An Evaluative Study of RT Component Libraries

Pradip K. Jha, Nikil D. Dutt and Daniel D. Gajski

Technical Report# 93-11
March 31, 1993

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
Phone: (714) 856-8059
FAX: (714) 856-4056
Email: pradip@ics.uci.edu

Abstract

The system-level design process typically involves refining a design specification down to the point where each of the system's components is described as a block diagram or netlist of abstract Register-Transfer (RT) level components. Although no standard set of RT components seems to exist across different design methodologies and backend technologies, on closer examination, we see that there indeed does seem to be a universally accepted set of RT-components that are used in the initial phase of design refinement, much before its implementation in a particular target technology. In this report, we describe the need for such a standard RT component set, describe such a parameterized library of standard (or generic) RT components, and evaluate its utility in the system design process. We survey several backend technology libraries, and study the relative coverage of the generic RT component library with respect to these target technology libraries. We then describe the problem of high-level technology mapping, and illustrate this process for a few RT components. Finally, we perform a set of experiments on the HLSW92 benchmarks to evaluate the usefulness of generic RT component libraries. In particular, we compute the overhead incurred by using a generic RT component library over directly using the technology-specific components for the selected benchmark designs. Our preliminary results indicate that the penalty in using the generic components is quite low (approximately 10%), and is more than compensated by the advantages of designing with a generic RT component library.

Contents

1 Introduction 1

2 Generic RT Component Libraries and RT Technology Mapping 2

3 High-level Library Mapping 4

4 Generic Component Coverage 6

4.1 Coverage Across Different Technology Libraries 6

4.2 Coverage Across Different Parameters 7

4.3 Coverage Across Different Component Classes 7

4.4 Components Not Present in GENUS 11

4.5 Summary 12

5 Component mapping 12

5.1 Canonical Component Representation 12

5.2 Priority functions 13

5.3 Mapping Algorithm and Heuristics 14

5.3.1 Component Mapping Algorithm for ALU 14

5.3.2 Component Mapping Algorithm for Register Files 17

6 Experiments and Results 17

6.1 Example Designs 17

6.2 Experiments and Analysis 23

7 Summary 26

8 Acknowledgements 26

A Generator-wise comparison 28

B Component mapping details 37

B.1 High-level mapping for ALU 37

B.1.1 ALU specification 37

B.1.2 Bit-width mapping 37

B.1.3 Function mapping 37
B.1.4 Control mapping 39
B.1.5 Status signal mapping 39

C Designing with different libraries **42**

List of Figures

1 Functional mapping of an ALU . . . 3

2 Functional decomposition of an ALU . . . 3

3 High-level library mapping of an ALU . . . 4

4 Mixed level mapping: (a) Design level (b) Component level . . . 5

5 Coverage of GENUS components across parameters . . . 8

6 Coverage of GENUS components(combinational) . . . 9

7 Coverage of GENUS components(sequential and miscellaneous) . . . 10

8 Generators(features) not in GENUS library . . . 11

9 Flowchart for mapping an ALU . . . 15

10 An example for ALU mapping:(a) Required ALU(R) (b) TS ALU(T) (c) Realized ALU . . . 16

11 Flowchart for Reg-file mapping . . . 18

12 An example for Reg-file mapping: (a) Required reg-file(R) (b) TS reg-file(T) (c) Realized reg-file . . . 19

13 Block diagram of the Am2901 microprocessor . . . 20

14 Block diagram of the Am2910 microprogram sequencer . . . 21

15 Block diagram of SRT interface . . . 22

16 Block diagram of CB interface . . . 24

17 Comparison of different designs(gate count) . . . 25

18 Generator-wise comparison . . . 29

19 Generator-wise comparison . . . 30

20 Generator-wise comparison . . . 31

21 Generator-wise comparison . . . 32

22 Generator-wise comparison . . . 33

23 Generator-wise comparison . . . 34

24 Generator-wise comparison . . . 35

25 Generator-wise comparison . . . 36

26 Set of functions for an ALU . . . 38

27 Bit-width mapping for an ALU . . . 39

28 Generating arithmetic functions from an adder . . . 40

29 Realizing extra logic functions in an ALU 40
30 Realizing comparison functions in ALU 41
31 Design of 2901 with GENUS components 42
32 Design of 2901 with VTI components 43
33 Design of 2901 with Toshiba gate array components 44
34 Design of 2910 with GENUS components 45
35 Design of 2910 with VTI components 46
36 Design of 2910 with Toshiba gate array components 47
37 Design of SRT with GENUS components 48
38 Design of SRT with VTI components 49
39 Design of SRT with Toshiba gate array components 50
40 Design of Circular Buffer with GENUS components 51
41 Design of Circular Buffer with VTI components 52
42 Design of Circular Buffer with Toshiba gate array components 53

1 Introduction

Present-day design methodologies involving schematic capture and simulation require the system designer to partition, refine and specify a design as an interconnection of components drawn from a vendor's library. These components can vary in their level of complexity from simple logic gates, to sequential components such as counters and registers, to arithmetic blocks such as ALUs, and all the way up to complex components such as CPU cores. However, the register-transfer (RT) level is a common design entry point that is supported by most of the existing CAD tools on the market. Furthermore, the RT-level has had a long history of use as a design entry point, as evidenced by the frequent use of TTL databook component names by designers, as well as in digital system design courses outlined in standard textbooks and taught at schools. We also note that most data sheets for product specifications (either being designed, or after they have been designed) are often composed of register-transfer schematics typically drawn up by system level designers.

Component sets and libraries play an important role in the context of synthesis; well defined component sets at the input and output are critical for the successful realization of any synthesis tool. We typically use *generic* components to specify the input or intermediate results of synthesis, and map the design into components from a *technology library* [GDWL92]. For instance, logic synthesis uses generic components such as simple logic gates (e.g., AND, OR, INVERT) at the input and for intermediate synthesis steps, but the last step of logic synthesis involves technology mapping of the generic design into components drawn from a technology library (e.g., complex CMOS gates, or a different logic gate family such as NOR-NOR). Generic component sets facilitate technology independence, and allow the capture of a design in a standard form that can be retargetted to different libraries (or technologies) without changing the input description. Of course, technology independence needs to be coupled with good technology mapping strategies that can effectively map generic designs to target library components with low overhead.

Although RT-level components are commonly used in specifying, documenting, refining and synthesizing designs, there doesn't seem to exist a standardized set of RT components that can facilitate unambiguous documentation, communication and design use. This is in contrast to the logic-level, where the designs can be expressed as netlists of well-understood standard components such as the equivalent 2-input NAND or NOR gate. The lack of a standardized RT-level component set is a serious roadblock to elevating the design process beyond the RT-level and will affect the capability of effectively synthesizing large-scale system designs in an efficient manner.

Another important requirement for system-level design is the capability of specifying the design once, but using this specification to predict technology-specific design characteristics (e.g., area, speed, power) for different target implementations. System-level designers would like to perform early design space exploration by delaying binding of system-level components to a particular technology or implementation, but need the capability of rapid technology projection for different target libraries. The concepts of delayed binding, technology projection and effective estimation for system-level design cannot be performed without the support of a well defined component set and associated tools for technology mapping and prediction.

With increasing interest in high-level synthesis and higher-level design methods, the need has thus evolved for a well defined generic RT component set, along with schemes for mapping these generic RT components to technology-specific components at different levels. This report describes a generic RT component library *GENUS* and provides an evaluative study of this generic RT library with respect to different target technology libraries. We also introduce the problem of high-level

technology mapping, where generic RT components are mapped to technology library components of equal or similar complexity. High-level technology mapping is useful if the target library contains highly optimized complex RT blocks that can be used directly in a system-level design. The use of such higher-level, optimized components may yield a better design than if each generic component was decomposed to lower-level technology primitives. High-level technology mapping thus requires both a good coverage of the technology library by the generic component set, as well as well defined cost functions to guide the mapping process. We illustrate this process of high-level technology mapping on some high-level synthesis benchmarks and attempt to measure the penalty incurred by using generic RT components during design refinement.

The report is organized as follows. Section 2 describes different approaches for implementing generic components in technology-specific designs, while Section 3 briefly defines the overall problem of high-level library mapping. In Section 4, we survey the coverage of generic components in the GENUS generic library relative to different backend technology libraries. Section 5 illustrates the task of high-level component mapping with two examples. Section 6 describes the experiments we performed to evaluate the effectiveness and the overhead incurred by specifying designs using generic RT component libraries and the high-level library mapping approach. Section 7 concludes with a summary and identification of the open problems.

2 Generic RT Component Libraries and RT Technology Mapping

GENUS [Dutt88] [Dutt91] is a parameterized generic RT component library developed at U.C. Irvine for use with simulation and high-level synthesis tools. A design that is initially specified in a hardware description language such as VHDL can be implemented with RT components drawn from the GENUS library, either through manual design refinement or using high-level synthesis tools such as state schedulers, component allocators, component and connectivity binders [LiGa88].

After the design has been specified as an interconnection of generic RT components, we have to map the design to a layout technology so as to satisfy design constraints such as area, time, etc. Several paths exist for this technology mapping phase. This section outlines each of these paths for technology mapping.

A generic component in the GENUS library can be mapped to technology specific components at different levels, depending on the complexity of the building blocks used. We identify four approaches to component mapping based on different levels of building blocks used to realize a generic component:

Functional mapping At the lowest level, a generic component's functionality can be described using Boolean equations for the transformation of the inputs into outputs. These equations can then be mapped to low-level technology-specific components such as gates, flip-flops and latches [BRSW87] [Keut87] [VaGa88]. For example (Figure 1), an ALU can be described with Boolean equations for each output(O0, OCOUT and OZERO) that use the inputs I0, I1, ICIN and C. Each of these equations can be mapped to components from a logic library (e.g., NOR gates). This type of functional mapping is also commonly called logic-level technology mapping.

Functional decomposition At a slightly higher level, a generic component can be mapped to

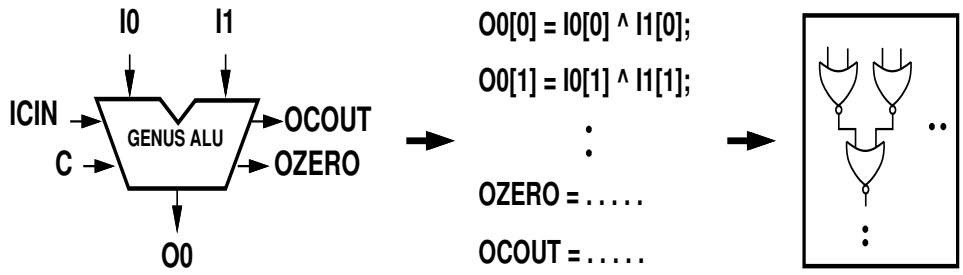


Figure 1: Functional mapping of an ALU

MSI-level blocks from a technology library. The given generic component is functionally and/or structurally decomposed into smaller building blocks. This decomposition is organized hierarchically and can be represented as a tree in which the generic component is at the root of the tree. Leaves of the tree consist of the MSI/SSI level blocks from the technology library, while intermediate nodes represent hierarchically decomposed components of the design. DTAS [Kipp91] follows this approach. Figure 2 shows a decomposition tree for a generic ALU. This ALU is realized by composing the leaf cell blocks (such as 4-bit adders, FAs, MUX2, gates) from a technology-specific component library.

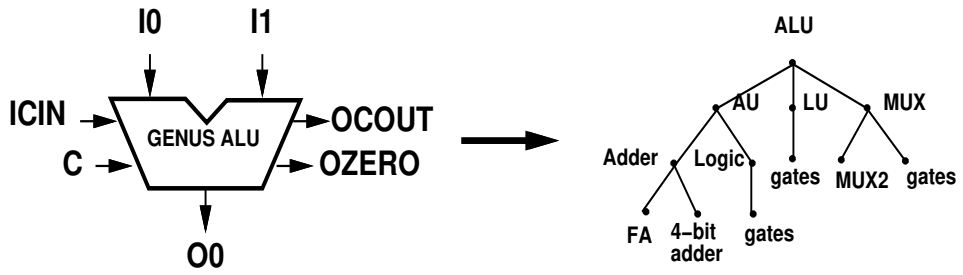


Figure 2: Functional decomposition of an ALU

High-level library mapping At the highest level, an abstract component can be mapped directly to a library-specific component at the same level. In this approach, a generic ALU will be mapped to a technology-specific(TS) ALU, a Reg-file to a TS Reg-file, and so on. Extra logic may have to be added around the TS component when the functionality of the TS component does not exactly match that of the generic component. Figure 3 illustrates the high-level mapping for an ALU.

General Case In the general case, the mapping of a generic component to a TS component may

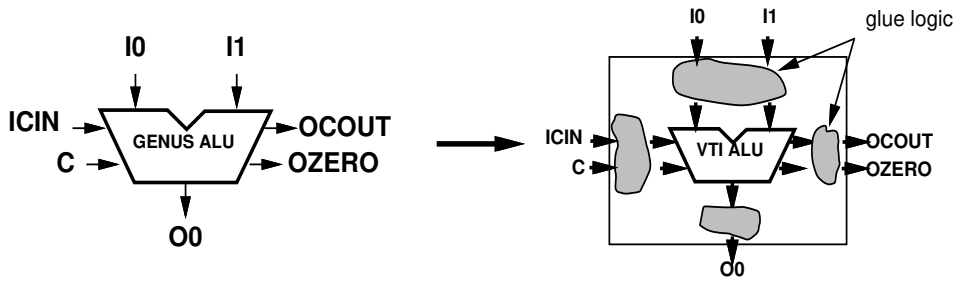


Figure 3: High-level library mapping of an ALU

require a combination of the three approaches mentioned above. This mapping approach could be used at different levels of granularity as shown in Figure 4. At the design level, different components of the structural netlist of a design could be mapped using different levels of mapping. At the RT-component level, a particular component itself could be mapped to the library components using different levels of mapping. For example, the arithmetic functions of an ALU could be mapped on to an MSI component ADDER, whereas the logic functions could be expressed as a set of Boolean equations and then mapped to TS gates.

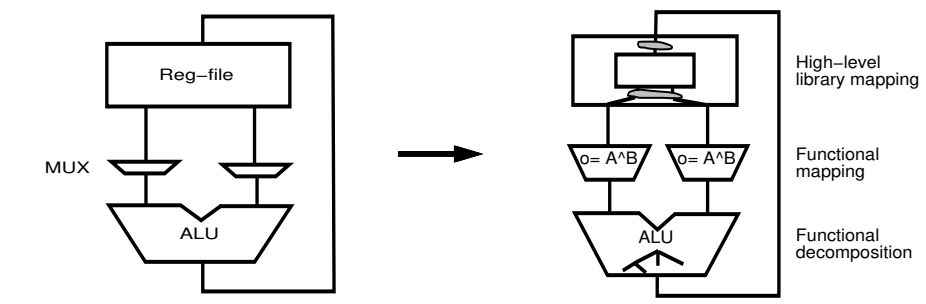
In this report, we concentrate on the third approach, that is, **High-level library mapping**.

3 High-level Library Mapping

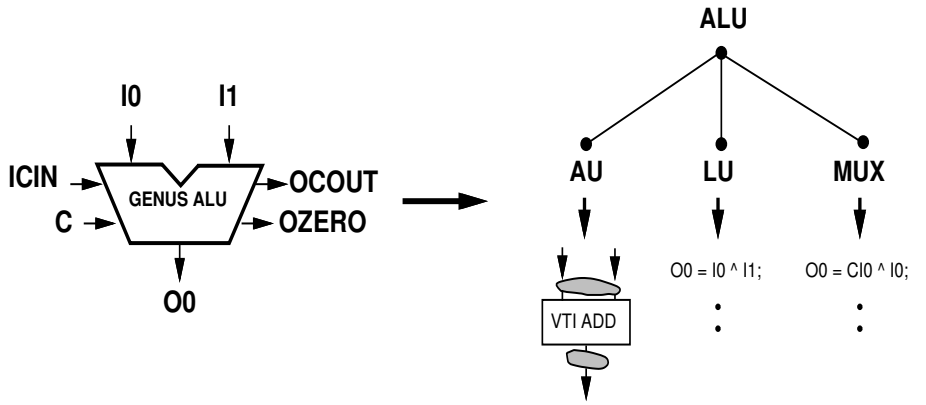
High-level library mapping, as described in the previous section, refers to the mapping of a generic component to a TS component of similar complexity. It involves finding a TS component with similar functionality that may need additional logic to account for the differences. In order to do an effective job of high-level library mapping, we need to perform the following steps.

