus reducing the overheads of changing encodings.

The structure of this paper is as follows: Section 2 defines a method of specifying codes and encodings for the construction of datapaths and describes possible protocols that may to be employed to allow datapaths with different encodings to interact. Section 3 introduces the Balsa synthesis system and handshake components. Section 4 describes the adaptation of the Balsa system, in particular new implementations of handshake components, to synthesise heterogeneously-encoded circuits. Gradated code assignment is outlined in section 5. The results of implementing the gradated encoding mechanism on two microprocessor implementations are presented in section 6.

2. Heterogeneously Encoded Systems

In order to create arbitrary-encoded DI circuits, a specification must be designed to allow for the description of DI codes and datapath structures; decisions must also be made on how different datapaths interact with each other within components. The term *channel* is used to describe atomic elements of datapaths which may be assigned different encodings.

2.1 Code Definitions

There are three important properties of a code:

- *Size* - The number of symbols contained in each code-group.
- *Width* - The number of wires in the code-group

- *Values* - The values of the code-group and their order.

Given this information, the structure of datapaths for any width can be determined, provided a method for constructing datapaths from code groups is defined.

It is inefficient to use large codes on their own to construct arbitrary-width datapaths. Where a datapath's size is not a multiple of the code's size, the number of symbols must be *truncated* to fit the number of symbols of the datapath. A more efficient structure for the channel may be constructed by employing smaller codes within the channel to achieve a better fit. Each code definition can contain a set of *diminished* codes which are used to implement varying sizes of datapaths. For example, the 1-of-4 code contains only one diminished code (dual-rail) to represent single data bits in odd-width datapaths. Larger codes comprise code sets that contain several smaller codes decreasing in size, that are used when appropriate. Once the set of code-groups covering a datapath has been determined, the number of wires in the datapath and the datapath symbols may be determined easily.

The set of codes that can be described in this manner is limited by practical considerations such space and computation time. Large codes are difficult to define because of the need to describe all of the code values, which requires enumerating every code word.

2.2 Channel Encodings

Once codes have been defined, describing the encodings of datapaths is relatively simple. A heterogeneous encoding for a datapath consists of a sequence of (*code . width*) pairs. e.g.

```
(("2-of-7" 4) ("dual-rail" 1) ("2-of-5" (symbols 8)))
```

The *code name* is the reference to a code description and the *width* is the number of bits the code represents. The *symbols* keyword allows the width of the code to be determined by the actual number of symbols representd, allowing non-binary width codes to be represented. All further information required to implement channels can be determined from the code descriptions.

2.3 Channel Interaction

In heterogeneously encoded systems, each channel may have a different encoding and so interaction between different channels is complex. Interaction is defined by a function which may be a logical or arithmetic function, or a connection function which implements datapath manipulations such as splitting, combining etc. The implementation of this function is determined by the model of interaction employed between the two channels.

For two channels to interact, the channels must be decomposed into atomic groups, called *segments*, on which operations (including connection) can occur. Values within the datapath are determined by the products of the individual segments; the position of segment in the datapath determines the significance of its values in the value of the datapath. In binary datapaths, each segment consists of a single binary bit. In encoded datapaths, each segment consists of a single code group. The *boundaries* of a segment are defined by the symbols of the datapath represented *before* the segment (lower boundary - s_l), and the symbols represented by the datapath up to the end of the segment (upper boundary - s_h). For example, the 16th bit of a binary datapath forms a segment with $s_l = 2^{15} = 32768$ and $s_h = 2^{16} = 65536$.

Where channels with separate encodings interact, segments have to be constructed by combining the code groups of each channel. Segment boundaries may coincide with boundaries of code groups or occur on factors of the code groups size (given by s_h / s_l).

Figure 2.1 shows the segments of the combine component of figure 1.1. The boundaries of the input code groups and the output code-groups do not coincide within the datapath and so a decision must be made where to place segment boundaries to implement the function.

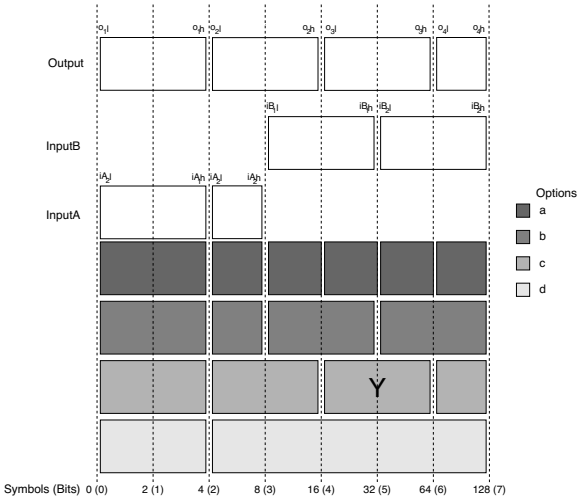

Figure 2.1: Code-Group Boundaries of the Combine Component of figure 1.1

There are four possible methods of combining code groups in order to decompose a datapath into atomic segments:

(a) Groups are combined at the highest common divisor of all group sizes.

(b) Groups are combined on group boundaries of inputs and integer divisors of any output sizes.

(c) Groups are combined on group boundaries of outputs and integer divisors of any inputs.

