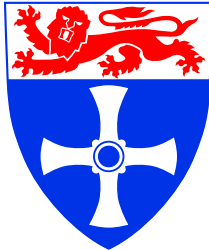

School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE



VERISYN: Tool support for Verilog Asynchronous Synthesis - A Tutorial Guide

F.Burns, D.Shang, A.Koelmans, A.Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2004-104

September 2004

Contact:

f.p.burns@ncl.ac.uk

Delong.Shang@ncl.ac.uk

Albert.Koelmans@ncl.ac.uk

Alex.Yakovlev@ncl.ac.uk

Supported by EPSRC grant GR/R16754

NCL-EECE-MSD-TR-2004-104

Copyright © 2004 University of Newcastle upon Tyne

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

<http://async.org.uk/>

VERISYN: Tool support for Verilog Asynchronous Synthesis - A Tutorial Guide

F.Burns, D.Shang, A.Koelmans, A.Yakovlev

September 2004

Abstract

This report provides details of a software tool VERISYN, a support tool for asynchronous behavioural synthesis. The VERISYN tool operates as a front end as part of a behavioural synthesis system. It takes as input high level behavioural descriptions written in Verilog and parses and schedules them and translates them into an intermediate multi-Petri net format. The multi-Petri net format is comprised of two net types: coloured Petri nets and labelled Petri nets which are subsequently used as an intermediate format for direct asynchronous mapping to both datapath and control (local and global). This tutorial describes the net generation methodology used by VERISYN from a basic set of Verilog behavioural constructs.

Contents

1 Introduction

- 1.1 Background and related work
- 1.2 Overview of toolflow
- 1.3 Basic concepts

2 Tool Interface

- 2.1 Projects
- 2.2 Managing files
- 2.3 Compiling files
- 2.4 Simulation environment
- 2.5 Net generation
- 2.6 Generating hardware output
- 2.7 Handling schedules

3 ICARUS and the Verilog language

- 3.1 Data types
 - 3.1.1 Physical data types
 - 3.1.2 Abstract data types
- 3.2 Binary/Unary Operators
- 3.3 Commands and Control flow
 - 3.1.1 Blocking and Non-blocking Procedural assignments
 - 3.1.2 Selection - if and case statements
 - 3.1.3 Repetition - while, for and repeat statements
 - 3.1.4 Functions and Tasks
- 3.4 Program structure

4 GCD example

- 4.1 Specification
- 4.2 Simulation
- 4.3 Scheduling and net generation
- 4.4 Hardware output

5 Net Constructs

- 5.1 Sequential statement nets
 - 5.1.1 Consecutive assignments

5.1.2 Common input assignments

5.1.3 Repeated consecutive assignments

5.2 While loop nets

5.3 Repeat loop nets

5.4 For loop nets

5.5 Conditional nets

5.5.1 If then else netss

5.5.2 Nested if then else nets

5.5.3 Case nets

5.6 Fork nets

5.7 Function nets

5.8 Task nets

6 CD decoder example

6.1 Specification

6.2 Net generation

7 Pipelining

7.1 Introduction

7.3 Pipeline example

8 AES example

8.1 Specification

8.2 Net generation

9 References

1 Introduction

1.1 Background and related work

Compared to synchronous CAD tools which are very mature and accepted by industry, there is still a shortage of mature CAD tools to support asynchronous circuit designs [1]. There are only a few tools available which generate complete datapath and control asynchronous solutions from high-level descriptions. Examples of such tools are Tangram [2] and Balsa [3]. These tools use specification languages based on handshaking and generate completely asynchronous implementations.

One of the reasons for the lack of commercial tools is the level of complexity of generating complex asynchronous control circuits. Synchronous tools produce 'clocked' designs whereas asynchronous produce 'self-timed' ones. The gap between these two has so far meant a lack of interest in incorporating asynchronous methodologies into synchronous design domains or in using standard specification languages as inputs for asynchronous tools.

Attempts at using commercial synchronous languages as specification inputs for asynchronous tools has so far been largely targeted towards the control domain. In [4] the Verilog language has been used to specify micropipelined designs out of which controllers are generated. This technique, however, suffers from the state explosion problem when designs become larger because they refine to signal transition graphs (STGs) and subsequently use logic synthesis in Petrify [5]. In [6] the authors overcome this problem using Verilog descriptions which are partitioned to generate chained FSM or micro-engine controllers but the circuits are not speed independent.

As in previous approaches our approach avoids the state explosion problem by using a synthesis technique which combines the advantages of using a direct mapping method [7] but using Verilog as a specification language. Ultimately we wish to be able to generate guaranteed speed independent asynchronous circuits for designs which compete with those generated from mature asynchronous tools such as Balsa. Our aim is to explore the design space more like [8] using high-level synthesis, but making use of a unique direct mapping approach from an intermediate Petri net format which enables the direct mapping to asynchronous hardware [9, 10]. This document describes a prototype toolset version V1.0.0 which provides the net generation support from a high level language (Verilog) for the direct mapping of asynchronous datapaths and controllers.

In [8] VHDL constructs are translated into Petri net representations which are used as an intermediate format for subsequent translation into synchronous systems. Our approach which is targeted towards direct mapping requires net generation of a different kind. The nets that we generate must support direct asynchronous mapping for both datapath and control (local and global). The net generation supported by our design flow is described in the following subsections.

1.2 Overview of toolflow

Fig. 1 shows the generalized toolflow for our asynchronous synthesis technique.

The synthesis approach we use inputs a high-level Verilog specification, optimizes and schedules it, and then translates it into a unique intermediate Petri net representation. This is subsequently synthesized into datapath circuits and control circuits using a combination of direct mapping techniques. The intermediate format outlined in this document uses a combination of datapath nets based on Coloured Petri Nets (CPNs) [11] and control nets based on Labelled Petri nets (LPNs) [12]. The control nets are split into two types for

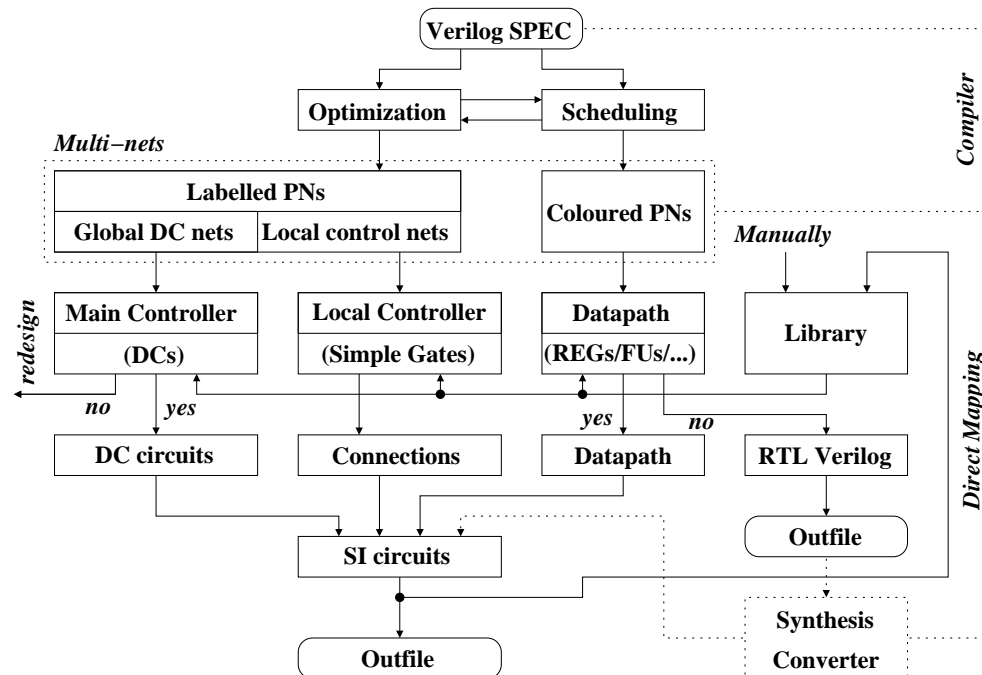


Figure 1: Diagram of synthesis flow

mapping: (i) global control nets which are used for direct mapping to David Cells (DCs) [14, 15] and (ii) local control nets for mapping to simple control gates. Subsequently, after the nets are generated, hardware components are selected from a basic library for mapping. An RTL Verilog description can also be output to a synthesis converter: i.e. a synchronous synthesis tool (e.g. Synopsys Design Compiler) generates circuits which are converted back to asynchronous circuits by another in-house tool Verimap [16]. After mapping optimized speed independent circuits are generated.