The first task is to survey the set of the components in the generic library and the corresponding set of components available in the technology library. This survey will highlight the **coverage** of the generic component set with respect to different technology libraries. Hence we need to examine several back-end design methodologies (e.g., synthesis tools, custom design). After tabulating the list of components available in various technology libraries, we need to compare each component's semantics, functions performed, port names, size, and other attributes. This survey of component coverage is a prerequisite for high-level mapping, since it identifies feasible technology-specific candidates for the high-level mapping approach. The issue of relative coverage of component libraries can be translated into the following questions:

1. What percentage of the generic components have corresponding TS components onto which they could be mapped with low overhead?
2. Given a generic component and a similar TS component, in what aspects and how much do they differ?



(a)



(b)

Figure 4: Mixed level mapping: (a) Design level (b) Component level

3. Is high-level mapping feasible? If the set of generic components does not effectively cover the TS library, then the high-level mapping approach may not be useful.

Once we have the information on the generic component and TS component set, we need to specify a **mapping strategy**, consisting of a set of mapping heuristics for each class of component generator. This strategy will require the capture of relevant information describing both the generic component to be mapped and a set of TS components that are candidates for the mapping. A specific algorithm then needs to be developed to find the TS component with the best match and, if necessary, to add extra logic to the TS component selected so that the resultant design mimics the functionality of the given generic component.

Finally, we need to evaluate the resultant design. We need to define a cost function that quantifies the total cost involved in realizing generic components. The cost function measures the effectiveness of high-level library mapping and helps us evaluate the practicality of this approach as well as its merit relative to the other techniques such as functional decomposition and functional mapping.

4 Generic Component Coverage

In this section, we present the results of our survey of library coverage with respect to different TS libraries that use varying layout styles for component implementation. In particular, we examined the following layout styles:

Standard cell In standard cell implementation cells are placed in pitchmatched rows with between-channel routing. It is typically used for implementing random-logic.

Bitslice In the bitslice method, for each component single-bit design is created and an n-bit component is realized by replicating the single-bit design n times.

Gate array In the gate array method, the component's logic is realized by specifying the connectivity of a prespecified array of gates.

Field programmable gate array(FPGA) Except for the fact that the connectivity is programmable in the field, this is similar to the gate array technology.

The first two layout styles typically result in more compact designs at the cost of longer design cycles, while the gate array and the FPGA styles provide a quick method for prototyping designs.

4.1 Coverage Across Different Technology Libraries

We considered the following technology libraries in our survey:

VTI Datapath Compiler[VTI91] This compiler generates components parametrized by bit-width. It can generate gate-level design or full-custom compiled layout.

Cascade Digital Library[Casc92] Cascade's EPOCH digital library also defines a compiler that implements parametrized components. From the parametrized specification of a component, a standard cell or bitslice implementation could be generated.

Toshiba Gate Array Library[Tosh90] The Toshiba gate array library contains a specific set of non-parametrized components.

XBLOX[XBLO92] Xilinx provides a set of RT-components that are parametrized by bit-width. These components can be directly mapped into Xilinx's FPGA library.

4.2 Coverage Across Different Parameters

Figure 5 pictorially illustrates the coverage of GENUS components across various parameters relative to different TS libraries. Please refer to [Dutt88] for the set of different parameters associated with various components. Each column in Figure 5 represents a parameter and each row shows a TS library. We examined the following parameters:

Set of components With respect to the set of components, VTI and Cascade have fairly good coverage, followed by Toshiba gate array.

Size of a component Since the VTI and Cascade libraries are parametrized by bit-width, GENUS covers these libraries fairly well in terms of the size of components. On the other hand, the Toshiba library has a fixed set of components and hence results in poor coverage with respect to component size.

Set of functions Although most of the components in these libraries do provide many of the functions mentioned in the GENUS library, TS components can perform specific subsets of these functions. They do not support the flexibility of GENUS where a multi-function unit (e.g., an ALU) can perform any subset of functions out of the full set of the functions for that generator.

Semantics Most of the components in the TS libraries closely follow the semantics with respect to the behavior of the GENUS components.

Style and Type GENUS provides an extensive set of styles and types for each generator. The TS libraries we studied did not cover all these types and styles.

Port names GENUS follows a specific convention in selecting the set of input-output ports and their names. Similarly, each TS library has its own port naming convention. This results in a mismatch between the port names of TS library components and GENUS components.

4.3 Coverage Across Different Component Classes

While Figure 5 gives an overall indication of the coverage across different technology libraries, it is useful to examine the relative coverage with respect to each component type. Figures 6 and 7 illustrate the relative coverage across different component classes. [Dutt88] contains a detailed list of the set of GENUS components and their classes. We can broadly classify the RT components into the following classes:

Combinational components These include primitive gates, shifters and arithmetic/logic functions. With respect to combinational components, Cascade's library is closest to GENUS, followed by the VTI compiler, and finally the Toshiba gate array library.

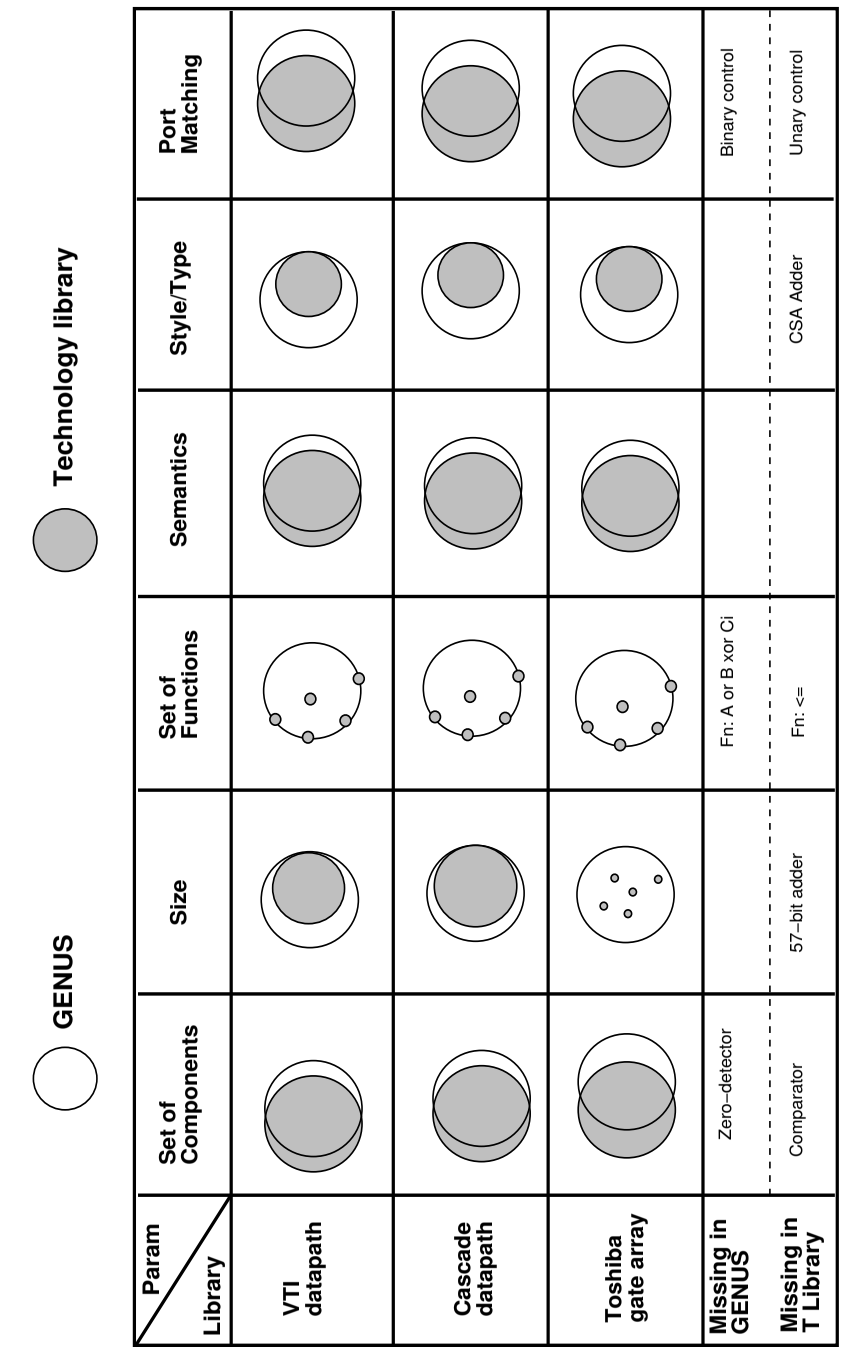


Figure 5: Coverage of GENUS components across parameters

Generator / Library	ALU Related fns (ALU, ADDSUB, LU, COMPARATOR)	Shifting fns (BARRELSHIFTER, SHIFTER)	Primitive fns (GATES)	Complex fns (MULT)	Others (MUX, SELECTOR, DECODE, ENCODE)
VTI datapath					
Cascade datapath					
Toshiba gate array					
Missing in GENUS	Fn: A or B xor Ci	Fn: Funnel	ANDOR	Fn: Mult/add	
Missing in T Library	Fn: GEQ	Shifter			ENCODE

Figure 6: Coverage of GENUS components(combinational)

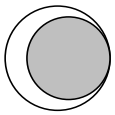
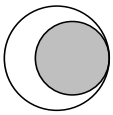
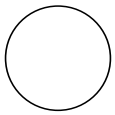
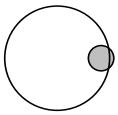
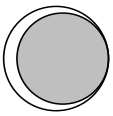
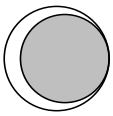
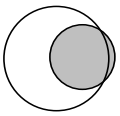
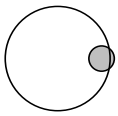
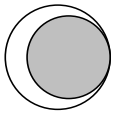
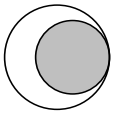
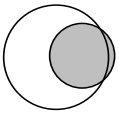
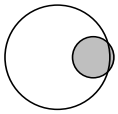
Generator / Library	Sequential components			Miscellaneous components
	Shift register, Counter	Memory, Reg–file	Stack, FIFO	(Buffer, Bus, Concat, ...)
VTI datapath				
Cascade datapath				
Toshiba gate array				
Missing in GENUS			Diff Semantics	Diff Semantics
Missing in T Library	Limited styles	Limited ports	Stack	Concat

Figure 7: Coverage of GENUS components(sequential and miscellaneous)

Sequential components This class contains all storage elements such as registers, counters, memories, etc. With respect to this class, VTI and Cascade provides pretty good coverage followed by the Toshiba gate array library.

Miscellaneous components This class contains all the miscellaneous components in GENUS. Several components in this class are "virtual" components that are present for simplifying synthesis, and may have no meaning in hardware (e.g., the bit-manipulation operations *concat* and *extract*). As a result, these components show poor coverage with respect to the technology libraries.

Appendix A provides a detailed comparison across different component generators.

4.4 Components Not Present in GENUS

GENUS does not cover all the components present in the TS libraries. Some of these components are completely missing in GENUS, whereas others may have extra features that are not available in a corresponding GENUS component. Figure 8 lists these components. Future extensions of GENUS may need to incorporate some of these components or features if they are justified by their frequency of use.

	Specific features		
Generators	**VTI compiler**	**Cacade compiler**	**Toshiba gate array**
ALU	Fn: $\overline{A} + \overline{B}$	Lots of arithologic functions	Lots of arithologic functions
Zero–detector	available	available	
Mult	Different bit–width for the two inputs	Mult–add fn available	
Complementer		2's complement	
Gates	complex gates	complex gates	complex gates
Lookahead carry generator			available
Bilbo		available	
Barrel–shifter		fn: funnel	

Figure 8: Generators(features) not in GENUS library

4.5 Summary

In summary, TS libraries that are parametrized(Cascade and VTI) provide fairly good coverage for GENUS components. The Toshiba gate array library provides components of specific sizes; components of other sizes have to be built from the available components. The other major difference was the availability of a specific set of functions in realm of multifunction components. We also observed a common problem with mismatches in the port names.

5 Component mapping

From the previous section, we observed that generic components cover the technology libraries we surveyed fairly well; this establishes the feasibility of the high-level library mapping approach. In this section we define the actual mapping strategy.

Given a generic component specification, the task of high-level component mapping is the problem of selecting a TS component that is "closest" to a generic component. The closeness function can be formulated to require the addition of minimal logic to the TS component to realize the generic component; the "closeness" is thus defined in terms of some cost function that encourages functional similarity and that penalizes additional logic for dissimilarities.

In order to effectively perform mapping, we first need to define a **canonical representation** that captures the essential features of all the components across different libraries including the generic component library. Secondly, we need to define some **priority function** to guide the component selection. Thirdly, we need to define the actual **mapping algorithm** for realizing a generic component from a TS component. In the remaining part of this section, we discuss these steps with some examples.

5.1 Canonical Component Representation

We need a representation scheme that covers the essential features of the component set in a standard form so that similarities and differences can be identified, and a standard set of mapping algorithms can be developed for each component. This representation must cover not only the generic component set, but also the component sets across various technology libraries.

We propose a representation similar to one followed by the GENUS library [Dutt88]. Components of similar behavior are grouped together into generators such as ALU, REGISTER, MUX, etc.. Each generator has an associated set of parameters; a component is instantiated by specifying values to each of the parameters associated with the corresponding generator. The GENUS parameter set has to be extended to cover the features of different libraries.

The canonical component representation must capture the following typical parameters and attributes:

Bit-width, which characterizes the size of the component in terms of the input and output bit-widths. The canonical representation should be able to specify a particular bit-width (for a particular component) as well as a range of bit-widths (for the generator as a whole).

Set of Functions, for multi-functional components.

Ports, that specify the mode (input, output or input-output) and the features (latched, inverted, etc.)

Data Representation, in formats such as sign magnitude and 2's complement.

Control Specification, for multi-functional units indicating the encoding (unary or binary) and the function table.

Status signals, that are generated as a result of primary operations in the component (e.g., zero-bit or overflow).

Besides these important parameters, we need to represent some parameters that are specific to certain classes of generators. These include port-specification for reg-files and memories, enable-lines for some components, and component implementation styles such as ripple-carry, carry-lookahead and counter styles.

We illustrate the elements of the canonical representation using the ALU generator. A typical ALU generator has the following parameters:

Bit-width Theoretically, this could be any positive integer value.

Ports Names of the input and output ports have to be specified.

Set of Functions A general ALU could perform any subset of the following functions: 16 arithmetic functions, 16 logical functions and 6 comparison functions.

Control Specification The control encoding and functionality could be specified in tabular form.

Status Signals An ALU could generate any subset of these signals: {Cout, Overflow, ZERO, SIGN}.

Data Representation The input and output data could be any of these formats: sign magnitude, 2's complement, 1's complement, etc.

A detailed description of the canonical ALU representation is given in Appendix B.

5.2 Priority functions

After a set of candidate technology-specific components are identified, we need to find a TS component that has the best match with respect to the generic component. This requires a search process guided by a set of goals or a priority function. Each priority function for the mapping can be defined in terms of the generator's parameters (e.g., bit-width, set-of-functions, control encoding, etc.) that specify a component. For instance, the closeness function for an ALU generator can be defined in terms of its parameters: bit-width, set-of-functions, control signals, status signals, data representations, etc.