(d) Groups are combined only on code-group boundaries.

Figure 2.1 shows the various segment options for the combine example. Each of the segments is depicted as filled rectangles and each option has a different shading.

Where segment boundaries do not coincide with code-group boundaries, mapping functions must be used to map between the values of the input code-groups to the segment values and vice versa. For each segment there is a mapping function: $map(x_1, \ldots, x_n) \rightarrow s_i$ where x_i corresponds to a value of an input code group and s_i is a value of the segment. Associated with each input code group are two sets of boundaries: the boundaries of the segment within the code group c_l and c_h and the boundaries of the code-group within the segment s_l and s_h. c_s is the size of the segment within each code group. Mapping functions take the form:

$$map(x_1, \ldots, x_n) = \left(\left(\frac{x_1}{c_{l_1}}\right) \bmod c_{s_1}\right) s_{l_1} + \ldots + \left(\left(\frac{x_n}{c_{l_n}}\right) \bmod c_{s_n}\right) s_{l_n}$$

These values can be illustrated using the example segment *Y* marked in figure 2.1. The two least significant symbols of segment *Y* are determined by code-group InputB1, and so the boundaries of this code-group within the segment are $s_l = 1$ and $s_h = 2$. Segment *Y* intersects the first code-group and so the boundaries of the segment within InputB1 are $c_l = 2$ and $c_h = 4$. The second code-group InputB2 determines the two most significant symbols of segment *Y* and so $s_l = 2$ and $s_h = 4$. The boundaries of segment *Y* within InputB2 are $c_l = 1$ and $c_h = 2$.Therefore the mapping function for segment *Y* marked in figure 2.1 is:

$$map(x_1, x_2) = \left(\left(\left(\frac{x_1}{2}\right) \bmod 2\right) \times 1\right) + \left(\left(\left(\frac{x_2}{1}\right) \bmod 2\right) \times 2\right)$$

The code-groups of output channels dominate the construction of segments. Input code-groups may be used in several different functions, whereas the values of output groups must be calculated in a single logic stage.

The size of large logic functions can sometimes be reduced, particularly in functions with two or more inputs, by performing the input and output mappings in two separate stages.This requires assigning an intermediate DI-code to the values of the segment and usually involves "converting" complex codes into simpler codes in order to perform logic. However, employing two-stage mapping may detract from the advantages of employing complex encodings. As all operations are performed upon small groups and then recombined, extra logic stages and transitions are required. Options (*a*) and (*b*) implement such a mapping because the segments produced under these schemes are not delimited by the output code-group boundaries and so several segments may need to be combined together to form a each output code-group. Option (*d*) is the simplest of the channel interaction models as it does not require input mapping. However, as in the example above, segments can be very large if the code groups of channels do not coincide at regular intervals.

3. A Self-Timed Synthesis Test-bed

Balsa [8] presents the ideal system to explore heterogeneously encoded systems since it synthesises behavioural descriptions into an intermediate form which abstracts away implementation details from the behavioural description allowing several different implementations to be created from a single description. The structure present in the intermediate form allows functional components within the system to be identified and tools devised to implement the circuit components in an arbitrary coding scheme.

3.1 Handshake Circuits and Balsa Synthesis

Balsa uses syntax-directed translation to synthesise descriptions into *handshake circuits*, which consist of interconnected *handshake components* [4]. Handshake components are connected via *channels*, which communicate via handshaking. Most components have a set of parameters, such as width, number of ports, operation or decoder/encoder specification, which allows a wide range of circuits to be created. An example handshake circuit is shown in figure 3.1.

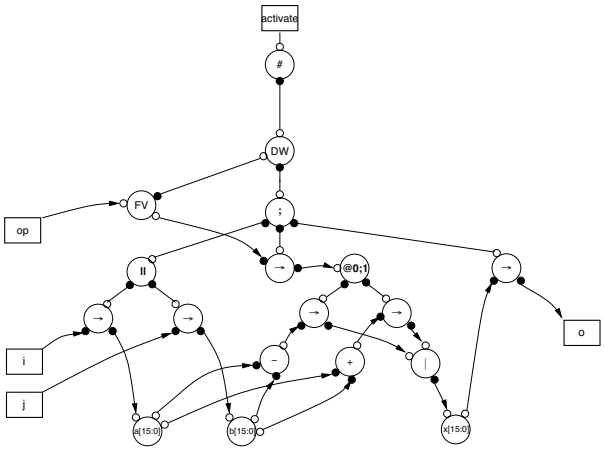

Figure 3.1: ALU Handshake Circuit

No information about the internal structure of channels or handshake components is given in a handshake circuit graph, and so new circuit styles may be synthesised by providing new implementations of each of the handshake components. Currently Balsa can synthesised bundled-data, dual-rail and 1-of-4 encoded *QDI* implementations. As Balsa handshake components can be parameterised, instantiating arbitrary component implementations is non-trivial. The Balsa synthesis system contains a complete language to aid the creation of handshake component descriptions, as well as containing several file and netlist formats to allow implementation technologies and cell-libraries to be targeted easily [2].