1.3 Basic concepts

Initially, the ICARUS Verilog compiler [17] is used for compiling the Verilog behavioural specifications. The parsed output generated from the backend interface of the compiler is used for generating the intermediate format used by the front end scheduling tool. The intermediate format is first passed to an optimizer and scheduler. For scheduling, sequential assignments are scheduled in their natural order allowed by their dependencies and consecutive independent assignments are scheduled in parallel.

Our tool handles a basic set of Verilog constructs. Verilog constructs handled by the scheduler include sequential = and parallel => assignments, conditional statements, i.e. if..else and case, and fork and join statements. A variety of looping constructs are also handled including cyclic behaviour, i.e. always, and repetitive behaviour, i.e. while, repeat and for loops. In addition higher level constructs can be handled such as functions and tasks, (procedures).

The Petri-net synthesizer takes the control output from the scheduler and uses this information to generate the intermediate Petri-net description. The Petri-net description uses a multi-net format comprising three basic parts: a datapath net (coloured net), a global control net (labelled net) and local net (labelled net). These are output as three different information streams, one containing information on the datapath

and internal local control flow, another containing information about the global control nets and another about the local control flow nets which flow between the datapath and global control nets.

The Petri-nets generated from the net generation tool have differing types. The data-path net which is subsequently mapped to the datapath is based on Coloured nets [11]. These are a higher level type of net which use token types as opposed to ordinary Petri-nets which use a single token type. This means that higher level constructs which are required for data-path representation can be represented. Control nets are represented using ordinary labelled Petri-nets.

Once the datapath nets and control nets have been constructed, a syntax-based approach can be used to map them directly into circuits. The CPNs are directly mapped into datapath circuits and the LPNs are directly mapped into control circuits (DCs).

In [9], an informal method using DCs instead of global clock signals is introduced, which makes an SI control circuit easier to implement. In the synthesis of synchronous circuits, all operations are scheduled in several stages, with each stage being distinguished by using a global clock. In our method, DCs are used instead to activate the scheduled control steps. This provides the possibility to divide all the operations into several stages based on the sequence of operations. The difference is that in each stage, asynchronous circuits need the completion signals of the relevant operations because of the various stage lengths rather than the fundamental mode assumption in synchronous circuits.

A block diagram of the connection of a sequence of DC cells representing the token flow through a linear global control net is illustrated in Fig. 2. One cell is equivalent to one global place and transition in the global control net. Two adjacent DCs interact according to the STG flow sequence at the bottom of the diagram.

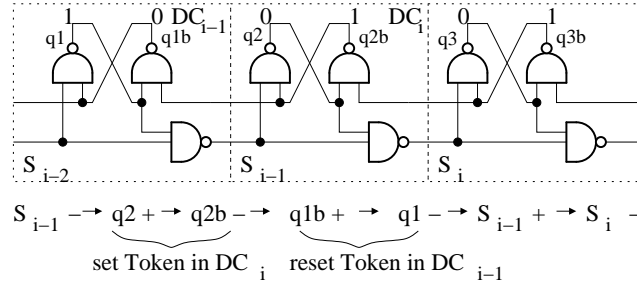


Figure 2: Linear global control DC sequence

DCs can be joined to make more elaborate Verilog control constructs such as fork and join. Other Verilog control constructs such as conditions and loops can be implemented using DCs in a similar way.

We have designed our own basic library of specialized dual-rail components (AMS 0.35um cell library). All operations, specified at the high-level (Verilog behavioural specification), can be implemented using these components. The library consists of all the basic asynchronous components, such as C-elements, Muxes, D-elements, and various kinds of DCs, e.g. a DC cell with an internal OR gate. In addition, some components are being developed that realize all possible datapath operations, such as add, subtract, compare, multiply, divide, multiplex, and so on. We have also developed some basic functional units (FUs) at gate level and/or transistor level such as add and compare.

2 Tool Interface

A tool interface is provided called 'Verisyn' which has been constructed using a C++ linux windows library which is also portable to Windows. Typing the Linux command 'vsyn' invokes the Verisyn tool windows interface. The screen in Fig. 3. shows the main window for the tool. This acts as a front-end for editing, compiling and running Verisyn commands. Menus and command buttons appear at the top. The left hand side contains a browser for displaying files and the right hand side contains an edit and display window.

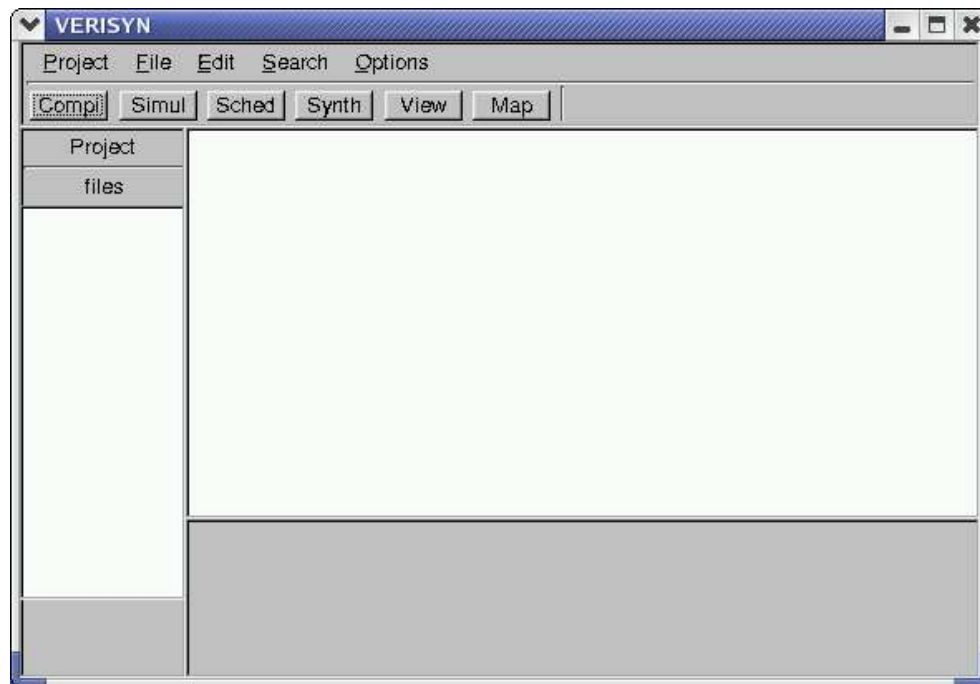


Figure 3: Diagram of tool interface

2.1 Projects

Projects are used for storing Verilog files. An existing project can be opened by selecting "Project => Open" from the menu and this displays a Project selection window. After entering a project name e.g. 'Test' followed by clicking on the OK button the name will be displayed in the Project files box together with a directory listing which displays all the files used in the project.