The cost function used to guide the search process in the mapping can be formulated in several ways. One scheme may have the user rank the parameters in order of decreasing importance, so that components that match parameters with higher priority are selected first, followed by components that are matched on lower priority parameters. Another scheme can require the parameters to

be pre-ranked according to their general importance. For example, for the ALU generator, we select components that are closer in bit-width, followed by functional similarity. A general scheme could create a cost function that is a composite function of the all the parameters, represented as a weighted sum of the different parameter values, where weights specify their relative importance.

While specifying the relative priority, one should remember that some of the parameters may have an overriding effect on the feasibility of component realization. That is, if a parameter value is not satisfied by the TS component, the generic component cannot be realized at all. For example, in case of the register-file generator, if the TS register-file does not have a sufficient number of ports as required by the generic component, the mapping becomes infeasible, since the TS register-file does not have enough parallelism in its ports to support simultaneous access to the required number of RF locations. Such overriding parameters should be given special consideration while defining the priority function.

5.3 Mapping Algorithm and Heuristics

Once the candidate TS components and the prioritized mapping cost functions are identified, the next task is the actual mapping algorithm required to realize efficient implementation of the generic component. After selecting a proper TS component, this mapping algorithm adds extra logic to compensate for the mismatches that arise because of the differences in the parameter values of the two components.

Let R be the required component and T the TS component to be used for realizing R. Specifically, we need to add logic that makes up for the differences between R and T with respect to following parameters:

Bit-width If Bit-width(T) is greater than Bit-width(R), then a proper subset of the data width of T has to be selected to realize R.

Set-of-functions If T is missing some of the required functions, logic has to be added to generate those missing functions.

Control lines A decoder may be required to map the control lines of R to the control lines of T.

Status signal Extra logic may have to be added to T to generate the extra status signals present in R.

Data representation Some logic might be required to match the data representation of R and T.

We briefly describe component mapping algorithms for two components: the ALU and the Reg-file. The detailed mapping procedure can be found in Appendix B.

5.3.1 Component Mapping Algorithm for ALU

An ALU is a multifunctional unit characterized by the following parameters: bit-width, set-of-functions, style, control lines, status signals, and data representation. The set of possible values that these parameters can take is shown in Appendix B.

For the ALU generator, each of these parameters needs to be factored into the overall priority function. We could either rank the parameters or specify a weighted sum of them as a priority function for the search procedure. Note that some of the operations of the ALU (such as ADD or a minimum of two comparison functions) and most of the status signals (except ZERO and SIGN), when required, are overriding parameters that determine the feasibility of the mapping. Details of the priority function are described in Appendix B.

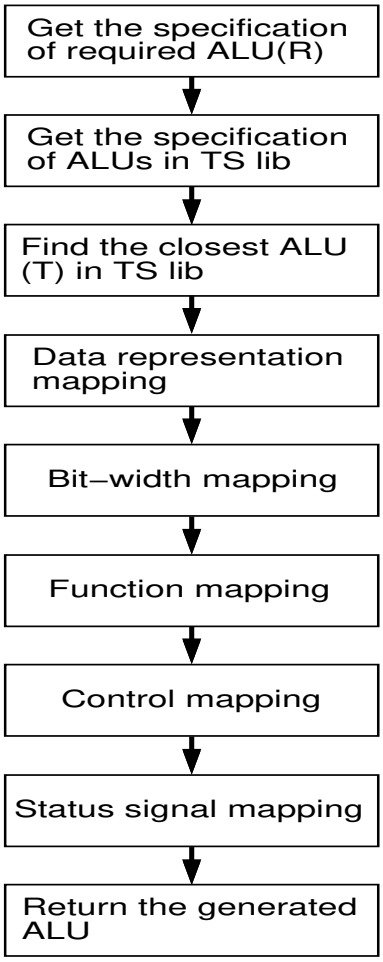


Figure 9: Flowchart for mapping an ALU

Figure 9 shows the overall flowchart of the mapping algorithm for the ALU generator. We start by obtaining the canonical representation of the required component R and the TS ALU's. Then we find the closest ALU T in TS library. Next, we start padding T with logic to account for the differences in R and T. For each of the parameters, we specify some rules to make up the differences. We start with data representation mapping, followed by bit-width mapping, function mapping, control mapping and finally status signal mapping. Each of these steps either generates some additional logic for T if feasible, or returns a false value signifying the inability of mapping

R to T. The detailed steps are given in Appendix B.

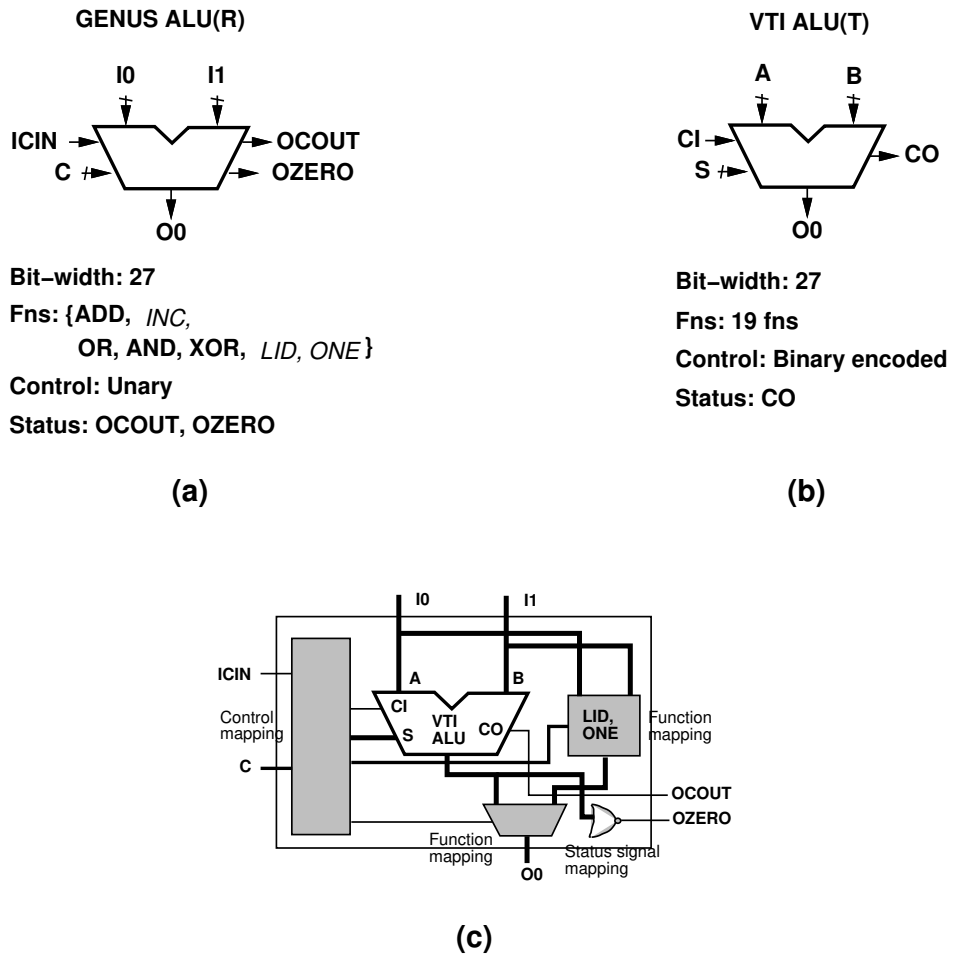


Figure 10: An example for ALU mapping:(a) Required ALU(R) (b) TS ALU(T) (c) Realized ALU

Figure 10 shows the application of this mapping algorithm to an example where the required component R is a GENUS ALU(Figure 10(a)) that can perform seven functions (ADD, INC, OR, AND, XOR, LID, ONE) and has two status signals (OCOUT, OZERO). The ALU R's bit-width is 27, and the data is represented in 2's complement notation. The control lines for the ALU R are unary encoded. We wish to map this ALU to an ALU from the VTI Datapath Library. The closest ALU T that could be found in the VTI library (Figure 10(b)) is a 27-bit ALU that performs 19 functions and provides only one status signal OCOUT. Specifically, it is missing one arithmetic function INC, two logic functions(LID, ONE) and a status signal OZERO. T's control lines are binary encoded. Figure 10(c) shows the final realization of R using T. The shaded boxes represent the extra logic that has to be added in order to realize R. For this example, we do not require data representation mapping, and bit-width mapping. Function mapping is performed by the square,

shaded box on the right and a mux at the bottom. The shaded rectangular box on the left of Figure 10(c) takes care of control mapping, while the multi-input NOR gate at right-bottom corner of Figure 10(c) generates the extra status signal OZERO. With this we have ALU R realized using the VTI Datapath Library.

5.3.2 Component Mapping Algorithm for Register Files

A Reg-file is a sequential component characterized by the following parameters: bit-width, num-words, read-ports, write-ports, enable, and clock.

As with the ALU generator, each of these parameters determine the priority function for the search process. For a reg-file, the read-ports and write-ports are the overriding parameters that determine the feasibility of the mapping: if the TS library does not have reg-files with a sufficient number of the ports as required, we cannot generate a register-file configuration to support the desired data parallelism.

Figure 11 shows the overall mapping strategy for a reg-file. Once again we start by obtaining the canonical representation of the required reg-file R and TS library reg-files. Next, we find the reg-file T in the TS library that most closely matches R. If required, we then expand the bit-width and num-words. Then we do port mapping, followed by address mapping, clock mapping and finally rest of the ports are handled.

Figure 12 shows the application of the above algorithm for generating a reg-file(R) of size 768*72 that has one R/W port and one R port. In the technology library, we have reg-files of different sizes with 2 R ports and 2 W ports. The reg-file that is finally realized is shown at the bottom of Figure 12. This example involves bit-width expansion, num-words expansion, port matching and address matching.

6 Experiments and Results

In order to evaluate the practicality of high-level library mapping, we performed some experiments with various designs derived from the HLSW92 benchmark suite [DuRa92]. The goal was to test our approach on designs of various sizes, and that encompass different sets of components so as to exercise the major component types in the GENUS component set. In our preliminary set of experiments, the designs varied in complexity from a few hundred gates to a couple of thousand gates. We chose these designs so as to cover different GENUS component classes that have varying attributes. The mapping experiments were performed with respect to two different technology libraries: the VTI Datapath Compiler[VTI91] and the Toshiba Gate Array library [Tosh90]. We chose these libraries since they had published gate counts for their databook components, thereby allowing us to compare the effectiveness of mappings for different designs.

6.1 Example Designs

We considered RT-level designs from different categories such as processors, DSP and interface circuits. In particular, we considered the following designs:

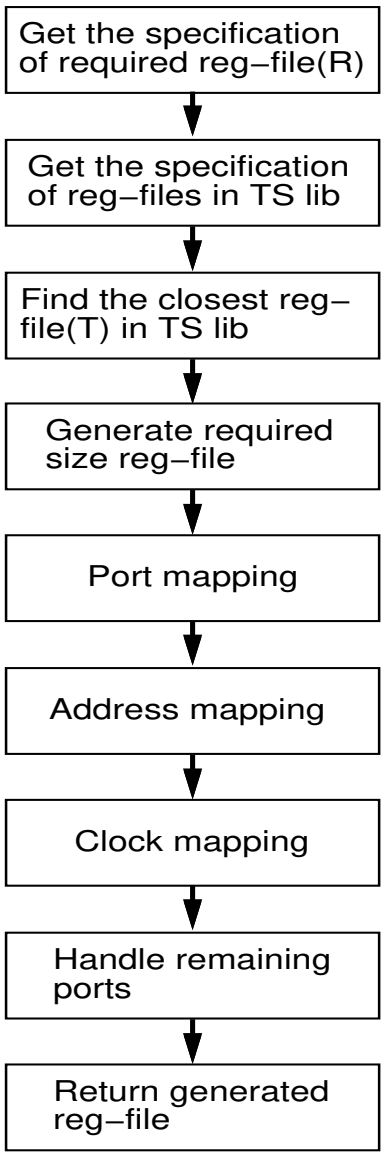


Figure 11: Flowchart for Reg-file mapping

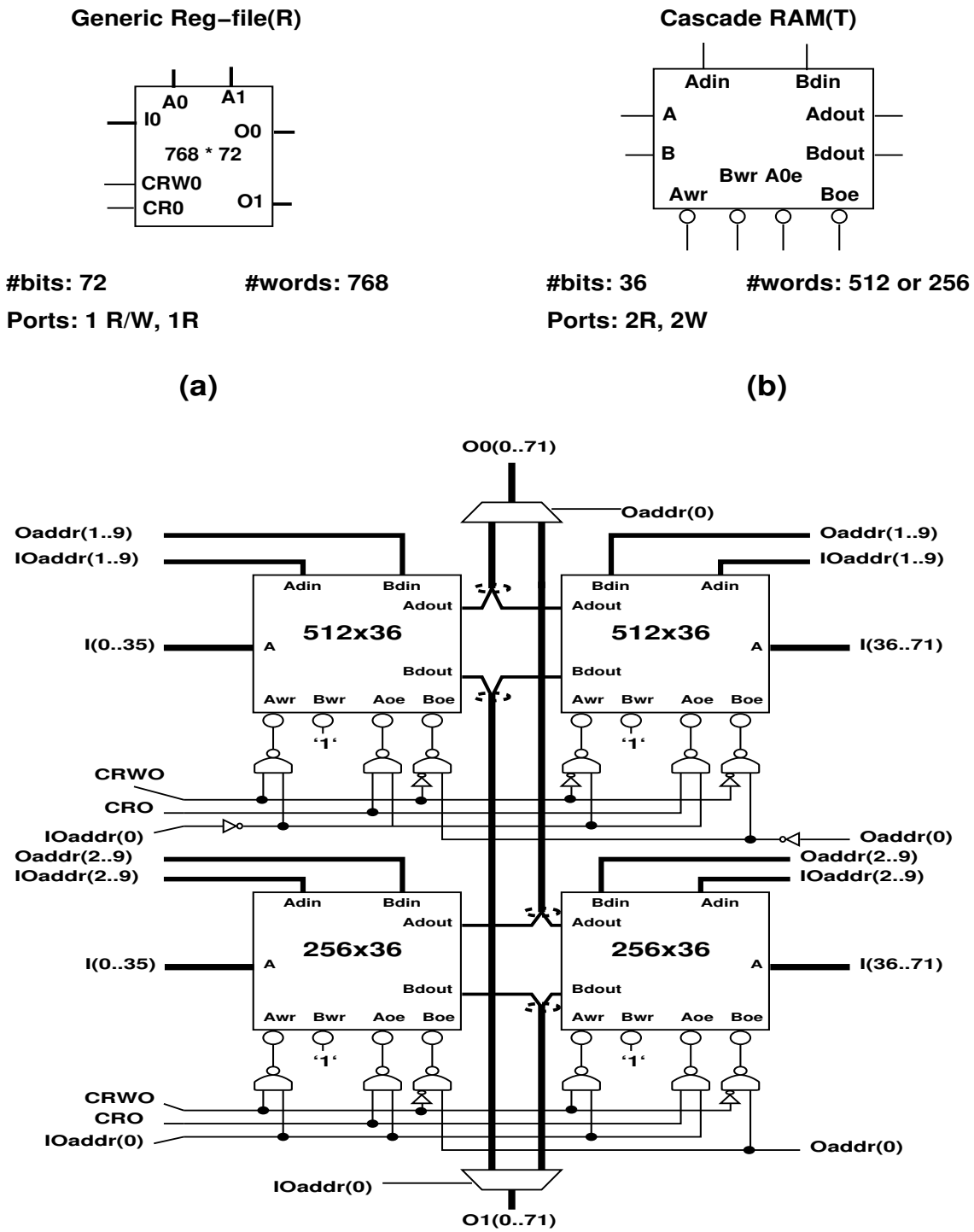


Figure 12: An example for Reg-file mapping: (a) Required reg-file(R) (b) TS reg-file(T) (c) Realized reg-file

The AM2901 Microprocessor The AM2901 is a four-bit microprocessor slice which can be used in CPUs, peripheral controllers, and programmable microprocessors. The device, as shown in the block diagram (Figure 13), consists of a 16-word by 4-bit two-port RAM, a high-speed ALU, and the associated shifting, decoding and multiplexing circuitry. The nine-bit microinstruction word is organized into three groups of three bits each and selects the ALU source operands, the ALU function, and the ALU destination register. Further details can be found in [Am2901].