As Balsa does not contain its own combinational logic synthesis engine, combinational logic functions in Balsa are constructed from a set of pre-defined *primitives*: gate-level modules consisting of adders, comparators, etc. Information about the structure of channels is hard coded into handshake component descriptions and primitives must be defined for each logical or arithmetic function. Complex encodings require the generation of many different primitive cells, as cells must be created to cope with every possible datapath structure. The 1-of-4 component implementations required 66 different primitives to cope with all possibilities of odd and even datapath widths in logical operations.

To allow Balsa to synthesise heterogeneously-encoded circuits, new generic implementations of handshake components and channels must be created. Each channel may be assigned a unique structure and handshake component implementations must be created that allow each port to contain a different encoding. As the encoding of components is initially undetermined, logic functions can not be implemented using pre-defined primitives, and combinational logic synthesis must be used to implement all functions.

4. Implementing Handshake Components

4.1 Design Methodology

There are several different self-timed methodologies which vary in the assumptions made about the delays within circuit components. Robust methodologies that make few assumptions about the delays within the circuit components are more tolerant to variations in the process and the environment and therefore require less timing verification post-layout. In order to study the feasibility of heterogeneously encoded systems, the Quasi-Delay-Insensitive methodology [9] was adopted, allowing highly complex systems to be constructed and laid-out without the need for complex timing constraint checking to ensure correct behaviour.

It should noted that much of the complexity in constructing generic handshake components concerns the handling and interaction of the encodings and so other self-timed methodologies may be employed by replacing the combinational logic synthesis engine, and performing the appropriate constraint checking and code conversion at the input and out ports of the components.

There exist very few QDI combinational logic synthesis tools, which are capable of synthesising arbitrary logic functions. Most of these require auxiliary tools, such as cell compilers or conventional logic synthesis systems, in order to implement functions. For this reason a new synthesis method was developed to allow arbitrary QDI functions to be created using standard cells. The method was implemented in a tool called Oolong and was used to implement all of the combinational logic within the new system. A description of the Oolong synthesis flow is given in [15].

Oolong employs algebraic extraction to implement multi-level implementations of *strongly-indicating* QDI circuits. The restrictions of the strongly-indicating model mean that conventional minimisation techniques, such as those exploiting 'don't care' states within functions, are difficult to employ without violating the model. However, large circuits may be synthesised without the need to check whether the QDI requirements are upheld throughout the synthesis procedure, which limits the applicability of other synthesis engines to datapath circuits.

4.2 Function Generation

All the logical functions implemented by the Balsa system operate on the binary values of encoded channels, rather than the signalling within these channels, and so operations are dependent only on the *lexicographical value* [11], rather than the binary value, of the input code word. The lexicographical value of a code-word is the position of the word in an ordered list of code words. For each segment in a datapath, a function is synthesised using the Oolong synthesis tool. Functions are generated by enumerating all the possible values of the set of input code-groups. Mapping functions, described in section 2.3, are used to determine the lexicographical value of function outputs for each input code-word. This value may then be used to extract the correct code-word from an ordered list of output values. This truth table is input to Oolong which returns a list of synthesised functions written in Balsa's implicant format, a set of (*don't-care-mask*, *value*) pairs. Abstract gates are generated from each implicant by mapping the original inputs to the input names provided to the function-generator. Once the abstract gates have been generated, they are evaluated and expanded using Balsa's internal gate expansion system.

4.3 Completion Detection

Completion Detection is used extensively throughout DI-encoded circuits to acknowledge the receipt of data. For 1-of-*n* codes, completion detection is simply an OR of the wires of the code group but for more complex codes, it can be substantially more expensive. The Sorting Network (SN) based approach defined by Piestrak [12], provides an efficient method to implement completion detection for arbitrary *m*-of-*n* codes and was adopted to implement completion detection throughout the system.

Sorting Networks are switching networks which "order", by logic level, a set of inputs to a set of outputs. The SNs used by Piestrak are based on Batcher's Odd-Even Merging Networks [3]. Odd-Even Merge Networks take two ordered lists and merge them into a single ordered set by splitting them into odd and even merging networks for the odd and even indices of the two lists. Their outputs are then combined with a row of comparison elements. The inputs are split into two

sets and ordered with smaller odd-even merging networks, before being combined in a large merging network.

Piestrak's method for completion detection relies on the fact that a "1" on the i^{th} input of a SN signifies that i "1"s have been received by the network and hence can be used in the detection of m-of-n codes. In Piestrak's completion detection, inputs are split into two odd-even merge networks, the outputs of which are then combined to cover all possibilities of receiving m active inputs in the two sorting networks.

In order for the sorting networks to maintain the QDI assumptions, the outputs of the networks must be pruned (as suggested by Piestrak) to prevent unacknowledged transitions occurring within the network. As networks are constructed recursively, pruning is a relatively simple operation achieved by reducing the *weight* of the code with each iteration of recursion. Pruning has the added advantage that networks are reduced in size. 1-of-n networks are reduced to OR-gate trees and so the technique provides efficient completion detection for all unity weight codes. For sorting networks with m active inputs, outputs O_1 to O_m will all transition, although only O_m will be used within the combination network to determine the arrival of data. To acknowledge all of the outputs and the internal transitions that generated them, a set of cascading C-elements must be placed on the outputs of the sorting networks. The C-elements delay the transition of O_m until after the outputs O_1 to O_{m-1} have transitioned, allowing all transitions within the pruned sorting network to be acknowledged. Figure 4.1 shows a QDI sorting-network implementation of a completion detector for a 2-of-7 code. The shaded C-elements and