2.2 Handling files

The directory listing of files is displayed on the left. By clicking on the directory list a Verilog file can be selected e.g. 'seq.v'. When a file has been selected the name will appear at the top right. The file is now editable in the edit window on the right. To make changes to the file the edit window should be used and the file can be saved by selecting "File => Save".

2.3 Compiling files

Verilog files are compiled by clicking on the 'Compile' button at the top. For an example of compiling a Verilog file click on the file seq_.v in the project directory list. This is then compiled by clicking on the compile button. This file contains a deliberate error and shows an error message in the runtime window at the bottom.

2.4 Simulation environment

For simulation a testbench file must be opened. The above example can be simulated by clicking on "Project => Open" and then clicking on the project called 'testb'. The file 'tb_seq_.v' should then be selected. Once it appears in the edit window click on the compile button to compile it. To invoke the simulator the simulate button needs to be selected. This displays the simulation output in the display window at the bottom.

2.5 Net generation

As an example of generating nets we will use the Verilog specification shown in Fig. 4.

```
Module Seqntl (In1,In2,In3,Out);
    input [7:0] In1;
    input [7:0] In2;
    input [7:0] In3;
    output [7:0] Out;
    reg [7:0] X;
    reg [7:0] Y;
    reg [7:0] Z;
    reg [7:0] Out;
    always
    begin
        X = In1;           // (1)
        Y = In2;           // (1)
        Z = In3;           // (1)
        Z = X + Z;         // (2)
        Z = Y + Z;         // (3)
        Out <= Z;          // (4)
    end
endmodule
```

Figure 4: Verilog sequential example

The directory ./test should be opened and the file seqntl.v should be selected. When this is displayed in the edit window click on the compile button to compile it. After seqntl.v has been compiled in order to schedule it click the schedule button must be clicked in the Exe list. A statement list will be displayed (see Fig. 5) showing information linking statements in the code to transition names, schedule steps, fu instances and paths etc. The schedule is displayed for each statement in the code under the column sched. For example, the first statement, x is assigned transition name Xa and is scheduled in step 1.

The synth button must be clicked on to synthesize the nets. When synthesis is complete some net generation completion messages will appear in the bottom window. After the nets have been generated the view button should be clicked on and an option window will appear. This provides options for viewing

```
//Schedule output
lhs      par1      fn      par2      trans      sched      fu      path
X        In1              Xa        1        0        0
Y        In2              Ya        1        0        0
Z        In3              Za1       1        0        0
Z        X          +      Z        Za2       2        1        0
Z        Y          +      Z        Za2       3        1        0
Out      Z                      Outa      4        0        0
```

Figure 5: Schedule output

textual and diagrammatic information relating to the nets. To view the data net output select the text and data options in the option window. After clicking on the OK button a file will be displayed in the edit window containing net information about the datapath. The textual format will be similar to that shown in Fig. 6.

```
//data net output
trans  place  i/o    d/c    I/O    ini    fn    cncr
m_xa   m_in1   i      d      I      ini    fn    cncr
m_xa   m_cla   i      c      I      ini    fn    cncr
m_xa   m_dla   o      c      I      ini    fn    cncr
m_xa   m_x     o      d      V      ini    fn    cncr
m_ya   m_in2   i      d      I      ini    fn    cncr
m_ya   m_clb   i      c      I      ini    fn    cncr
m_ya   m_dlb   o      c      I      ini    fn    cncr
m_ya   m_y     o      d      V      ini    fn    cncr
```

Figure 6: Example of data net output

The textual data net output contains the names of transitions on the left with place names appearing to the right together with input and output types. d/c stands for data/control. The ini column is used for initialization of places and fn is used to specify whether the transition is a function or not. The Petri-net diagram for the seqpar example for the datapath is shown in Fig. 7.

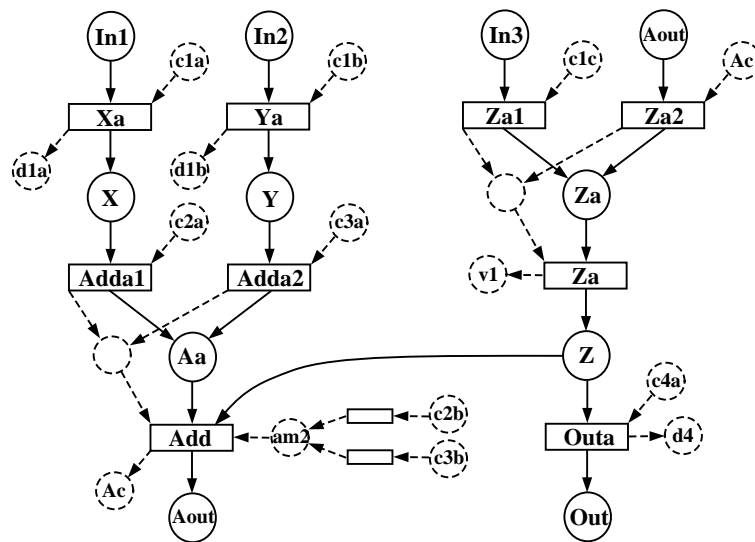


Figure 7: Seqntl data net

In the diagram transitions are used for representing functional units [10]. Multiplexor nets (centre left) are used for multiplexing inputs 'X' and 'Y' to the left hand side of the adder 'Add'. Local control net connections, e.g. signals 'c2a' and 'c3a', are shown flowing into and out of the data net.

To view the local net output select the text and data options in the option window. After clicking on the OK button a file will be displayed in the edit window containing the local net information. The textual format will be similar to that shown in Fig. 8. This information describes the control signals for activating specific parts of the datapath. For example, transition 'c1' fires control signals (tokens) to places 'c1a', 'c1b' and 'c1c'.

```
//local net output
trans  place  i/o    d/c    ini
m_c1   m_c1    i      c      ini
m_c1   m_c1a   o      c
m_c1   m_c1b   o      c
m_c1   m_c1c   o      c
m_c1   m_c1d   o      c
m_v1   m_c1d   i      c
m_v1   m_v1    i      c
m_v1   m_d1d   o      c
m_d1   m_d1a   i      c
m_d1   m_d1b   i      c
m_d1   m_d1d   i      c
m_d1   m_d1    o      c
```

Figure 8: Example of local net output

The diagram showing the local nets generated for the example is shown in Fig. 9.