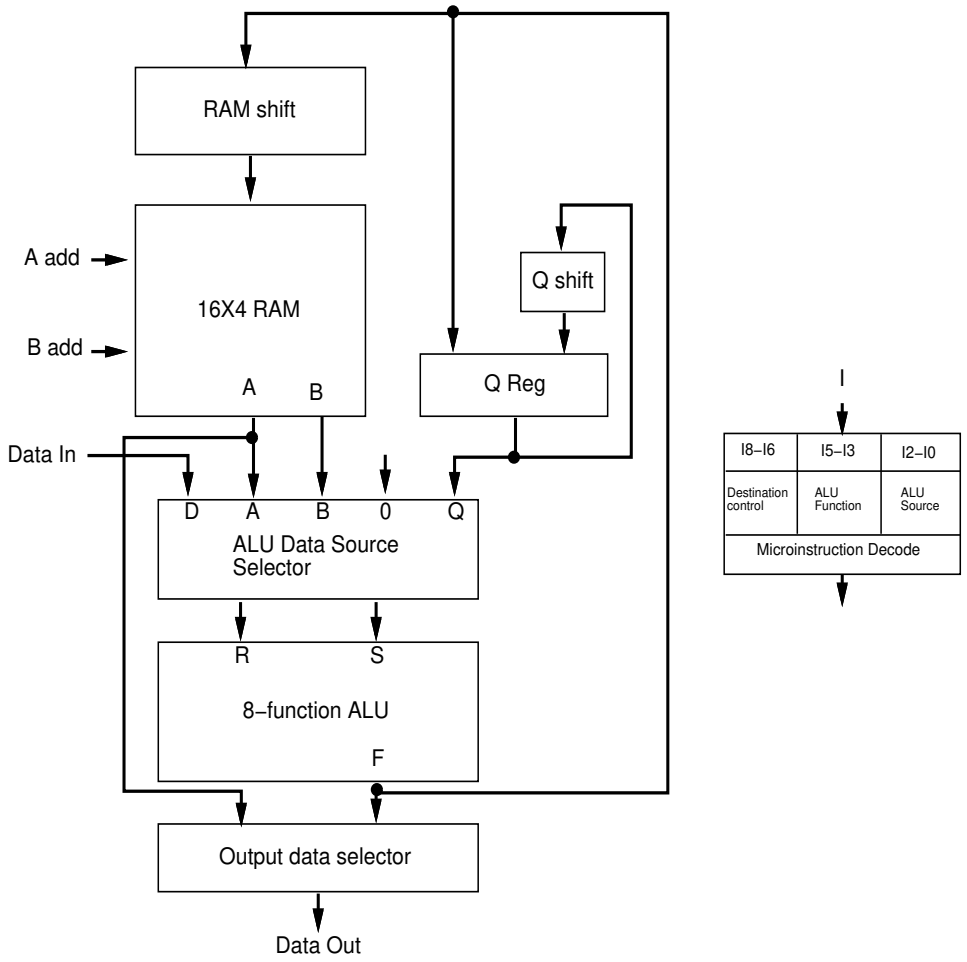


Figure 13: Block diagram of the Am2901 microprocessor

The AM2910 Microprogram Controller The AM2910 is a 12-bit microprogram address sequencer intended for controlling the sequence of execution of microinstructions stored in microprogram memory. Figure 14 shows the block diagram of AM2910. It consists of a last-in, first-out stack of depth 5. The multiplexer selects the next address from these sources: direct data input, top of stack, counter and microprogram counter. The next address could be loaded to the microprogram counter unmodified or after being incremented. The device

provides 16 instructions that select the address of the next microinstruction to be executed. The output is tri-state buffered. Further details can be found in [Am2910].

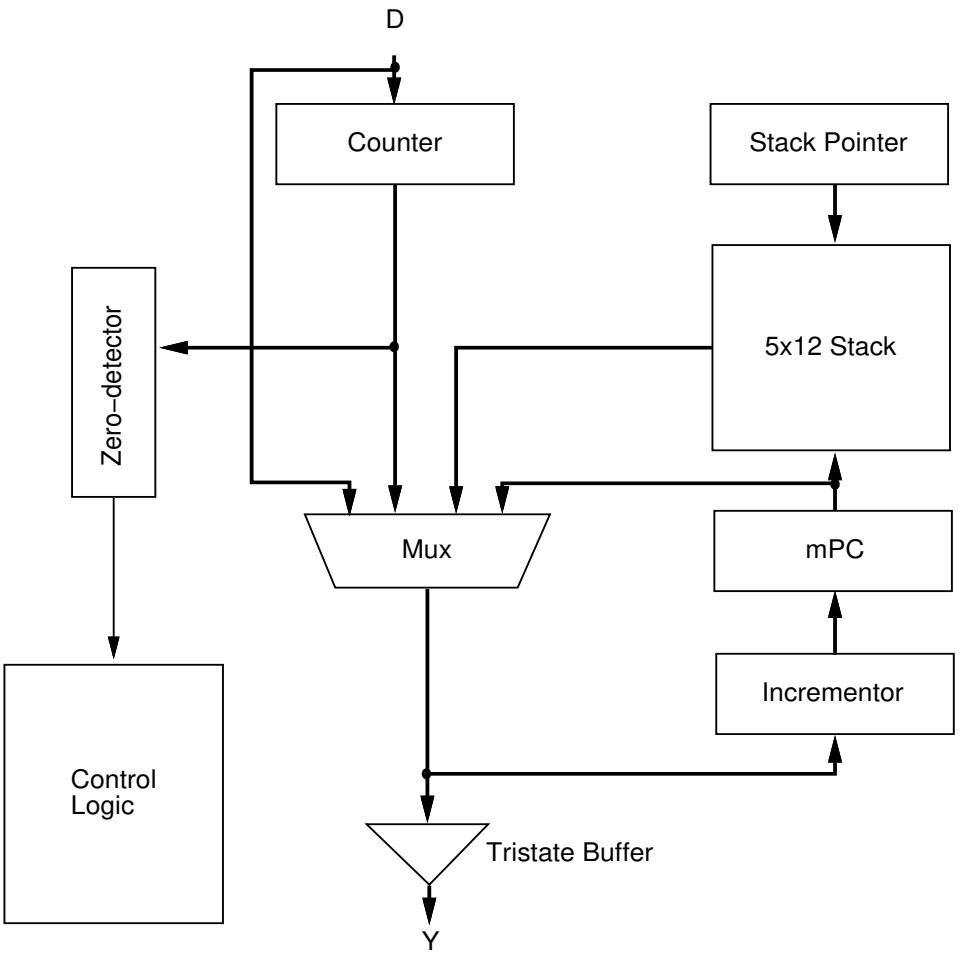


Figure 14: Block diagram of the Am2910 microprogram sequencer

The Serial Receiving and Transmitting (SRT) Interface The SRT interface is used for serial communication between a CPU and its peripheral devices. The main functions of the SRT are to transmit and receive serial data. When transmitting, the parallel data is converted into a serial stream and transmitted serially. When receiving, the data stream is received serially and converted into a parallel format.

Figure 15 shows the block diagram of the SRT interface. The core of the circuit consists of four registers: Control register(CW), Status register(STAT), Transmission shift register(tran-buffer), and Receiving shift register(rec-buffer). Besides these registers, it has some control logic that handles the handshake protocol with the CPU and the peripheral devices. Further details can be found in [Li1993].

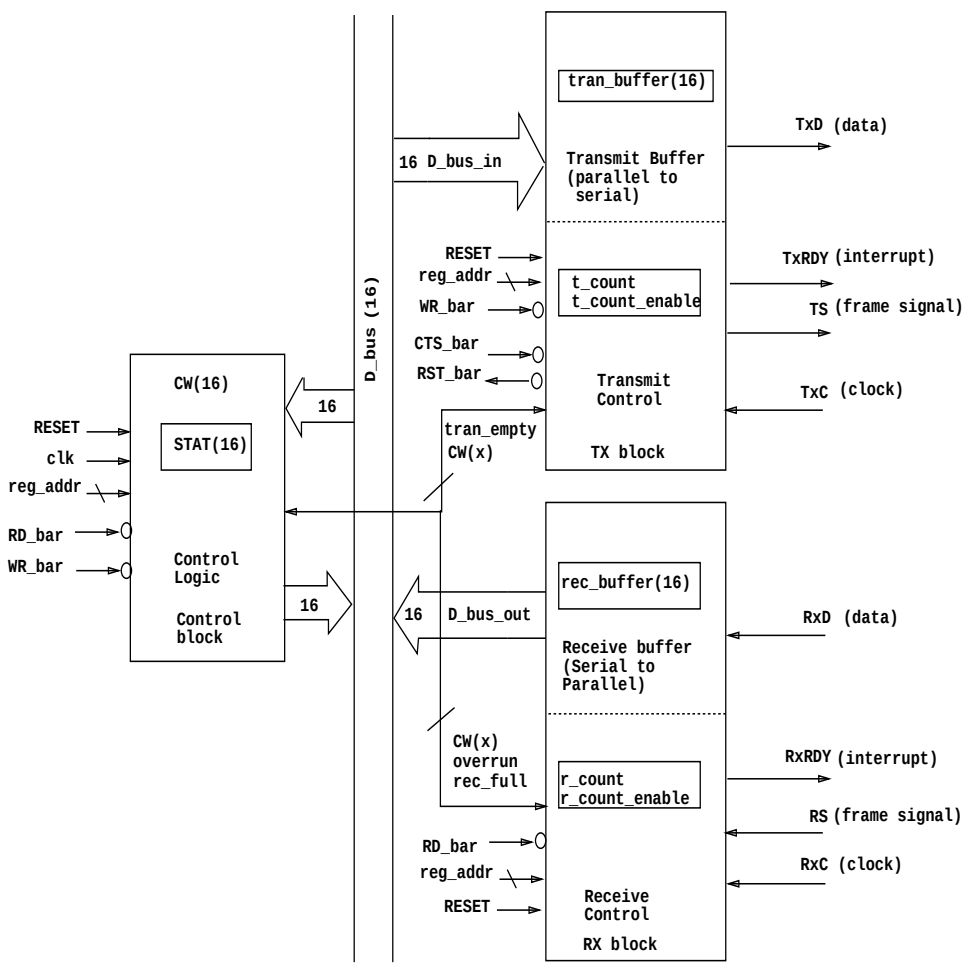


Figure 15: Block diagram of SRT interface

The Circular Buffer(CB) interface The CB interface is used between a DSP(digital signal processor) and a host CPU. The main function of the CB is to transfer data between the DSP and the host. The data transfer consists of two steps. First, the data to be transferred is stored in the circular buffer. Next, the stored data is then read by the recipient (DSP or the host CPU).

Figure 16 shows the block diagram of the CB interface. It consists of a 8-word by 32-bit FIFO with two address pointers: head-pointer and tail-pointer. The FIFO can be accessed per byte. Besides the FIFO, we have a status register (circ-status) and some logic to handle the handshake protocol between the host and the DSP. Further details can be found in [Li1993].

6.2 Experiments and Analysis

In our experiments, we designed each of these circuits using three different paths, as summarized by the results in Figure 17. First, we designed the circuit using only generic components from the GENUS library (I). Then we designed the same circuit using TS library components only (II). Finally, we took the design with GENUS components and mapped each of these generic components to the TS library components (III). The goal was to examine the penalty incurred by designing with a generic component library, followed by technology mapping, as opposed to directly implementing the designs with TS-components. Detailed designs for each of these circuits are presented in Appendix C.

For each of these designs, we calculated the total gate count. For a GENUS library component, we used the component's boolean equations for calculating the gate count measured in terms of the total number of 2-input AND, 2-input OR and NOT gates. For the technology libraries, a gate is equivalent to the layout area of 4-transistors.

Figure 17 tabulates the gate-counts for various designs across different libraries. For each design and each technology library, we present the gate-count for the three methodologies (I, II, III). Also, we present the percentage difference (in terms of the gate-count) between implementing a component in the technology library (II) and mapping the GENUS component design to the technology library (III). This percentage gives a measure of how much overhead is incurred by using generic components from GENUS.

In Figure 17, we observe that the percentage difference between the two design methodologies (II and III) vary from 0.00% to 12.53%. For the SRT interface, the two designs (II and III) are very close in gate-count. This is because the SRT circuit is fairly simple and primarily uses lower-level components such as logic gates and flip-flops that have good coverage in technology libraries. The lack of complex RT components enables a very simple and effective mapping with a resulting overhead that is very low.

On the other hand, consider the mapping of the 2901 microprocessor and the CB interface to the Toshiba gate array library. There is a significant difference in gate-counts for the two methodologies (II and III). These designs use higher-level components such as ALUs and register-files which have a larger mismatch with respect to the technology library components.

Based on these preliminary experiments, we observe that not much penalty (generally $\leq 10\%$) is incurred in using generic components first and then performing high level library mapping. This supports our hypothesis that high-level mapping is feasible and practical for the designs we examined.