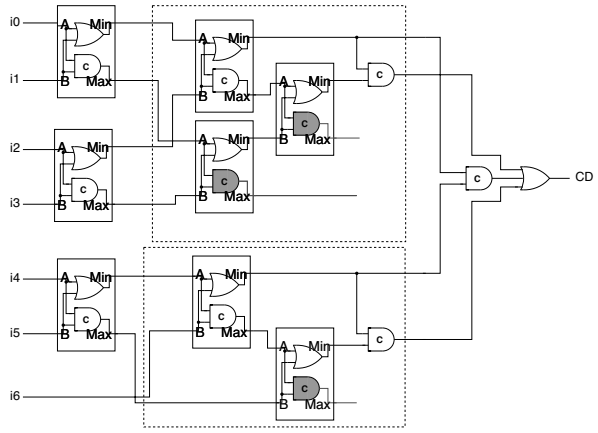

Figure 4.1: 2-of-7 Completion Detection

wires represent components that have been pruned from the original sorting-network.

For DI codes without unity weighting, Piestrak's method cannot be used as it relies on summing the number of transitions rather than a comparison or enumeration based approach. Therefore, to implement non m-of-n codes, enumeration-based methods employing QDI logic synthesis are used.

4.4 Encoders and Decoders

Several Balsa components contain encoder and decoder elements. Decoders activate a single control channel from an array of channels, based on the binary value of a data channel and encoders encode binary values onto a channel based on an event on a control channel. Encoders and decoders are design dependent and, unlike logic functions, must be synthesised rather than generated from primitives. In Balsa, they are synthesised using the Espresso logic minimiser [5].

In order to implement a completely QDI generic backend, Espresso-based synthesis of decoders was replaced with QDI synthesis. However, to implement QDI synthesis, each input transition must be acknowledged correctly. In order for the synthesis tool to achieve this, the set of all possible input transitions must be supplied. This requires constructing truth-tables using an entry for each symbol that is possible in the datapath. For large datapaths, the size of these tables becomes prohibitive, consequently synthesis is difficult and time consuming and often results in large implementations. Datapaths therefore need to be decomposed to implement decoders efficiently.

Synthesis of QDI decoders is a two stage process: the datapath is first decomposed into segments for synthesis, and secondly, the outputs of the segments are combined to create the encoder outputs. Figure 4.2 shows a 24-bit dual-rail decoder implementation.

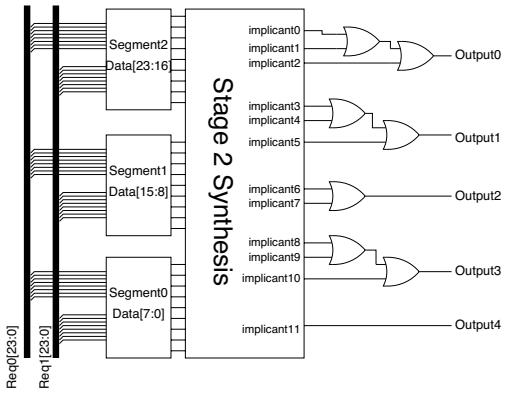

Figure 4.2: Two Stage Decoder Implementation

In order to ensure there are no unacknowledged transitions on the outputs of the segments in the first stage, the segment outputs must reflect not only the results of the encoder output functions over the values of the segment, but also the

relationship between encoder outputs for specific values. If two implicants, *a* and *b*, share a common value within a segment, then, upon receipt of this value, the outputs of the segment for *a* and *b* will both transition. As the decoder outputs are mutually-exclusive, only one of the implicants, *a*, can be true in all segments of the datapath, and produce an output. Therefore, the transition on output *b* of the segment will not be acknowledged by the second stage of the decoder. Instead of describing the state of individual implicants, the outputs of each section describe the relationship between implicants. For each input value in the segment, the implicants activated are recorded. The output of the segment is a one-hot code with an output for each different combination of active implicants. This process is similar to the technique for determining the symbolic cover for a set of states using positional cube notation [7].

If the implicants within a segment are mutually-exclusive, i.e. no implicant covers the same value, then there will be one output for each implicant. If all the implicants cover all values of the segment, only a single output is generated and the segment is part of the don't care set of the decoder and the logic can be replaced by completion detection. In the worst case-scenario, if each input value causes a different set of implicants to be activated, then there will be as many segment outputs as input values and hence, the second stage decoder logic will have as many input terms. However, as the outputs of each segment are one-hot, there are fewer concurrent inputs in the second stage and so synthesis will be simpler.

The second stage logic combines the one-hot outputs of the first stage together and generates signals for the original implicants. All the different possibilities of segment outputs must be enumerated to enable QDI synthesis. The size of the two stage decoders is often a lot larger than implementations generated using Espresso, where don't care values are removed from the expressions entirely. Ironically, the presence of large don't care sections within decoder specifications can actually increase the size of QDI implementations. Each implicant covers a larger range of values, leading to an increase in the number of implicant possibilities within segments and, hence, an increase in the number of segment outputs.