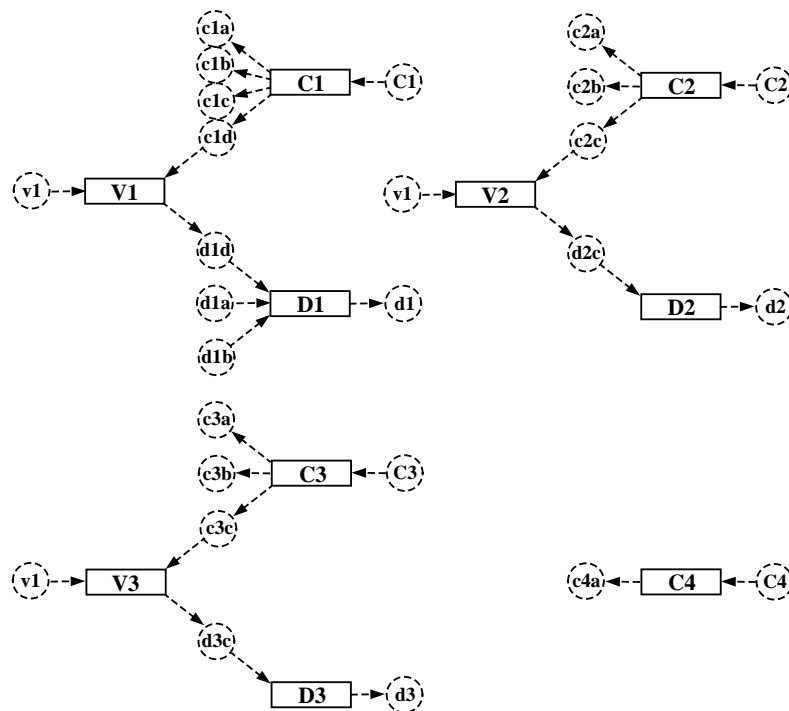


Figure 9: Seqntl local net

Clicking on the Global net button will display information about the global DCnets. Click on View Global net to show this in the text window. These can be viewed by selecting 'diagram' and 'global' in the view options window. The nets are shown in Fig. 10.

2.6 Generating hardware output

To generate hardware from the nets a map button is provided. This provides a link to map generation software which maps the intermediate format into hardware descriptions.

2.7 Handling schedules

To show how to generate different schedules first modify the specification to that shown in Fig. 11.

Now click on the compile button followed by the schedule button. The schedule in Fig. 12 will be displayed in the edit window.

To change the schedule so that it executes in parallel click on the Options menu and select the Schedule option which will bring up a dialog window. Now fill in the box next to the Add label so that it contains the digit 2. Now close the dialog window and invoke the scheduler. Study the resulting schedule. Note now that the schedule now executes the addition statements in parallel. Now click on the synth button. The datapath net file contains a description of the datapath net shown in Fig. 13. Note that two addition nets are generated for executing the additions in parallel.

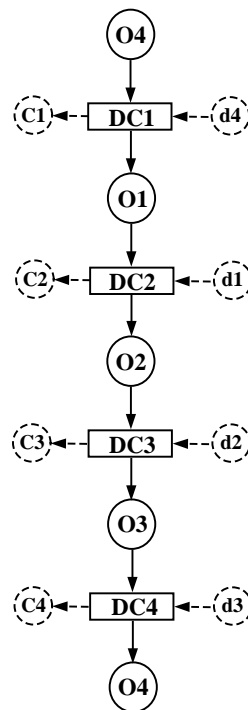


Figure 10: Seqntl global net

```

Module Sched (In1,In2,In3,Out);
  input [7:0] In1;
  input [7:0] In2;
  input [7:0] In3;
  output [7:0] Out;
  reg [7:0] X;
  reg [7:0] Y;
  reg [7:0] Z;
  reg [7:0] Out;
  always
  begin
    X = In1;
    Y = In2;
    Z = In3;
    X = X + 1;
    Y = Y + 2;
    Z = X + 3;
    Out = Z;
  end
endmodule

```

Figure 11: Verilog schedule example

Nets for the local nets are shown in Fig. 14.

A diagram of the DCnet is shown in Fig. 15.

```
//Schedule output
```

lhs	par1	fn	par2	trans	sched	fu	path	fork
X	In1			Xa1	1	0	0	0
Y	In2			Ya1	1	0	0	0
Z	In3			Za1	1	0	0	0
X	X	+	1	Xa2	2	1	0	0
Y	Y	+	2	Ya2	3	1	0	0
Z	X	+	3	Za2	4	1	0	0
Out	Z			Outa	5	0	0	0