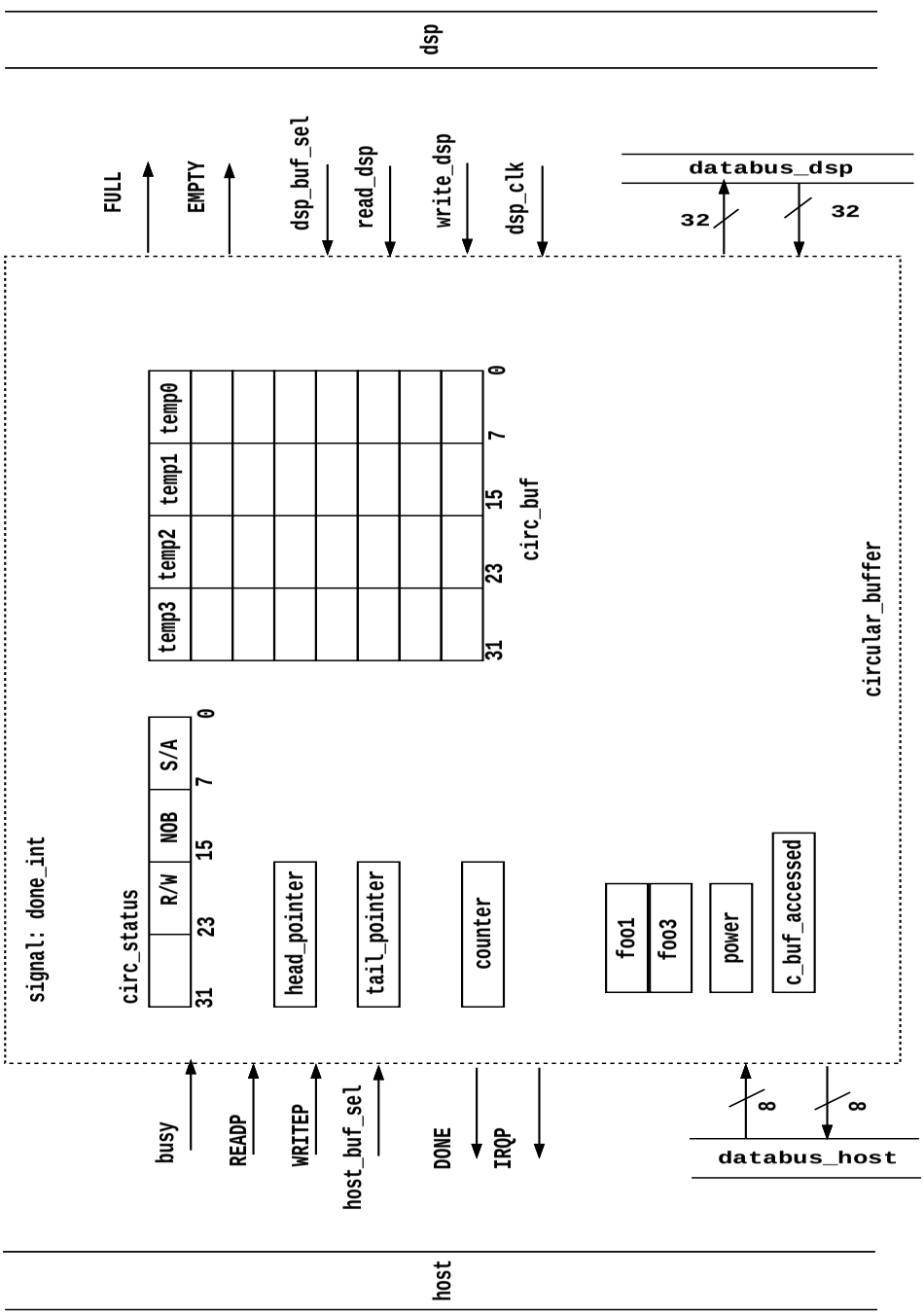


Figure 16: Block diagram of CB interface

Design		GENUS Library (I)		VTI Datapath			Toshiba Gate Array		
Name	**Components**			**Using VTI Comps (II)**	**GENUS Mapping (III)**	%over–head	**Using TGA Comps (II)**	**GENUS Mapping (III)**	%over–head
		#(Dff)	#(gate)	#(gate)	#(gate)		#(gate)	#(gate)	
2901	Regfile	72	248	134	134	0	1225	1225	0
	ALU		215	100	100	0	84	84	0
	ALU glue logic		22	10	54	444	21	37	76.2
	Q–register	8	35	28	28	0	38	38	0
	Muxes		104	80	80	0	117	117	0
	Mux glue logic		22	15	15	0	14	103	635.7
	Buffer		32	24	24	0	24	24	0
	Total	80	678	391	435	**11.25**	1523	1628	**6.89**
SRT	Register	41	244	369	36	0	388	388	0
	Counter	8	38	184	184	0	118	118	0
	Buffer		96	96	96	0	64	64	0
	Decoder		6	6	6	0	6	6	0
	Control FF	4		36	36	0	36	36	0
	Control gates		30	44	56	27.3	62	62	0
	Total	53	414	735	747	**1.63**	674	674	**0.00**
CB	Regfile	256	534	624	624	0	1032	1032	0
	Regfile glue logic	24	96	251	251	0	20	250	1150
	Counter	10	44	230	230	0	156	156	0
	Comparator		48	40	42	5	28	28	0
	Decode/encode		6	6	6	0	0	18	—
	Buffer		192	144	144	0	144	144	0
	Register	32	128	288	288	0	304	304	0
	Mux		208	176	176	0	208	208	0
	Control FF	5		45	45	0	72	72	0
	Control gates		15	28	30	7.1	15	15	0
	Total	327	1271	1832	1836	**0.22**	1979	2227	**12.53**
2910	Regfile	60		192	192	0	776	776	0
	Refile glue logic		35	27	52	92.59	5	5	0
	Stack–pointer	3	46	73	73	0	73	73	0
	Micro PC	12	37	84	96	14.28	84	111	32.14
	Incrementer		228	60	60	0	135	135	0
	Counter	12	163	276	276	0	207	207	0
	Mux		84	60	60	0	108	110	1.85
	Zero–detector		12	24	24	0	44	44	0
	Buffer		96	36	36	0	96	96	0
	Total	87	701	832	869	**4.44**	1528	1557	**1.89**

Figure 17: Comparison of different designs(gate count)

7 Summary

In this paper, we performed an evaluative study of different RT component libraries. In particular, we motivated the need for generic RT component libraries, defined the high-level technology mapping problem, surveyed the relative coverage of the GENUS generic RT library with respect to some technology libraries and illustrated the high-level technology mapping problem for two RT components: ALU and Register-File. Finally, we performed some experiments on some high-level synthesis benchmarks to study the penalty incurred by using generic RT components followed by technology mapping, versus directly implementing the designs in the technology libraries. Our preliminary results are encouraging, indeed even promising, since the maximum overhead we observed was in the range of 10% for area.

We believe that the benefits of using a standard component set such as GENUS greatly outweigh the small penalty that may be incurred during technology mapping to target libraries. We have several directions to pursue for future work. One line of work will investigate the penalty incurred for other design criteria (e.g., delay), and will study the effectiveness of this high-level technology mapping approach across a broader range of design examples. Another line of work will investigate efficient techniques for high-level technology mapping of RT components. We can then attempt to integrate different technology mapping schemes (i.e., functional mapping, functional decomposition and high-level mapping) and compare the effectiveness of these schemes across a range of designs.

8 Acknowledgements

This research was supported in part by NSF grant #MIP9009239 and in part by SRC contract #92-DJ-146. We are grateful for their support.

References

[Am2901] "Am2901c: Four-bit Bipolar Microprocessor Slice," *Advanced Micro Devices, Sunnyvale, California,* 1993.

[Am2910] "Am2910A: Microprogram Controller," *Advanced Micro Devices, Sunnyvale, California,* 1993.

[BRSW87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. R. Wang, "MIS : A multiple-level logic optimization system," *IEEE Transaction on Computer-aided Design,* pp1062-1081, November 1987.

[Casc92] "Cascade Design Automation Databook," *Cascade Design Automation, Bellevue, WA,* 1992.

[DuKi91] N. D. Dutt and J. R. Kipps, "Bridging High-Level Synthesis to RTL Technology Libraries," *Proc. 28th Design Automation Conference,* June 1991.

[DuRa92] N. D. Dutt and C. Ramachandran, "Benchmarks for the 1992 High Level Synthesis Workshop," *Technical Report 92-107, University of California at Irvine,* 1992.

[Dutt88] N. D. Dutt, "GENUS:A Generic Component Library for High Level Synthesis," *Technical Report 88-92, University of California at Irvine,* 1988.

[Dutt91] N. D. Dutt, "Generic Component Library Characterization for High-Level Synthesis," *Proc. VLSI Design 91,* January 1991.

[GDWL92] D. Gajski, N. Dutt, A. Wu and S. Lin, "High-Level Synthesis: Introduction to Chip and System Design," *Kluwer Academic Publishers* 1992.

[Keut87] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching" *The 24th ACM/IEEE Design Automation Conference,* 1987.

[Kipp91] J. R. Kipps, "An Approach to Component Generation and Technology Adaption," *Ph D Dissertation, University of California at Irvine,* December 1991.

[LiGa88] J. S. Lis and D. D. Gajski, "Synthesis from VHDL," *Proc. IEEE Int. Conf. on Computer Design'88,* pp.378-381, 1988.

[Li1993] J. Li, "VHDL Modeling for Silicon Compilation," *M.S. Thesis, University of California at Irvine,* 1993.

[Mano88] M. M. Mano, "Computer Engineering Hardware Design," *Prentice-Hall, Inc. New Jersey,* 1988.

[Tosh90] "Toshiba ASIC Gate Array Library," *Toshiba Corporation, Tokyo, Japan,* 1990.

[VTI91] "VDP300 CMOS Datapath Library," *VLSI Technology, Inc., San Hose, California,* November 1991.

[VaGa88] N. Vander Zanden and D.D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *Proc. 25th Design Automation Conference,* June 1988.

[XBLO92] S. H. Kelem and J. P. Seidel, "Shortening the Design Cycle for Programmable Logic Devices," *IEEE Design and Test of Computers,* pp40-50, 1992.

A Generator-wise comparison

Combinational components

GENUS	VTI compiler			Cascade compiler		
Generator	**Generator**	**Features**	**Differences**	**Generator**	**Features**	**Differences**
ADDSUB	VDP1ADD001 VDP1ASB001 VDP1ASB002 VDP1SUB001	High speed adder High−speed add−sub High−speed add−sub (B−A−1) High−speed sub	Only one type(CLA) Control signals mismatch	Adder	RC, CLA, CSA Optional ov signal	No sub, no control
ALU	VDP1ALU001 VDP5ALU001	Expanded 2901ALU (high speed) Expanded 2901 ALU (slow speed, RC)	Only fixed fn set binary control Fn mismatch Intermediate carry available No comparison fn Status OZERO missing	ALU Incrementer/ Decrementer	16 arithmetic fns, 16 boolean fns 4−bit CLA rippled Incrementer, Decrementer, both	Some new fns Fixed set of fns binary control Comparison fns missing Status OZERO missing
BARRELSHIFTER	VDP3BSH001	Barrel−shifter/rotator	Port mismatch Single fn Unary shift control	Barrelshifter	Fine fns: ROT, ASHR, SHL, SHR, Funnel	Only one fn at a time Port mismatch Fn ASHR missing
COMPARATOR				Magnitude comparator Equality Checker	Two fns: GT & LT Fn: EQ	Specific fn set No control Port mismatch

Figure 18: Generator-wise comparison

Combinational components

GENUS	Toshiba gate array		
Generator	**Generator**	**Features**	**Differences**
ADDSUB	FA1X	1–bit full adder	Specific bit–width
	HA1x	Half adder	Specific function
	FAS2	2–bit add–sub	Binary control
	FA2	2–bit adder	Extra G input, P output
	FA4	4–bit adder	Only one type(CLA)
	XADD32	32–bit CLA adder	
	XF283	4–bit binary full adder with fast carry compatible to 74LS283	
ALU	XALU32	16 arithmetic fns	Specific function set
		16 boolean fns	Specific bit–width
		32–bit CLA	No OZERO signal
		Fns based on 74LS181	Comparison fns missing
	XF181	4–bit ALU	Some new fns
		Comapatible to 74LS181	Binary control
	XFLS381	4–bit ALU	G, P signal
		Compatible to 74LS381	
BARRELSHIFTER	XBRL16	16–bit barrelshifter	Missing fn: ASHL
		4 fns: SHL, SHR, ASHR, ROT	Specific bit–width
			Binary control
			Specific fn set
COMPARATOR	CMP4	4–bit equality comparator	Specific bit–width
	CMP8	8–bit equality comparator	Specific fn set
	MAG2	2–bit expandable magnitude comparator	Port mismatch
		3 fns: GT, EQ, LT	
	MAG2H	2–bit magnitude comparator	
	MAG4	4–bit expandable magnitude comparator	
	XF688	8–bit equality comparator	
		Compatible to 74LS688	

Figure 19: Generator-wise comparison

Combinational components

GENUS	VTI compiler			Cascade compiler		
Generator	**Generator**	**Features**	**Differences**	**Generator**	**Features**	**Differences**
DECODE/ENCODE				Decoder	Generic random logic in truth−table format	Different format
DIV						
LU				LOU	16 boolean fns	Binary control
MULT	VDP3MLT001 VDP3MLT002 VDP3MLT003 VMLT03	MxN multiplier 2−stage pipelined Variably−piped multiplier NxM multiplier	Different bit−width for the two inputs Not all types are present Slight port mismatch	Simple multiplier High speed multiplier	2−stage pipeline multiplier/adder	Different bit−width for the two inputs Specific value of pipeline stage Extra functionality Port mismatch(set & clear)
MUX		2..8x1 mux with unary select 2x1, 4x1 mux with binary select	Binary select Limited num−inputs Inverting type is missing	Multiplexer	2x1,3x1,4x1 non−inverting, 2x1 inverting	Limited num−inputs Limited inverting/non−inverting
SHIFTER	VDP3SHF001	Up/down shifter	Binary control Fixed set of fns Extra enable signal			
SELECTOR						
GATES(GNOT, GAND, GOR, GNOR, GAND, GXOR, GXNOR, AND, NAND, OR, NOR, XOR, XNOR)		2−input gates(NOT, AND, NAND, OR, NOR, XOR, XNOR, ORAND, ANDOR)	Fixed num−inputs New type of gates	Gates	not, and2, and3, nand2, nand3, or2, or3, nor2, nor3, xor2, xnor2, andnor,ornand	Limited num−inputs New type of gates

Figure 20: Generator-wise comparison

Combinational components

GENUS	Toshiba gate array		
Generator	**Generator**	**Features**	**Differences**
DECODE/ENCODE		2x4, 3x8, 4x10, 3x8 with address latch 10x4, 8x3 priority encoder	Specific bit–width Function mismatch
DIV			
LU			
MULT	XMPY8 XMPY16	8–bit multiplier 16–bit multiplier	Port mismatch Specific bit–width Mode control lines Only one type
MUX		Several bit–width and num–input Some with latch, inverted output	Limited bit–width Limited num–input Limited inverted output
SHIFTER			
SELECTOR			
GATES	Gates	Various gates	Limited input–width Limited num–inputs

Figure 21: Generator-wise comparison

Sequential components

GENUS	VTI compiler			Cascade compiler		
Generator	**Generator**	**Features**	**Differences**	**Generator**	**Features**	**Differences**
COUNTER	VDP3CNT001 VDP5CNT001	Up/down synchronous counter with load and clear−moderate speed Slow−speed	Fixed type Port mismatch	Ripple counter Synchronous counter	up, down, up/down counter Very close to GENUS counter	Limited fns and options
FIFO				FIFO	#words: 4−128 #bits: 1−128	Extra ports
MEMORY	CRAM01 CRAM02 CRAM03 CROM01	Max #bits = 16,384 One R/W port with enables High−speed CRAM01 High−performance, low power ROM compiler	Port mismatch Restriction in size	Single port RAM High Speed RAM Dual port RAM Multi−port RAM High speed ROM	1 R/W port with enable 2 R/W port with enable 1−5 read ports with enable 1−3 write ports with enable #words:64K #bits: 64	Size restriction Port mismatch Limited number of ports
REGFILE	VDPRGF001	2 read ports, 1 write port Partial words could be read out	Port mismatch Limited size	Register file	1−2 read ports, enable 1 write port with enable	Limited size
REGISTER		Various DFF and latch	Limited set of functions and options Muxed input	DFF, JKFF, LATCH, Shift register		Limited type and fns
STACK						

Figure 22: Generator-wise comparison

Sequential Compononents

GENUS	Toshiba gate array		
Generator	**Generator**	**Features**	**Differences**
COUNTER		Various specific bit−width counter	Fixed bit−width Fixed type
FIFO	XFIxxyy	#words: 16, 32, 64 #bits: 4,5,6,7,..36	Limited num−words, input−width port mismatch Semantic mismatch
MEMORY	RAM−C RAM−E ROM−A/ROM−B	Limited size Single R/W port with write enable Enable(expnadable) Two read ports One write port with enable single port ROM	Limited size
REGFILE	XF670	4x4 regfile with 3−state output	Specific size Tristate output Limited bit−width and fns Tristate output
REGISTER		Variously sized DFF, JKFF with different fns	Muxed input Binary control
STACK			

Figure 23: Generator-wise comparison

Miscellaneous components

GENUS	VTI compiler			Cascade compiler		
Generator	**Generator**	**Features**	**Differences**	**Generator**	**Features**	**Differences**
BUFFER		Various buffers(inverting /non−inverting) of diff strength		Buffer	1x to 20x range inverting/non−inverting	
BUS						
CLOCKGEN						
CONCAT						
DELAY						
EXTRACT						
ONESHOT						
WIREDOR						
PORT						

Figure 24: Generator-wise comparison

Miscellaneous components

GENUS	Toshiba gate array		
Generator	**Generator**	**Features**	**Differences**
BUFFER		Lots of buffers	
BUS			
CLOCKGEN	OSCXc	Oscillator	
CONCAT			
DELAY	YDLY1X YDLY2X YDLY3X	Delay buffer	Specific delay value
EXTRACT			
ONESHOT			
WIREDOR			
PORT			

Figure 25: Generator-wise comparison

B Component mapping details

B.1 High-level mapping for ALU

An ALU is a multifunction unit with multiple control lines and multiple status signals. A specific ALU is characterized by assigning values to these parameters: bit-width, set-of-functions, control-lines, status signals, port specification, data representation. This section describes each of the steps involved for the mapping of an ALU (Figure 9).

B.1.1 ALU specification

Bit-width Theoretically, the input and output data could have any positive integer value as bit-width.

Set-of-functions An ALU could possibly perform any subset of the functions mentioned in Figure 26.