4.5 Storage

Generic storage implementations for *m*-of-*n* codes are expensive. This is because, to fulfil QDI requirements, implementations require completion detection to determine when data has arrived **and** when it has been stored. Figure 4.3 shows three possible generic storage implementations. A series of experiments into the properties of each implementation was conducted [15].

Figure 4.3.a shows a generic SR-latch implementation. Explicit completion detection is required on each of the wires to determine when to reset the latch if a valid data has been received on the other wires. For *m*-of-*n* codes with unity weighting, the completion detection can be implemented using Piestrak's method on the *m*-of-(*n*-1) network. For complex codes this implementation is vastly expensive in terms of area and power as it requires *n* different completion detection networks. Although it is the fastest, as the completion detection of *m*-of-(*n*-1) networks is much simpler and is performed in parallel.

The storage implementation of figure 4.3.b is constructed from a two-stage muller pipeline. This device requires two C-elements per data wire and three separate completion detection networks. The Muller pipeline implementation is much slower than the SR-latch implementation as the previous data must be cleared before new data may be stored. For codes where $n > 3$, the area and power consumption of the Muller pipeline is less than that of the SR-latch implementation owing to the reduction in number of completion detection networks.

Figure 4.3.c shows an attempt to combine the advantageous features of the two previous implementations. The generalised C-elements, shaded on the diagram, determine when the storage C-elements have latched the data by comparing it with the input data, and so reduce the amount of completion detection required. For complex code implementations, the device is smaller and lower power than both the previous two implementation, although not as fast as the SR-latch implementation. However, in order to acknowledge all transitions in the device, the C-elements must be implemented as a single gate which may not be practical in a standard cell library.

One final problem with implementing generic *m*-of-*n* storage is the initialisation of data. Balsa allows reads from variables before writes; in order to initialise the devices properly, a valid code word must be inserted into the device. In 1-of-*n* codes, this requires setting a data wire high, however in *m*-of-*n* codes this is more complex, and a valid code-word must be generated by setting or resetting each of the storage elements as appropriate. The implementations shown in figure 4.3 depict a 1-of-4 code group with a single wire set on initialisation.

5. Code Assignment

The aim of code assignment is to analyse the behaviour of circuits and assign separate codes to different parts, allowing more complex codes to be incorporated while reducing the overheads caused by extracting data values from large code groups. This process is different from state-encoding, which may also be applied to DI-encoded circuits, where code values and orderings are selected to reduce the cost of logic implementations. State encoding techniques for DI-codes have been suggested [16] although have not been implemented. It is anticipated that the system described in this

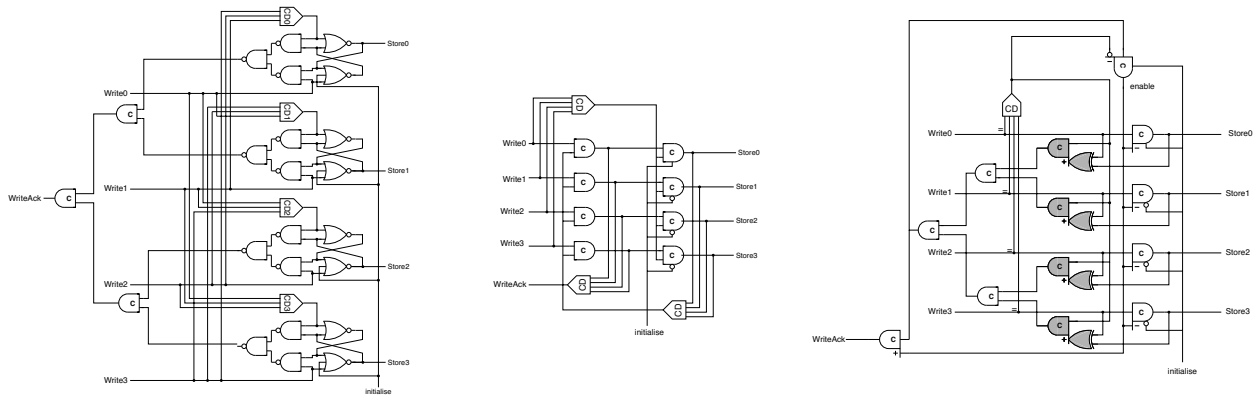

Figure 4.3: Generic Storage Implementations: *a)* SR-latch, *b)* Muller Pipeline, *c)* Asymmetric C-element

paper will provide a platform for further research in this area by creating a framework in which they maybe evaluated and heuristics generated to improve their performance.

The code assignment procedure presented in this paper attempts to encode circuits using *gradated* encoding. Encodings are assigned to sets of channels, called *paths*, within the circuit. Each encoding is determined by attributes of the current path and those of other paths in the vicinity. The effect on the encoding of other paths in the circuit decreases with their increasing distance from the path being encoded. Hence, the encoding changes gradually across the circuit, to accommodate the data requirements of the various parts of each circuit. Clearly, the *gradient* of the encoding - the extent to which the encoding changes across the circuit - greatly effects the circuit characteristics. If the gradient is too steep, then circuitry needs to be employed to convert between encodings. However, reducing the gradient homogenises the encoding of the circuit and often means that small size codes become dominant throughout the design reducing the application of complex codes.