Figure 12: Schedule output

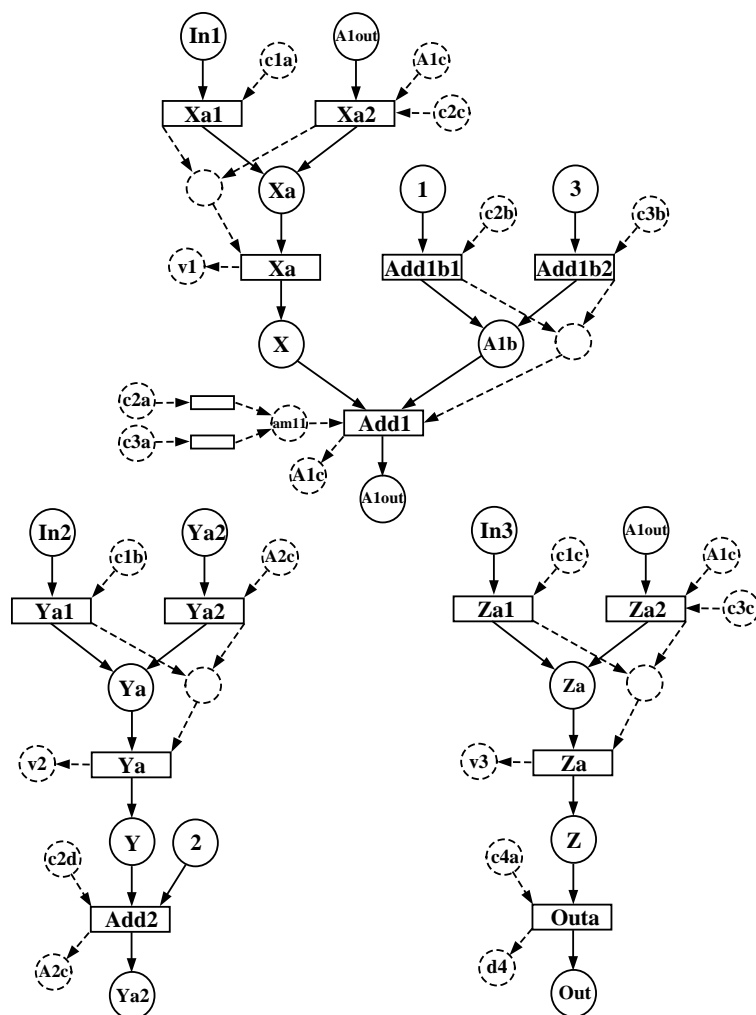


Figure 13: Par data net

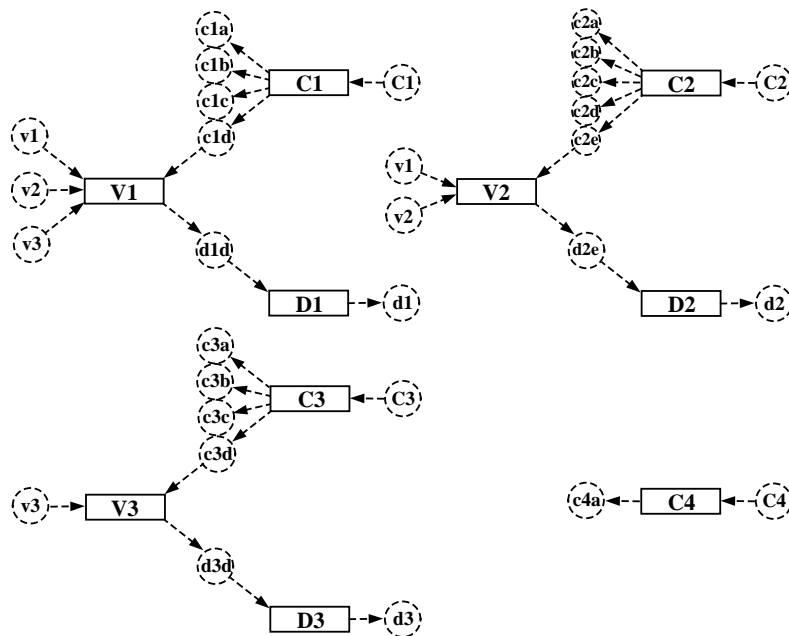


Figure 14: Par local net

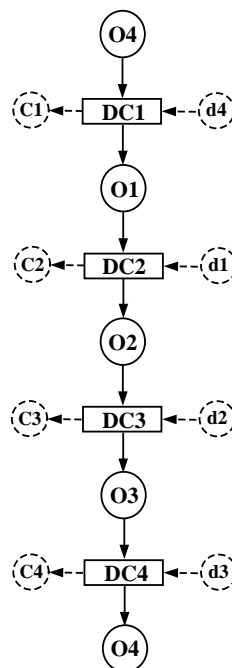


Figure 15: Par global net

3 ICARUS and the Verilog language

Verilog HDL is a Hardware Description Language (HDL). We use the ICARUS Verilog compiler [17] for compiling Verilog specifications. Icarus Verilog is a Verilog simulation and synthesis tool. It operates as a

compiler, compiling source code written in Verilog (IEEE-1364) into various target formats. For simulation, the compiler can generate an intermediate form called vvp assembly. For synthesis, the compiler generates netlists in the desired format.

The compiler is intended to parse and elaborate design descriptions written to the IEEE standard 'IEEE Std 1364-2001'. This is a fairly large and complex standard covering both low level and high level and as we are only interested in a subset of the language we present those constructs only of interest. This handout focuses on only the portions of Verilog which support the higher RTL or behavioural level. This level allows hardware designers to express their design with behavioral constructs, deferring the details of implementation to a later stage of design.

The main porting target is Linux, although it works well on many similar operating systems. The current release is available in source and a variety of binary forms in the FTP directory

[<ftp://icarus.com/pub/eda/verilog/v0.7/>](ftp://icarus.com/pub/eda/verilog/v0.7/).

However, we support our own version as our requirements are specifically tailored to net generation of a particular type.

3.1 Data types

At the higher level Verilog caters for the following data types: physical data types and abstract data types.

3.1.1 Physical data types

In Verilog the primary data types are for modeling registers (`reg`). The `reg` variables store the last value that was procedurally assigned to them.

The `reg` data object may have the following possible values:

0 logical zero or false, 1 logical one or true or x (unknown) logical value.

`reg` variables are initialized to x at the start of a simulation.

You may specify the size of a register in a declaration. For example, the declaration

```
reg [0:7] A, B;
```

specifies registers A and B to be 8-bits wide with the most significant bit the zeroth bit.

The bits in a register can be referenced by the notation [`<start-bit>`:`<end-bit>`].

The range referencing in an expression must have constant expression indices. However, a single bit may be referenced by a variable. For example:

```
reg [0:7] A, B;
```

```
B = 3;
```

```
A[B] = 1'b1;
```

Memories are specified as vectors of registers. For example, in the following, Mem is 1K words each 32-bits.

```
reg [31:0] Mem [0:1023];
```

The notation `Mem[0]` references the first word of memory. The array index for memory (register vector) may be a register.

3.1.2 Abstract data types

In addition to modeling registers, there are other uses for variables in a hardware model. For example, the designer might want to use an integer variable to count the number of times an event occurs. For

the convenience of the designer, Verilog HDL has several data types which do not have a corresponding hardware realization. These data types include integer, real and time. A reg variable is unsigned and an integer variable is a signed 32-bit integer e.g.

```
integer Count; // simple signed 32-bit integer
```

Arrays of integer and time variables (but not reals) are allowed. Multiple dimensional arrays are not allowed in Verilog HDL. For example:

```
integer K[1:64]; // an array of 64 integers
```

3.2 Binary/Unary Operators

The following table, Table 1, shows different types of operators used in Verilog. These consist of arithmetic, relational and logical operators.

Table 1: Table of Binary/Unary Operators.

Symbol	Operation	Operands	Function
+, -	add, subtract	reg, integer, real	binary arithmetic
*, /, %	times, divide, mod	reg, integer, real	binary arithmetic
=, !=	equal, not equal	all	logical
<, >, <=, >=	inequalities	numeric	logical
!, &&,	logical	logical	logical
~, &, , ^	bitwise	reg, integer, real	operates on bits
~&, ~ , ~^	~bitwise	reg, integer, real	operates on bits
~v, &v, v	unary reduction	reg, integer, real	single bit output
~&v, ~ v	~unary reduction	reg, integer, real	single bit output

Binary arithmetic operators operate on two operands. If any bit of an operand is unknown ('x') then the result is unknown.

Relational operators compare two operands and return a logical value, i.e., TRUE(1) or FALSE(0). If any bit is unknown, the relation is ambiguous and the result is unknown.

Logical operators operate on logical operands and return a logical value, i.e., TRUE(1) or FALSE(0). Do not confuse logical operators with the bitwise Boolean operators. For example, ! is a logical NOT and ~ is a bitwise NOT. The first negates, e. g., !(5 == 6) is TRUE. The second complements the bits, e. g., ~{1, 0, 1, 1} is 0100.

Bitwise operators operate on the bits of the operand or operands. For example, the result of A & B is the AND of each corresponding bit of A with B.

Unary reduction operators produce a single bit result from applying the operator to all of the bits of the operand. For example, &A will AND all the bits of A.

The conditional operator operates much like in the language C.

3.3 Commands and control flow

Verilog HDL has a rich collection of command and control statements which can be used in the procedural sections of code, i. e., within an initial or always block. Verilog HDL uses begin and end instead of { } brackets like C. The following subsections typically show only an example of each construct.

3.3.1 Blocking Procedural Assignments

The Verilog language has two forms of the procedural assignment statement: blocking and non-blocking. The two are distinguished by the = and <= assignment operators. The blocking assignment statement (= operator), which we use predominantly in our descriptions, acts much like in traditional programming languages. The whole statement is done before control passes on to the next statement. The non-blocking (<= operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides after it. For example, the following Verilog program

```
initial
begin: init
    X = 3;
    Y = X+1;
    X <= X+1;
    Z = X+1;
end
```

makes the following assignments: X = 3, Y = 4, X = 4, Z = 4.

The effect is for all the non-blocking assignments to use the old values of the variables at the beginning of the current simulation time unit and to assign the registers new values at the end of the current simulation time unit.