Control lines The mapping of control lines to the functions is specified in tabular form. For each function, we have a distinct set of values on the control lines.

Port Specification An ALU has two data inputs, control lines(including the carry input), a data output and status signal outputs.

Data representation In an ALU, input and output data could be in any of the following forms: sign magnitude, 1's complement, 2's complement.

B.1.2 Bit-width mapping

Given the bit-width of the required ALU(R) and the bit-width of the TS ALU(T), we first decide if the bit-width(R) is mappable on to the bit-width(T). If the bit-width(R) $>$ bit-width(T), then R can be implemented by functional decomposition and is thus not considered an option for high-level technology mapping; we thus return a false value. Otherwise, we map the data inputs and outputs of R to the most significant bits of the corresponding ports of R and disable the rest of bits by assigning '0' to them (Figure 27).

B.1.3 Function mapping

For an ALU, the task of function mapping is the most involved step. Let us assume that T and R are the TS ALU and the required ALU respectively. Let TX, TY, TCI, TCO and TF be the two data inputs, carryin, carryout and data output respectively. Similarly, RX, RY, RCI, RCO and RF be two data inputs, carryin, carryout and data output respectively for R.

Based on the function types, we classify the function mapping into three categories:

Arithmetic functions To realize an arithmetic function, we assume that the TS ALU(T), at the minimum, can perform the 'ADD' function. We provide the heuristics to add extra logic so that the basic adder could be used to perform other arithmetic functions. Figure 28 shows

Arithmetic			Logic		Comparison	
Function name	**Operation**		**Function name**	**Operation**	**Function name**	**Operation**
	CI = 0	**CI = 1**				
DEC	A minus 1	A	ZERO	0	EQ	A = B
	AB minus 1	AB	AND	AB	NEQ	A $\neq$ B
	$\overline{AB}$ minus 1	$\overline{AB}$	RINHI	$\overline{AB}$	LT	A < B
	minus 1	zero	LID	A	LEQ	A <= B
	A plus (A+$\overline{B}$)	A plus (A+$\overline{B}$) plus 1	LINHI	$\overline{A}$B	GT	A > B
	AB plus (A+B)	AB plus (A+B) plus 1	RID	B	GEQ	A >= B
SUB	A minus B minus 1	A minus B	XOR	$\overline{AB}$+$\overline{A}$B		
	A+$\overline{B}$	(A+$\overline{B}$) plus 1	OR	A+B		
	A plus (A+B)	A plus (A+B) plus 1	NOR	$\overline{A+B}$		
ADD	A plus B	A plus B plus 1	XNOR	AB+$\overline{AB}$		
	$\overline{AB}$ plus (A+B)	$\overline{AB}$ plus (A+B) plus 1	RNOT	$\overline{B}$		
	A+B	(A+B) plus 1	RIMPL	A+$\overline{B}$		
	A plus A	A plus A plus 1	LNOT	$\overline{A}$		
	AB plus A	AB plus A plus 1	LIMPL	$\overline{A}$+B		
	$\overline{AB}$ plus A	$\overline{AB}$ plus A plus 1	NAND	$\overline{AB}$		
INC	A	A plus 1	ONE	1		
LSUB	B minus A minus 1	B minus A		$\overline{AB}$ xor CI		
				(A+$\overline{B}$) xor CI		
				($\overline{A}$+B) xor CI		
				(A+B) xor CI		

Figure 26: Set of functions for an ALU

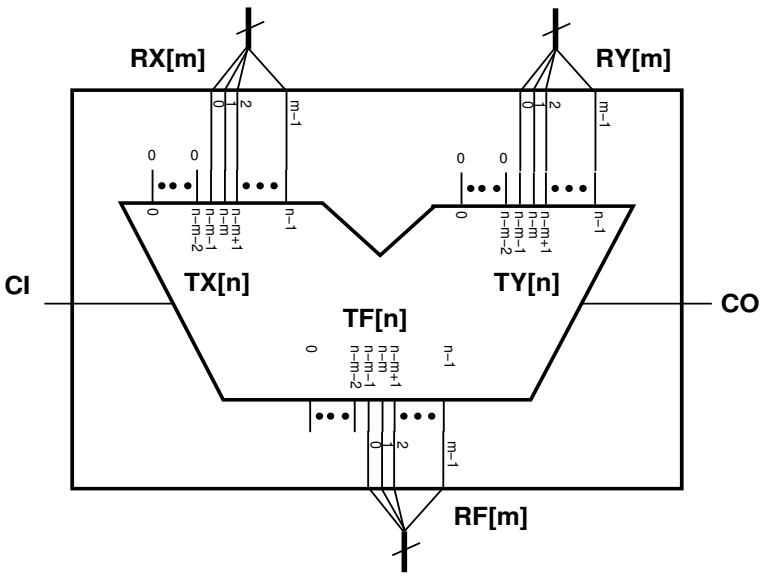


Figure 27: Bit-width mapping for an ALU

the relationship between the ports of R and T for important arithmetic functions. For all those functions that are not available in T and are required, we add the glue logic defined by the table in Figure 28.

Logic functions For the logic functions missing in T, we generate a block that implements these logic functions. The output of the this block is muxed with the original output of the ALU T, to get the final output(Figure 29).

Comparison functions An ALU could possibly perform the following six comparison functions: EQ, NEQ, LT, GEQ, GT, GEQ. If the TS ALU T, can perform any two of these in parallel, that are not complement of each other, the rest could be generated with very little logic. One such example is illustrated in Figure 30.

B.1.4 Control mapping

As mentioned before, the mapping between control lines and the functions they represent, is specified with a table, for both the ALUs R and T. We can use the standard Karnaugh map [Mano88] method to transform the table of R to the table of T.

B.1.5 Status signal mapping

We could generate these status signal with little logic, if they are not already not present in T.

SIGN SIGN is the most significant bit of the output and hence could be easily extracted.

IO / OP	TX	TY	TCI	RF	RCO
SUB	RX	$\overline{RY}$	RCI	TF	TCO
INC	RX	RY	RCI	TF	TCO
DEC	RX	0	RCI	TF	TCO
ADD	RX	RY	RCI	TF	TCO
LSUB	$\overline{RX}$	RY	RCI	TF	TCO

Figure 28: Generating arithmetic functions from an adder

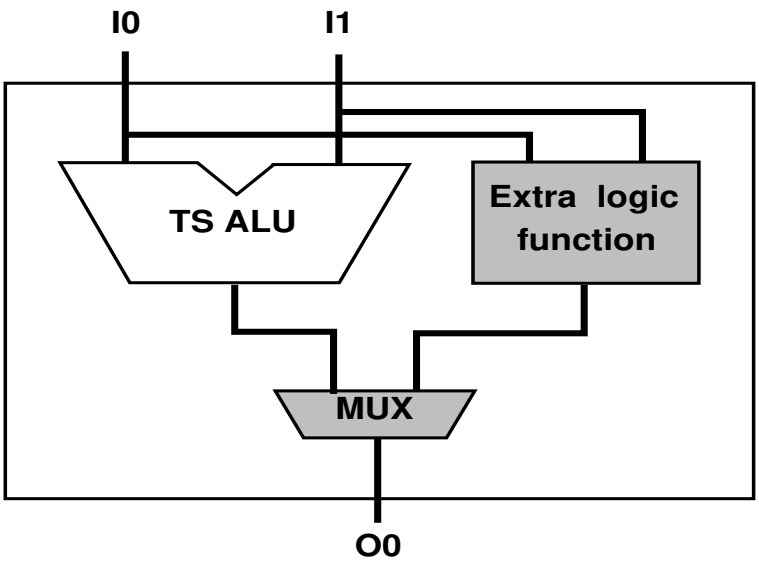


Figure 29: Realizing extra logic functions in an ALU

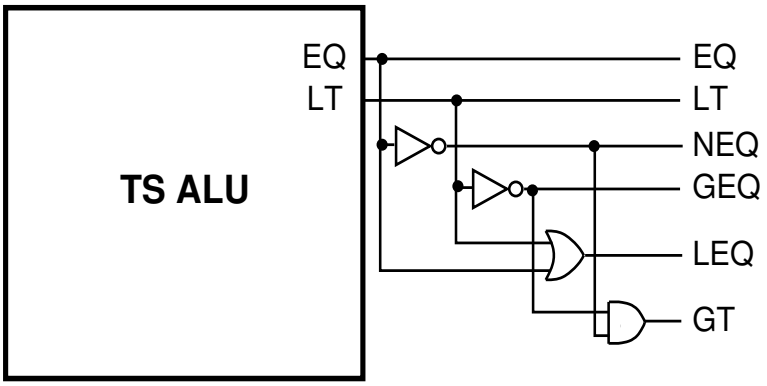


Figure 30: Realizing comparison functions in ALU

ZERO This signal specifies whether all the bits in the output are zero. This could be generated by a NORing all the bits of the data output together.

If the other status signals such as Carryout (CO) and Overflow (OV) are not specified in T, but are specified in the generic component R, then we may incur a large overhead in generating these signals.

C Designing with different libraries

In this appendix, we present the implementations of the example designs mentioned in section 6 using different component libraries. Specifically, we illustrate implementation for four designs: 2901 Microprocessor [Am2901], 2910 Microprogram Sequencer [Am2910], Serial Receiver and Transmitter [Li1993], and Circular Buffer [Li1993]. For each design, we first generate the implementation with generic components from GENUS [Dutt88]. Then, we implement the same design using components from two technology libraries: VTI Datapath Compiler [VTI91] and Toshiba Gate Array [Tosh90]. The shaded components in the figures that follow represent the extra or different component that had to be used in following two different strategies: 1) implementing the design directly with the technology specific components, and 2) taking the implementation with GENUS components and then mapping the generic components used in the design to TS components. In order to avoid visual clutter, we illustrate only the major components and the datapath routing.