Each path within a design may be surrounded by several different distinct paths. The aim of the encoding procedure is to produce an encoding for the path that is *compatible* with the encodings assigned to all adjacent paths. As each adjacent path may have separate or conflicting requirements, paths are assigned an encoding that will be suitable for all adjacent paths. The algorithm proceeds in four stages:

(i) Small single-width datapaths are determined throughout the design

(ii) These single width datapaths are amalgamated, where possible, into larger more complex datapath structures.

(iii) Conflicts between datapath structures are resolved by exchanging information about the datapath operations that operate on each structure.

(iv) An encoding for each datapath structure is determined based on its own datapath operations, and the requirements of the adjacent structures.

Each of these stages is discussed in detail in the following sections.

5.1 Components and Atomic Paths

The structure of handshake circuits allows codes to be determined and assigned very simply. As there exists only a small number of handshake circuit components, the operation, and hence data requirements, of a datapath can be determined easily. *Atomic paths* are small constant width paths through a datapath, and are the basic blocks of the encoding process. They are constructed from sets of channels and are determined by analysing the behaviour of the components to which each channel belongs. Each component with dataports in a handshake circuit, performs some role upon the paths in which the channels connected to its ports are contained. There are three possible actions a component can perform on paths:

- *Initiate*: initiate a path on an output port.
- *Terminate*: terminate a path on an input port
- *Distribute*: distribute a path between input and output ports.

Atomic paths are generated by traversing channels from circuit inputs and initiators to terminators and outputs. Figure 5.1 shows the atomic paths of the `alu` circuit introduced in figure 3.1. The circuit has 10 atomic paths labelled 0 .. 9. Notice that the inputs to the two binary function components are all on separate atomic paths, but the two outputs are on the same path. This is because the path between the outputs consists of only distributor components.

Each atomic path has associated *path attributes* that are used to determine its encoding requirements. A path's attributes are defined by handshake components on the path that perform logical operations or datapath manipulations on data ports. Path attributes take two forms: *structural* and

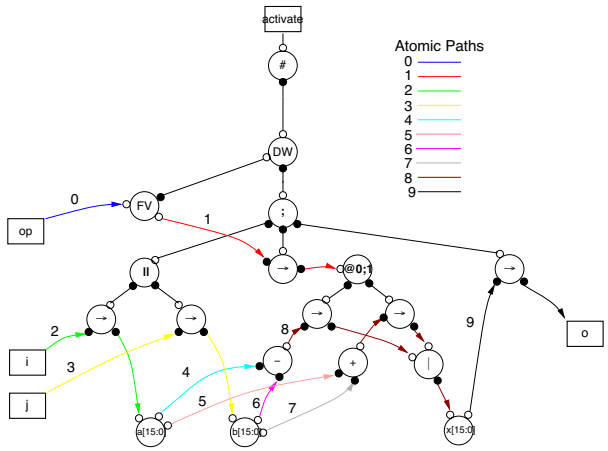

Figure 5.1: Atomic Paths of ALU Handshake Circuit Graph

functional. Structural attributes take the form of a binary mask and are generated in handshake components where the ports of a path are of different width. As all atomic paths have a single width, components which alter the size of datapaths initiate and terminate two different width atomic paths. However, to ensure that the encodings of the two paths are compatible, the larger path assumes a structural path attribute that describes the position of the smaller path within the larger.

Functional path attributes are generated in handshake components that perform logical operations on the ports of the components. No succinct way was available to describe these attributes in a general manner and so they are described by the parameters of the handshake component in which they were initiated. The encoding scheme described in this paper is concerned only with the structural attributes of paths. To employ the functional attributes of paths in a non-trivial way requires complicated state-encoding techniques.

5.2 Propagations

Assigning codes is a heuristic procedure and the complexity of the task can be reduced by combining paths together into a single structure where possible. As the encoding of each path is determined by the encodings of its neighbours, reducing the number of paths reduces the complexity of the encoding process. Therefore, after all atomic paths are generated, each path is examined to see if it may be expanded to incorporate other paths.

Paths of different widths may be amalgamated into multi-width path structures which consist of a dominant (largest width) path and a set of sub-paths. Each sub-path has a mask describing the location of the path within the dominant path. Where two multi-width paths are amalgamated, the sub-path masks of the smaller path are expanded to fit the larger paths and both paths are combined into a single multi-width path.

As handshake circuits may be instantiated within other handshake circuits, information about the role of the instantiatee on the circuit of the instantiator must be generated. This is achieved by creating information about the *external paths* of the instantiated circuit: the paths within the circuit connected to its ports.

The set of external paths of a handshake circuit forms the set of encodings that need to be supplied to the circuit in order for it to be instantiated, this allows different instances of the same handshake-circuit to have different encodings. Only completely internal paths may be encoded at this stage. Before the internal paths can be encoded or the external paths created, the conflicts between paths must be resolved.

5.3 Conflicts

Conflicts between paths are very important as they represent adjacent paths which, for some reason, could not be combined into a single encoding. The use of conflicts may be beneficial by reducing the length and attributes of paths, allowing different paths to be encoded independently. Conflicts between paths may be artificially inserted by the user to increase the number of encodings within a design. In order to reduce the logic required to communicate data between adjacent paths, the encodings of each path must be made *compatible* with the other. When paths are involved in a conflict, each path is encoded separately and attributes must be exchanged between paths to reduce the likelihood of incompatibilities in encodings. However, exchanging too much information increases structural dependencies between paths, increasing the homogenisation of the encoding, so a balance has to be achieved.