3.3.2 Selection - if and case Statements

The if statement is straightforward to use and takes the branch corresponding to the condition e.g.

```
if(A==4)
begin
    B = 2;
end
else
begin
    B = 4;
end
```

For a case statement the first value that matches the value of the expression is selected and the associated statement is executed then control is transferred to after the endcase e.g.

```
case(A==0)
1:  Z = 2;
2:  Z = 3;
end
```

3.3.3 Repetition - while, for and repeat statements

The for statement is very close to C's for statement except that the ++ and -- operators do not exist in Verilog. Therefore, we need to use $i = i + 1$ e.g.

```
for(I=0;I<100;I=I+1)
  begin
    $display("i= %0d", i);
  end
```

The while statement acts in the normal fashion e.g.

```
while(i<10)
  begin
    $display("i= %0d", i);
  end
```

The repeat statement repeats the following block a fixed number of times, in this example, five times.

```
repeat(5)
  begin
    i = i+1;
    $display("i= %0d", i);
  end
```

3.3.4 Functions and Tasks

The purpose of a function is to return a value that is to be used in an expression. A function definition must contain at least one input argument. The passing of arguments in functions makes use of input argument ports only. The definition of a function is the following:

```
function <range or type> <function name>;
  <argument ports>
  <declarations>
  <statements>
endfunction
```

where <range or type> is the type of the results passed back to the expression where the function was called. Inside the function, one must assign the function name a value. Below is a simple example of a function.

```
function [7:0] Add;
  input [7:0] X;
  input [7:0] Y;
  begin
    Add = X+Y;
  end
endfunction
```

An example of an assignment making use of the Add function is given below:

```
X = Add(4,7);
```

Tasks are like procedures in other programming languages, e.g., tasks may have zero or more arguments and do not return a value. Functions act like function subprograms in other languages. A task, however, may contain time controlled statements.

The definition of a task is the following:

```
task <task name>;  
<argument ports>  
<declarations>  
<statements>  
endtask
```

Port arguments in the definition may be input, inout or output. Below is a simple example of a task.

```
Task Add;  
    input [7:0] A;  
    input [7:0] B;  
    output [7:0] Y;  
begin  
    Y = A+B;  
end  
endtask
```

An invocation of a task is of the following form:

```
<name of task> (<port list>);
```

```
e.g. Add(4,7,Y);
```

where <port list> is a list of expressions which correspond by position to the <argument ports> of the definition. Input and inout parameters are passed by value to the task and output and inout parameters are passed back to invocation by value on return. Call by reference is not available.

3.4 Program structure

Usually we place one module per file. Modules may run concurrently, but usually there is one top level module which specifies a closed system containing both test data and the behavioural model.

Modules are specified behaviorally. A behavioral specification defines the behavior of a digital system (module) using traditional programming language constructs, e.g., ifs, assignment statements, etc., as opposed to a structural specification which expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules.

The structure of a module is the following:

```
module <module name> (<port list>);  
<declares>  
<module items>  
endmodule
```

The <module name> is an identifier that uniquely names the module. The <port list> is a list of input, inout and output ports which are used to connect to other modules. The <declares> section specifies data objects as registers and memories as well as procedural constructs such as functions and tasks.

The <module items> may be initial constructs or always constructs. Procedural assignment may appear inside initial and always constructs. A simple example of a Module specification is given below.

```
Module Model (A,Y);  
    input [7:0] A;  
    output [7:0] Y;  
    reg [7:0] X;  
    reg [7:0] Y;  
    always  
    begin  
        X = A + 1;  
        Y = X + 1;  
    end  
endmodule
```

The module repeatedly calls two assignment statements inside an always construct.

ICARUS also contains a built-in simulator. For simulation purposes the following can be used for testing data.

The \$display system task allows the designer to print a message much like printf does in the language C.

A typical Verilog test program consists of a combination of constructs, i.e., an initial construct to specify the length of the simulation, another initial construct to initialize registers and specify which registers to monitor and an always construct for the system you are modeling.

The statements in the block of the first initial construct will be executed sequentially, some of which appear to be delayed by #1, i. e., one unit of simulated time. The always construct behaves the same as the initial construct except that it loops forever (until the simulation stops).

4 GCD example

4.1 Specification

We now demonstrate the synthesis process using a Verilog example. The Verilog specification for the GCD example is shown in Fig. 16.

```

Module gcd (In1,In2,O);
    input [7:0] In1;
    input [7:0] In2;
    output [7:0] O;
    reg [7:0] X;
    reg [7:0] Y;
    reg [7:0] O;
    always
    begin
        X = In1;
        Y = In2;
        while(X!=Y)
        begin
            if(X<Y)
            begin
                Y = Y-X;
            end
            else
            begin
                X = X-Y;
            end
        end
        O <= X;
    end
endmodule

```

Figure 16: Verilog GCD example

4.2 Simulation

To simulate the GCD example a testbench file for simulation can be found in ./testb called tb_GCD_v.

4.3 Scheduling and net generation

After compilation and scheduling the scheduler generates the schedule output for the GCD which is shown in Fig. 17.

//Schedule output								
lhs	par1	fn	par2	trans	sched	fu	path	loop
X	In1			Xa1	1	0	0	0
Y	In2			Ya1	1	0	0	0
CMP	X	!=	Y	CMP	2	0	0	1
CMP	X	<	Y	CMP1	3	0	0	1
Y	Y	-	X	Ya2	4	1	1	1
X	X	-	Y	Xa2	4	1	2	1
Out	X			Oa	3	0	0	0

Figure 17: GCD schedule output

The Petri-net synthesizer takes the schedule output from the scheduler and uses this information to generate the intermediate Petri-net description. The CPN generated for the GCD example is shown in Fig. 18.

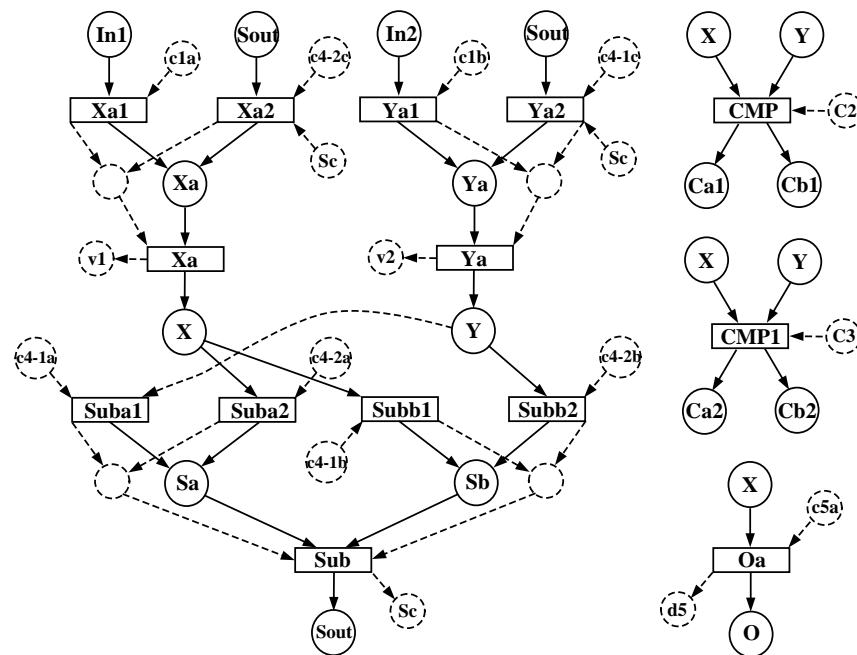


Figure 18: GCD data net

In Fig. 18 the comparator for the while loop is shown top-right. Multiplexing nets appear for X and Y at the top left. The subtractor also uses multiplexor nets to multiplex either X or Y to the left hand side input and X or Y to the right hand side input. The control signals for the paths generated by the second comparator take the form 'c4-1a' and 'c4-2b'. The LPNs generated for the GCD example are shown in Fig. 19. The path signals 'c4-1' and 'c4-2' are returned using signals 'd4-1' and 'd4-2'.