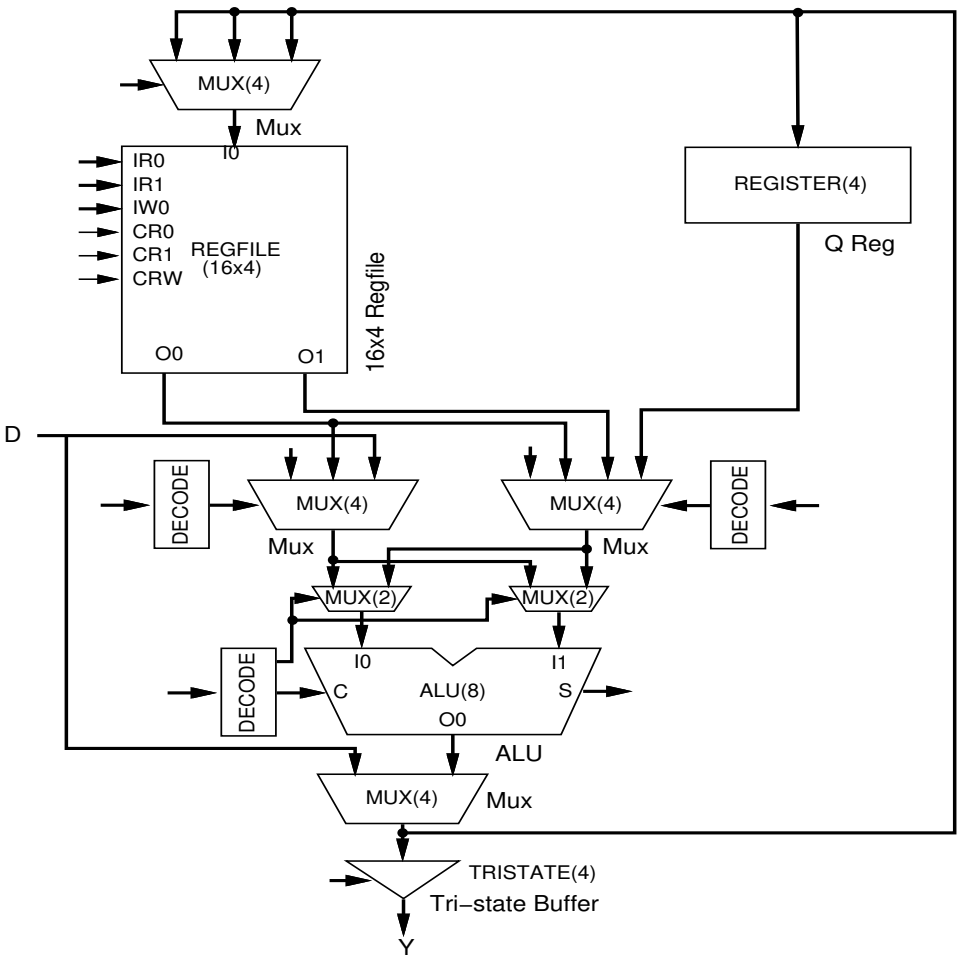


Figure 31: Design of 2901 with GENUS components

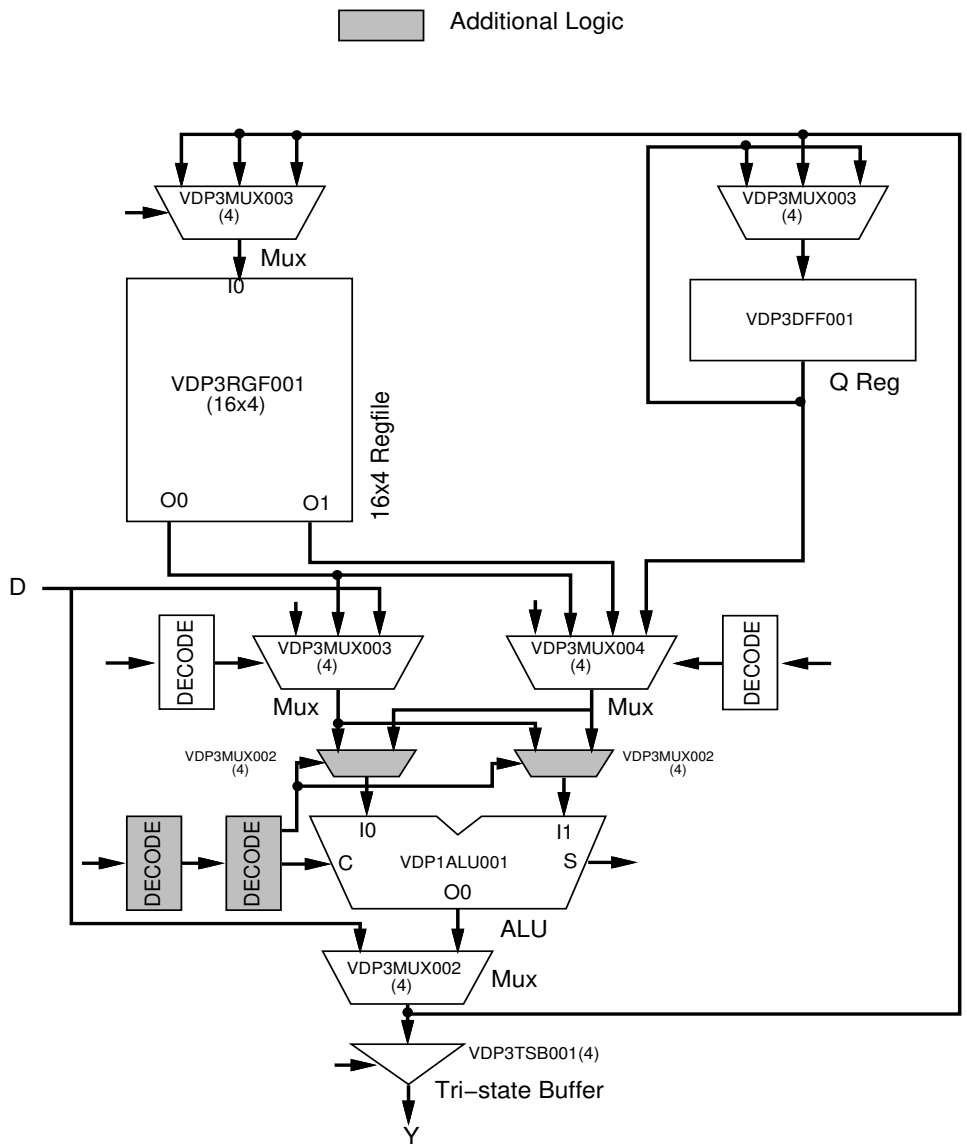


Figure 32: Design of 2901 with VTI components

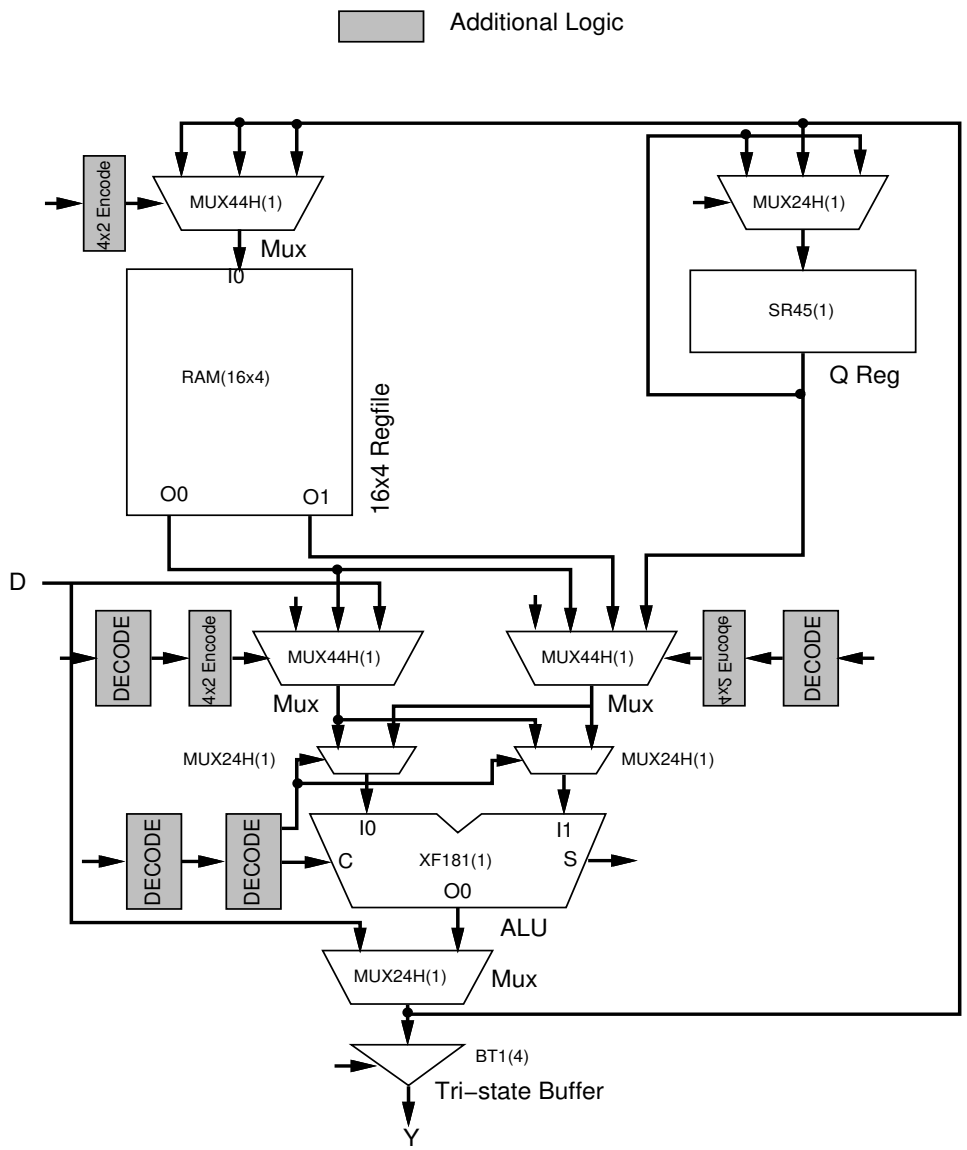


Figure 33: Design of 2901 with Toshiba gate array components

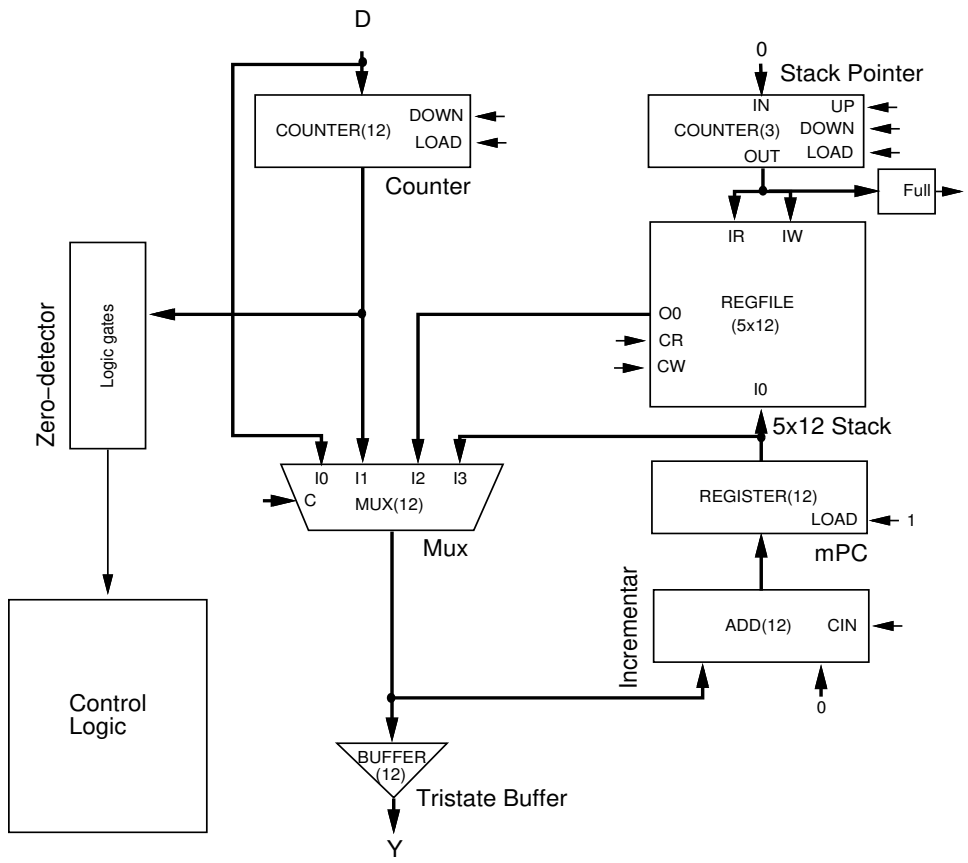


Figure 34: Design of 2910 with GENUS components

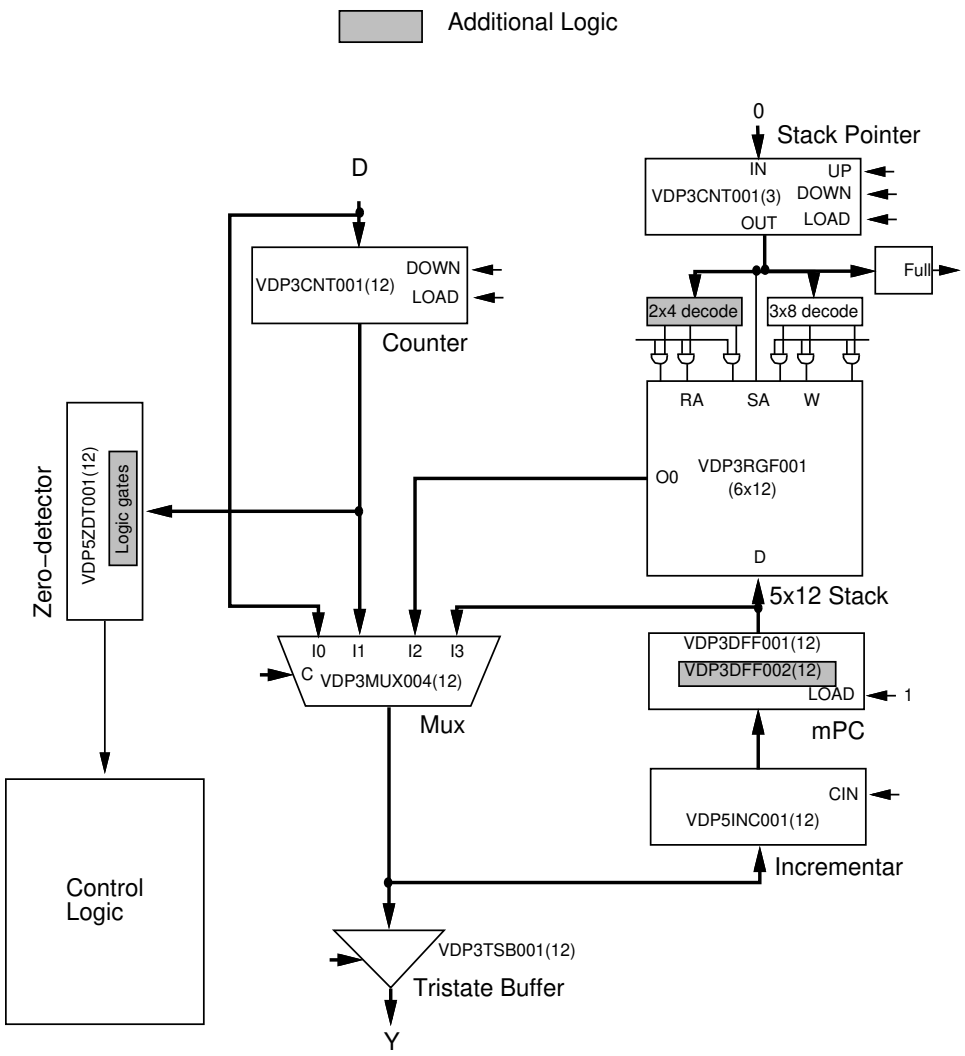


Figure 35: Design of 2910 with VTI components

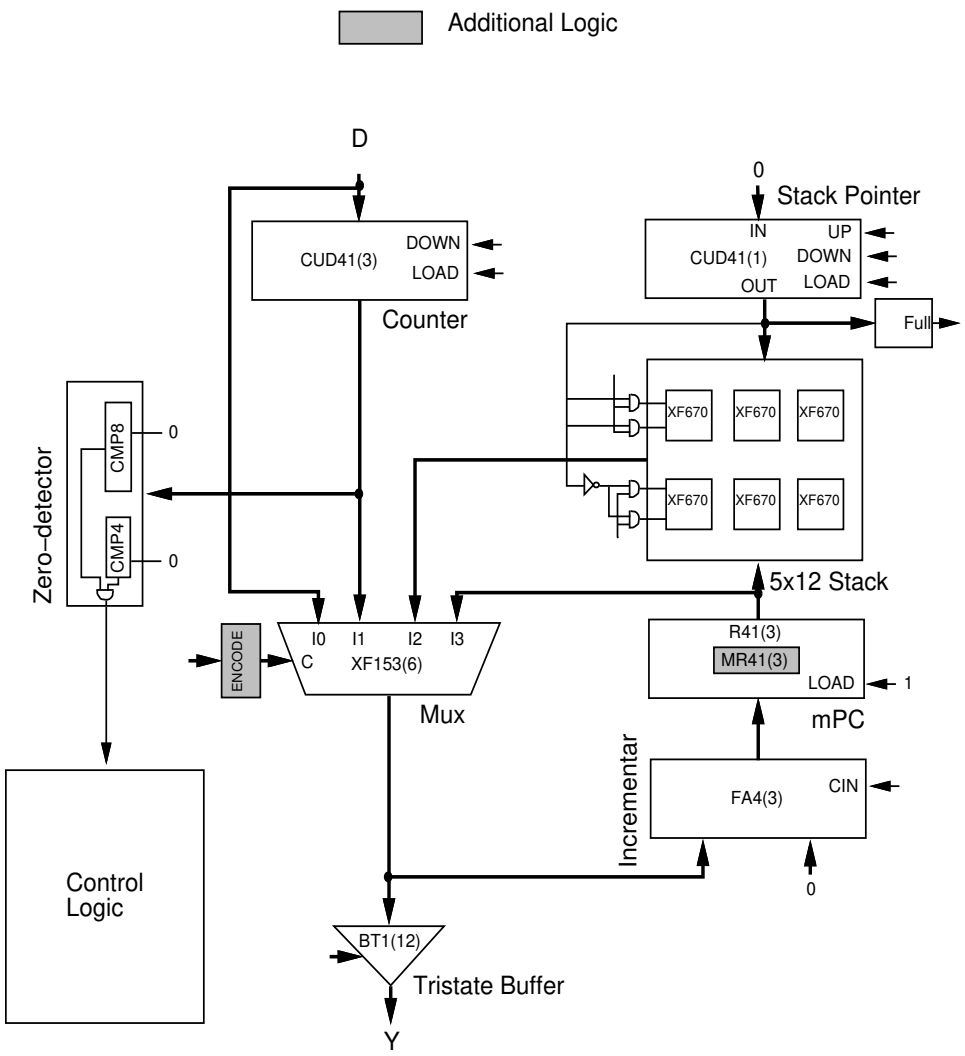


Figure 36: Design of 2910 with Toshiba gate array components