The point at which attributes are exchanged during conflict resolution is important. As the two paths are independent, they may collect further attributes which could create incompatibilities between the encodings of each path. In fact, the conflicts themselves form cyclic dependencies on each other, where the resolution of a conflict between two paths is affected by unresolved conflicts from each path. The unresolved conflicts are, in turn, dependent on the outcome of the resolution of the current conflict. In order to correctly resolve a conflict, information needs to be extracted not only about paths involved in other conflicts with the two conflictees but also about the other paths with which those paths are in conflict. To employ gradated encoding, the locality of the conflict information to the original path must be recorded and taken into account during the encoding process. In the process described in this paper, only structural information was exchanged during conflict resolution, although it would be possible to exchange functional or encoding information in the same way.

For each path, a set of structural masks are produced that describe the structure of paths that conflict with the original path at that level. Level 0 paths are those within the multi-

width path structure being examined. Level 1 paths are those that immediately conflict with level 0 paths and level *n* paths are those which conflict with level *n*-1 paths. The structural masks at each level are combined to produce a single structural mask, and these level based masks are then combined into one mask for the original path. Small regions are filtered out of masks from higher levels, to reduce the number of regions within the final mask.

If one of the paths involved in a conflict is external, then complete path information is not available at the time conflicts are resolved, leading to possible incompatibilities between adjacent path encodings. It may be necessary to export the internal path involved in the conflict as an external path so that it may be encoded when the handshake circuit is instantiated.

5.4 Encoding

Once all of the paths and their attributes have been determined, each path may be encoded. Encodings are *incompatible* if they result in a section being created within a handshake component that is too large to be effectively synthesised because of the resulting computational complexity. Once an encoding mask has been determined for a path structure, the path maybe encoded by selecting a suitable code for each region within the mask. In the circuits described in this paper, only code groups with a size equal to a power of two were used in order to reduce the complexity of encoding. For some codes this meant the number of valid symbols had to be reduced. Since no analytical encoding system exists for DI codes, the selection of code words and ordering for all codes used was arbitrary.

6. Circuit Implementations

The effectiveness of heterogeneously encoded systems was evaluated by encoding and synthesising two example circuits using the system described in the paper. Both circuits are complete commercially compatible 32-bit processor cores, designed entirely in the Balsa language. The circuits represent the largest designs attempted in Balsa to date and so give a reasonable indicator of performance of the encoder on complex systems. The circuits are:

- SAMIPS: A MIPS 3000 compatible processor developed by the Distributed Systems Group at the University of Birmingham [18]. The SAMIPS has a 5-stage pipeline and a Harvard Architecture.
- SPA: An ARM V5T compatible core developed at the University of Manchester [13]. The SPA is a 3-stage pipeline with a Harvard Architecture. It was initially developed as part of a project to increase the security of smartcards by homogenising power and timing of all instructions.

Table 7.1 shows area, speed and energy figures for several implementations of each processor. The circuits were automatically placed and routed with First Encounter from Cadence in a standard cell library developed at the University of Manchester using ST's HCMOS8D 0.18μm process. The speed and energy figures are based on running an example program on each processor, simulated with single-node capacitance in Synopsys's Nanosim. The example program for the SAMIPS was a single loop of the Dhrystone test bench. The SPA test program was T1 from the ARM validation suite.

The table shows the benefits of implementing complex codes over a dual-rail code as all of the complex-code implementations are faster and consume less energy than the dual-rail implementations, albeit at the expense of some area overhead. The improvements of the heterogeneously encoded systems over the standard 1-of-4 implementation are less dramatic and highlight the difficulties in implementing such circuits.

All the implementations were encoded using the techniques described in section 5, and were synthesised using the fourth channel interaction model (*d*) described in section 2. The dual-rail/1-of-4 implementations were generated by constraining the selection of possible encodings to dual-rail and 1-of-4 only. The *m*-of-*n* implementations were encoded using dual-rail, 1-of-4, 2-of-5, 2-of-7 codes. The use of each code can be measured by calculating the percentages of each code in the channels of the design, these percentages are shown in table 7.1.

As can be seen from the encoding percentages of the heterogeneously encoded systems, each contains a high proportion of dual-rail code. Despite this, significant improvements in speed are demonstrated for both the dual-rail/1-of-4 and *m*-of-*n* codes albeit at the cost of an increase in area and energy consumption. The use of complex codes may be increased by implementing a better model for channel interaction, which would reduce the size of segments in some cases for incompatible codes. However, increasing the proportion of the complex codes may also increase the gradient and therefore have a negative effect on the performance of the circuit. The encoding mechanism regulates the gradient of the encoding by using conflicts. To allow the user of the system to evaluate the effectiveness of the encoding some method of determining the gradient of the encoding must be employed. Defining the gradient of a system is non-trivial and is the subject of on-going research, it is believed that such information will also help generate heuristics to guide the encoding procedure.