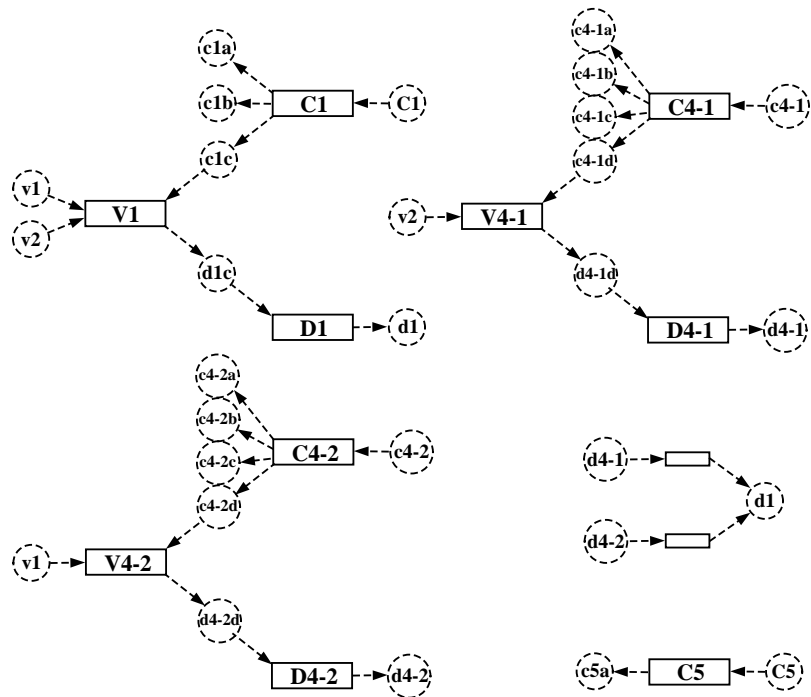


Figure 19: GCD local nets

The DC net generated for the GCD example is shown in Fig. 20.

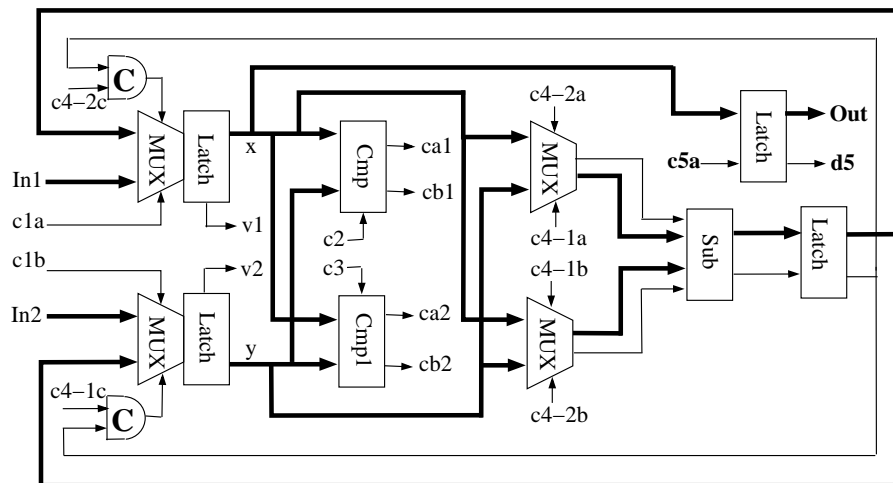


Figure 21: GCD datapath

For the example, the direct mapped output for the GCD controller is shown in Fig. 22. In this case after optimization the controller consists of 6 DCs (main controller), several buffers and several C-elements (local control).

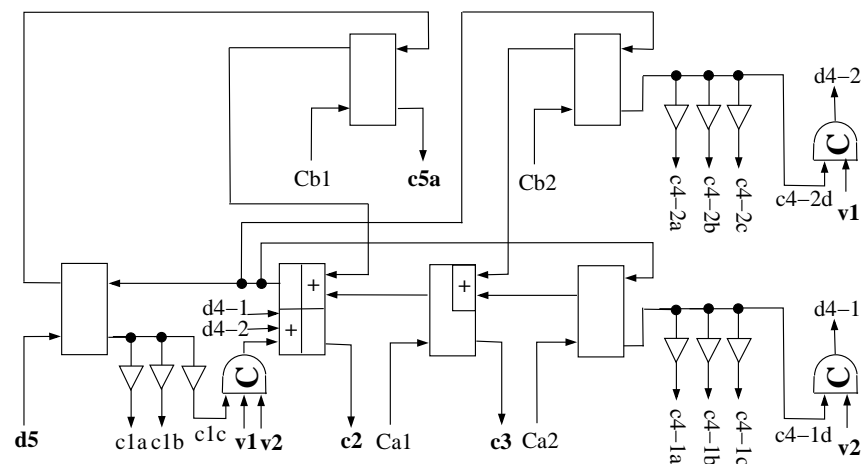


Figure 22: GCD controller

5 Net constructs

Verilog covers a wide range of behavioural constructs. The main Verilog constructs which the net generation tool covers are described in the following subsections.

5.1 Sequential statement nets

This subsection focusses on the net construction for different types of Verilog sequential statements.

5.1.1 Consecutive assignments

Here we present an example which uses consecutive statement assignments. The Verilog code for the example is shown in Fig. 23.

```
Module Seq1 (A,Z);
  input [7:0] A;
  output [7:0] Z;
  reg [7:0] X;
  reg [7:0] Y;
  reg [7:0] Z;
  always
  begin
    X = A + 6;      // (1)
    Y = X + 2;      // (2)
    Z = Y + 1;      // (3)
  end
endmodule
```

Figure 23: Verilog consecutive statement example 1

The module declares one input, three variables and one variable output. Assignments are made in consecutive order to the three variables X, Y and Z.

The data net generated for the example takes the form shown in Fig. 24.

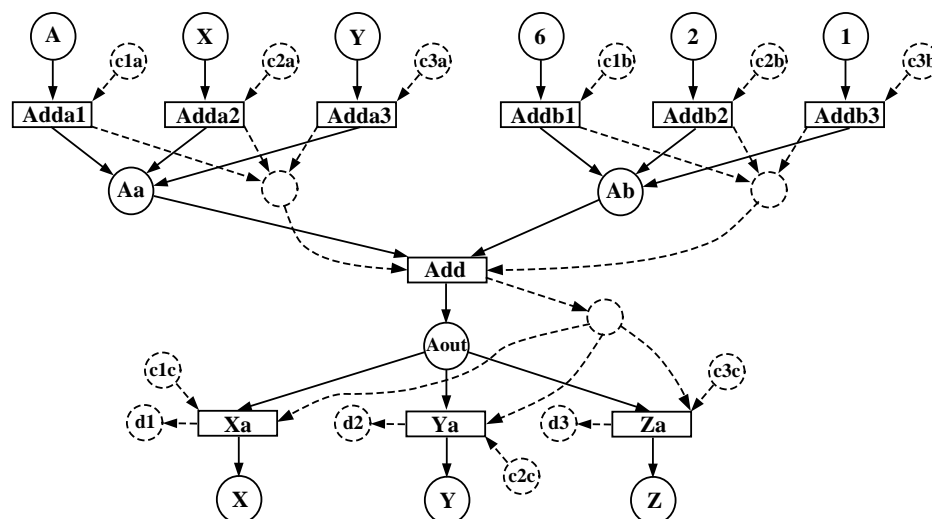


Figure 24: Seq1 data net

At the top of Fig. 24 two three way multiplexor nets are shown. The three left hand ones are shown to the left, 'Adda1', 'Adda2' and 'Adda3', and the three right hand ones are shown to the right, 'Addb1', 'Addb2' and 'Addb3'. The control selection for these are governed by the control selections 'c1a', 'c2a', 'c3a' and 'c1b', 'c2b', 'c3b' respectively. The 'Add' transition in the centre is used for all of the additions and the results from this are passed to place Aout. Subsequently depending on which control selector is selected 'c1c', 'c2c' or 'c3c' the corresponding assignment to X, Y or Z is selected at the bottom. Control

flow is returned from the data net via signals 'd1', 'd2' and 'd3'.