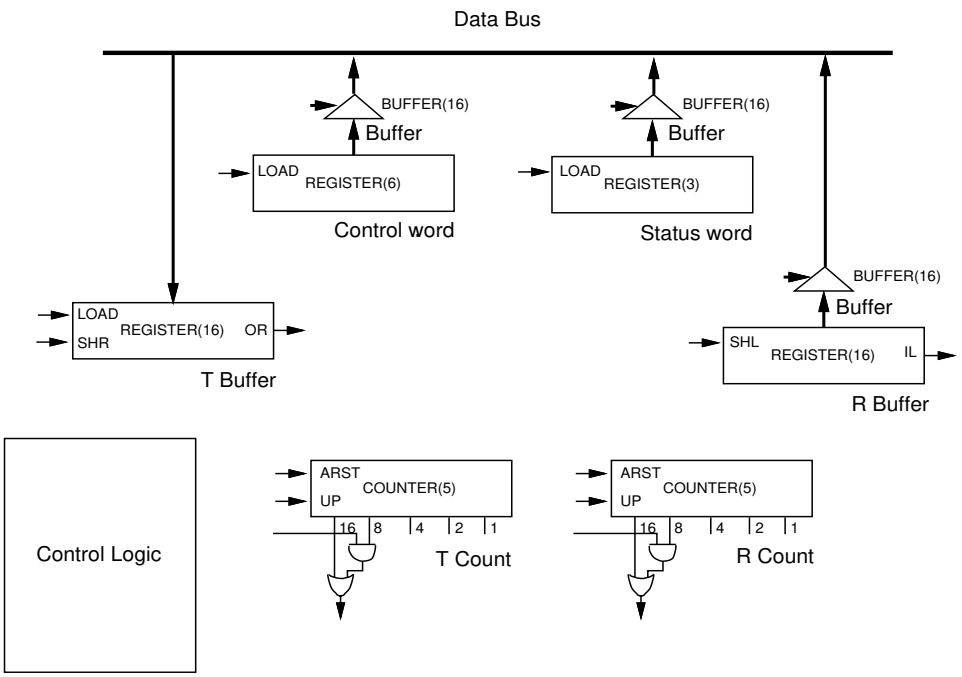


Figure 37: Design of SRT with GENUS components

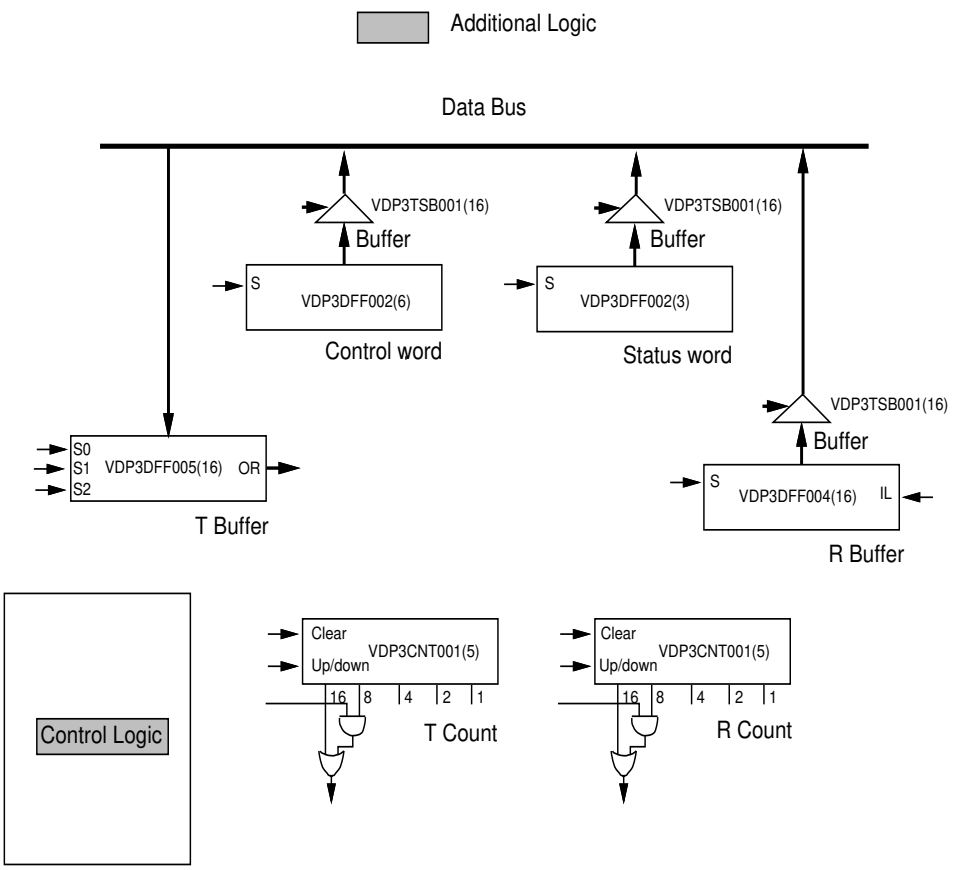


Figure 38: Design of SRT with VTI components

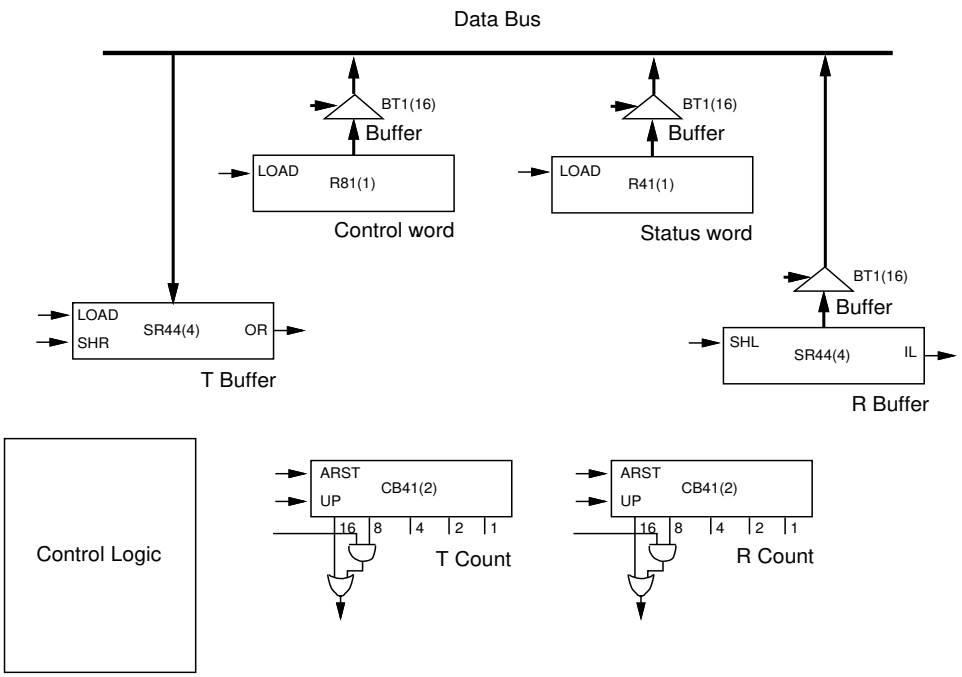


Figure 39: Design of SRT with Toshiba gate array components

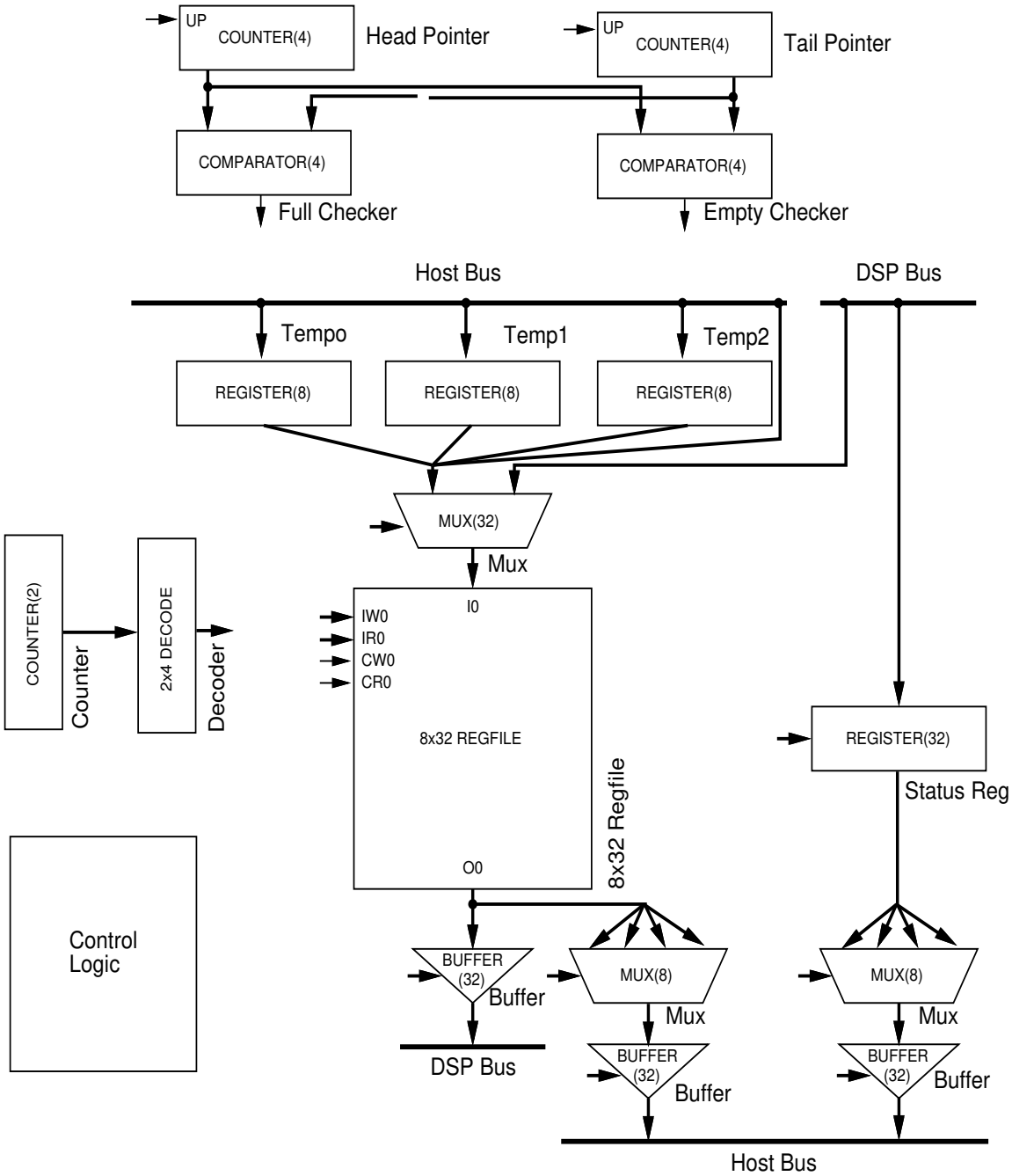


Figure 40: Design of Circular Buffer with GENUS components

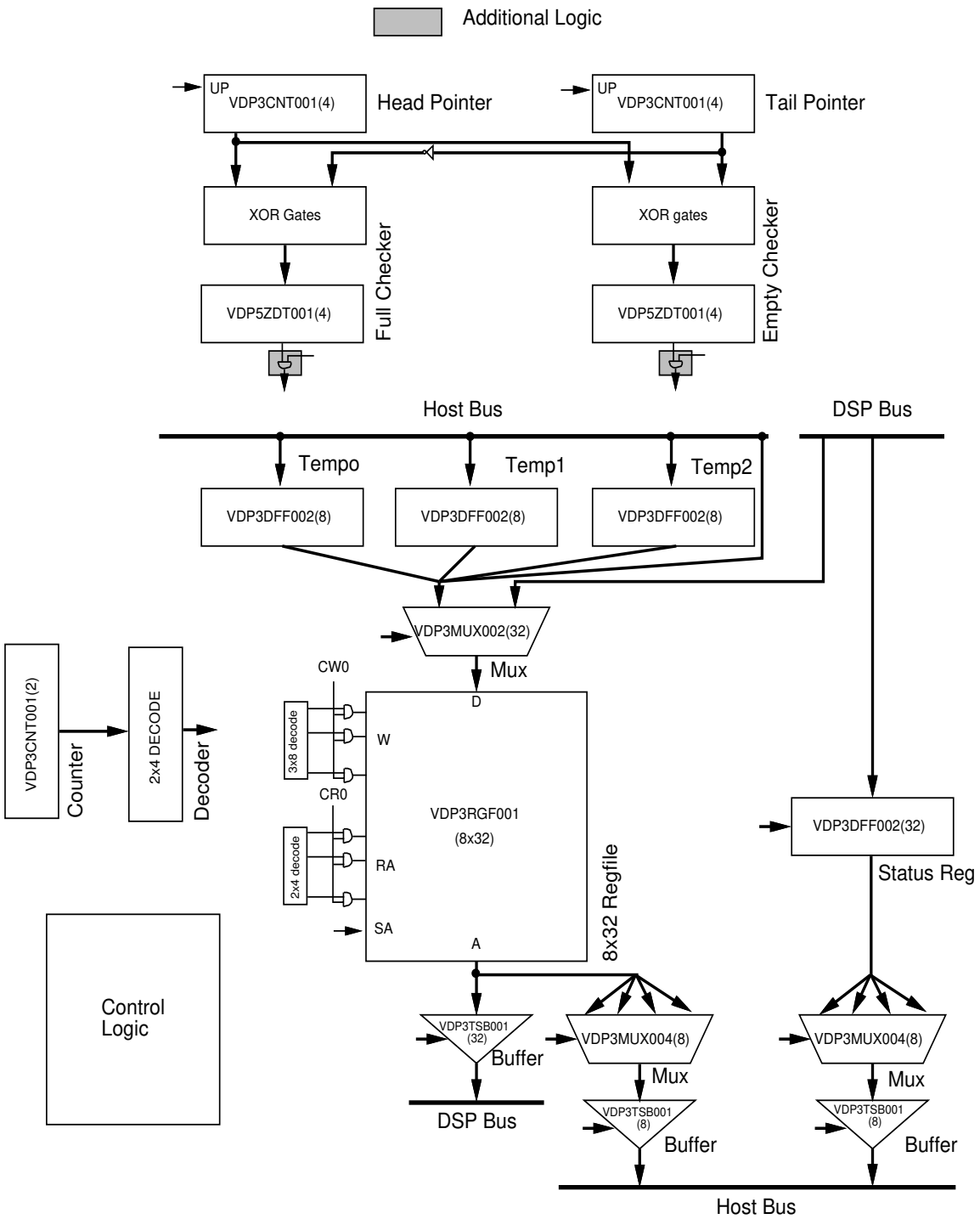


Figure 41: Design of Circular Buffer with VTI components

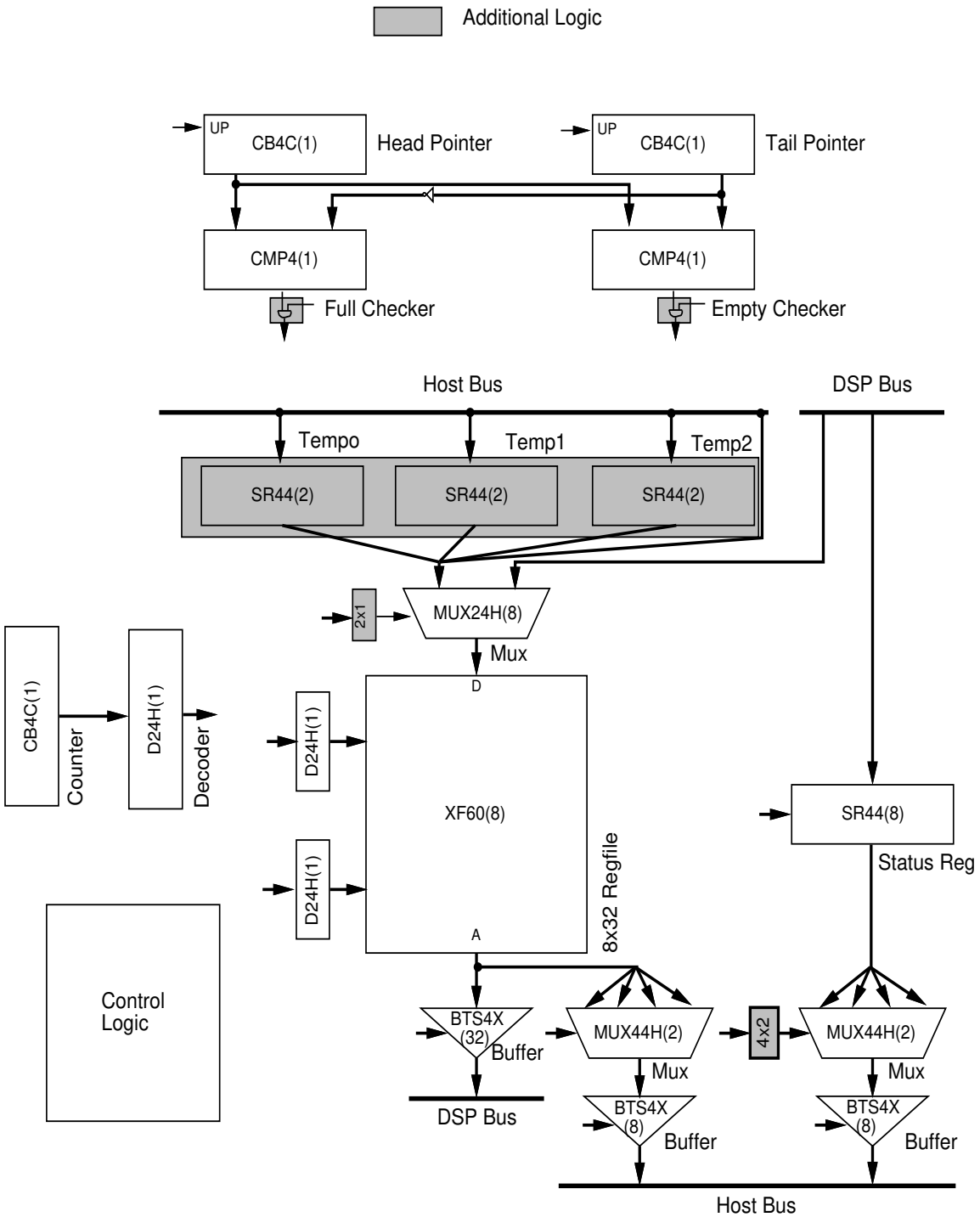


Figure 42: Design of Circular Buffer with Toshiba gate array components