7. Conclusions

This paper presented techniques to automatically encode and synthesise QDI circuits using a range of different DI-codes. There are many advantages to using complex

	SAMIPS								SPA						
Implementation	Dual-Rail	1-of-4	2-of-5	2-of-7	Area (mm^2)	Speed (MIPS)	Energy (μJ)		Dual-Rail	1-of-4	2-of-5	2-of-7	Area (mm^2)	Speed (MIPS)	Energy (μJ)
Dual-Rail	100%	-	-	-	2.1	6.8	2.6		100%	-	-	-	2.6	3.8	4.2
1-of-4	-	100%	-	-	2.7	7.7	1.6		-	100%	-	-	3.5	4.0	3.5
Dual-Rail/1-of-4	45%	55%	-	-	3.0	9.8	1.7		65%	35%	-	-	3.5	5.0	3.7
m-of-*n*	54.0%	11.5%	9.3%	8.6%	3.2	8.6	2.0		54.0 %	11.5 %	12.0 %	22.5 %	4.0	4.5	4.0

Table 7.1: Area, Speed and Energy figures for Heterogeneously encoded Microprocessor Implementations

DI-codes within circuits to increase performance and reduce energy consumption of circuits. However, implementing heterogeneously encoded circuits is complex. The methods presented in this paper merely represent an initial study into synthesis and encoding techniques. In order to implement such systems effectively, more research is necessary into several aspects of the research:

The significance of ordering and code word selection in reducing the size of codes has been highlighted elsewhere [1], in order to make effective use of *m*-of-*n* codes, state-encoding techniques must be applied to DI-encoded circuits such systems have already been conceptualised [16] but are yet to be implemented.

The combinational logic synthesis techniques used in the Oolong tool were developed for this system. The use of a strongly-indicating methodology often results in large circuit implementations. The performance of heterogeneously-encoded circuits could be increased by implementing combinational logic functions in a less restrictive methodology, allowing minimisation techniques to be implemented more successfully.

Both of these techniques have much wider applicability outside of the domain of heterogeneously-encoded circuits, and it is envisaged that the system described in this paper will present a framework to allow these techniques to be explored and refined.

8. References

[1] W.J. Bainbridge, W.B. Toms, D.A. Edwards, S.B. Furber. Delay-Insensitive, "Point-to-Point Interconnect using *m*-of-*n* Codes", *Proc. 9th International Symposium on Asynchronous Circuits and Systems*, pp 132-140, 2003.

[2] A. Bardsley. "Implementing Balsa Handshake Circuits", *Ph.D. thesis*, Department of Computer Science, The University of Manchester, 2000

[3] K. Batcher, "Sorting Networks and Their Applications", *Proc. of the AFIPS Spring Joint Computing Conference, Vol. 32*, pp 307-314, 1968.

[4] K. van Berkel. "Handshake Circuits - An asynchronous architecture for VLSI programming", *Cambridge International Series on Parallel Computers, Cambridge University Press*, 1993.

[5] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. Sangiovanni-Vincentelli, "Logic Minimisation Algorithms for VLSI Synthesis", *Kluwer Academic Publishers*, 1984.

[6] J. Cortadella, A. Kondratyev, L. Lavagno, C. Sotiriou. "Coping with the Variability of Combinational Logic Delays". *Proc. International Conf. Computer Design*, 2004.

[7] G. De Micheli, R. Brayton, A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines", *IEEE Trans. on Computer-Aided-Design*, Vol. 4, No. 3, 1985.

[8] D. Edwards, A. Bardsley, "Balsa: An Asynchronous Hardware Synthesis Language", *The Computer Journal*, vol 45, no 1, 2002.

[9] A.J. Martin. "The Limitations to Delay-Insensitivity in Asynchronous Circuits", *6th MIT Conference on Advanced Research in VLSI Processes*, 1990.

[10] S.R. Nassif, "Modeling and Forecasting of Manufacturing Variations", *Proc 6th Asia South Pacific Design Automation Conference*, 2001.

[11] van Overveld, W.M.C.J., "On arithmetic operations with M-out-of-N codes", *Computing Science Notes, no 85/02*, Dept of Mathematics and Computing Science, Eindhoven University of Technology.

[12] S. Piestrak. "Membership Test Logic for Delay-Insensitive Codes", *Proc. 4th International Symposium on Asynchronous Circuits and Systems*, 1998.

[13] L.A. Plana, P.A. Riocreux, W.J. Bainbridge, A. Bardsley, J.D. Garside, S. Temple. "SPA - A Synthesisable Amulet Core for Smartcard Apllications" *Proc. 8th International Symposium on Asynchronous Circuits and Systems*, 2002.

[14] C. Seitz. "System Timing", Chapter 7 in C.A. Mead and L.A. Conway, editors, Introduction to VLSI systems, *Addison-Wesley*, 1980.

[15] W. B. Toms "Synthesis of Quasi-Delay-Insensitive Datapath Circuits". *PhD. thesis*. Dept. of Computer Science, Manchester University, 2005.

[16] V.I. Varshavsky, ed. "Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems", *Kluwer Academic Publishers*,1990.

[17] T. Verhoeff. "Delay Insensitive Codes - an Overview." *Distributed Computing Vol. 3*, 1988.

[18] Q. Zhang, G. Theodoropoulos, "Modelling SAMIPS: A Synthesisable Asynchronous MIPS Processor", *Proc. of the 37th Annual Simulation Symposium*, pp 205-212, 2004