The local control nets for the example are shown in Fig. 25. Three transitions 'C1', 'C2', 'C3' are used to generate the three sets of control signals 'c1a .. c1c'; 'c2a .. c2c' and 'c3a .. c3c' respectively.

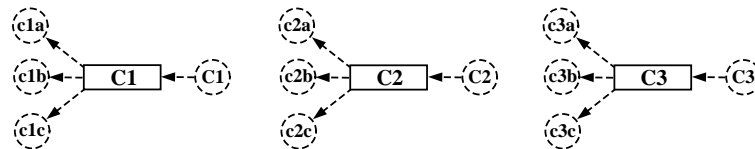


Figure 25: Seq1 local net

The global control net for the Seq1 example is shown in Fig. 26. A sequence of three DC places and transitions are shown in a loop.

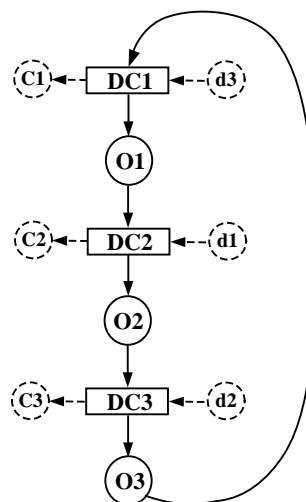


Figure 26: Seq1 global net

5.1.2 Common input assignments

Here we present an example using consecutive statement assignments in which there are shared inputs. The Verilog code for the example is shown in Fig. 27.

Here the module declares one input, three variables and one variable output. Assignments are made in consecutive order to the three variables X, Y and Z but in this example all the assignments share a common right hand side input A.

The data net generated for this example takes the form shown in Fig. 28.

```

Module Seq2 (A,Z);
  input [7:0] A;
  output [7:0] Z;
  reg [7:0] X;
  reg [7:0] Y;
  reg [7:0] Z;
  always
  begin
    X = 6 + A;      // (1)
    Y = X + A;      // (2)
    Z = Y + A;      // (3)
  end
endmodule

```

Figure 27: Verilog consecutive statement example 2

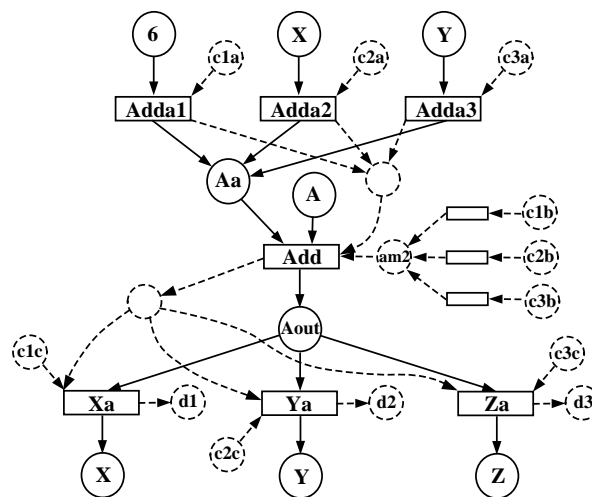


Figure 28: Seq2 data net

At the top of Fig. 28 one three way multiplexor net is used for multiplexing the left hand side inputs. The control selection for these are governed by the control selection signals 'c1a', 'c2a' and 'c3a' respectively. The common right hand side input uses a separate control net to collect control signals 'c1b', 'c2b', 'c3b' into one place. The Add transition in the centre is used for all of the additions and the results from this are passed to place Aout. Subsequently depending on which control selector is selected 'c1c', 'c2c' or 'c3c' the corresponding assignment net is selected at the bottom.

The local control nets for the example are shown in Fig. 29. Three transitions 'C1', 'C2', 'C3' are used to generate the control signals 'c1a .. c1c', 'c2a .. c2c' and 'c3a .. c3c' respectively as in the previous example.

The global control net for the example is shown in Fig. 30. As in the first example three DC places and transitions are used again in a loop.

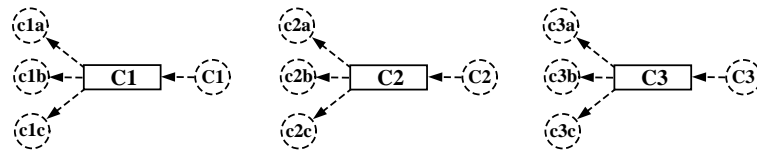


Figure 29: Seq2 local net

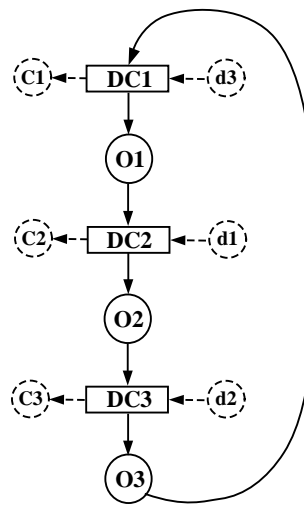


Figure 30: Seq2 global net

5.1.3 Repeated consecutive assignments

Here we present an example which uses another type of consecutive statement assignment. The Verilog code for the example is shown in Fig. 31.

```
Module Seq3 (A,Z);
  input [7:0] A;
  output [7:0] Y;
  reg [7:0] Y;
  always
  begin
    Y = A + 6;      // (1)
    Y = Y - 1;     // (2)
  end
endmodule
```

Figure 31: Repeated consecutive assignments

For this example the module declares one input, one variable and one variable output. Assignments are made in consecutive order but this time the assignments are made to one variable Y and there are no common inputs.

The data net generated for these consecutive assignments takes the form shown in Fig. 32.

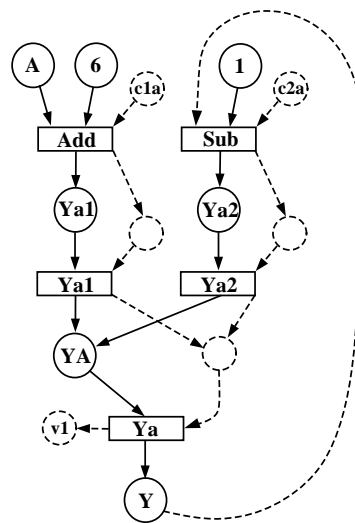


Figure 32: Seq3 data net

In the centre of Fig. 32 one two way multiplexor net is shown for multiplexing the functional outputs to variable Y. The control selection for these are governed by the internal control selection signals this time coming from the functional transitions 'Add' and 'Sub' above. The functional units are activated by the control signals 'c1a' and 'c2a' respectively. The 'Add' and 'Sub' transitions execute the addition and subtraction and the results from these are passed to places 'Ya1' and 'Ya2'. Y is fed back to the Sub net at the top.

The local control nets for the example take the structure shown in Fig. 33. Two control step transitions 'C1', 'C2' are used to generate the initial control signals 'c1a', 'c2a', and two return transitions 'D1', 'D2' are used for generating return signals 'd1' and 'd2' respectively. In this case an additional control signal 'v1' is required. This is because Y is assigned twice and the feedback signal 'v1' from 'Ya' must be acknowledged.

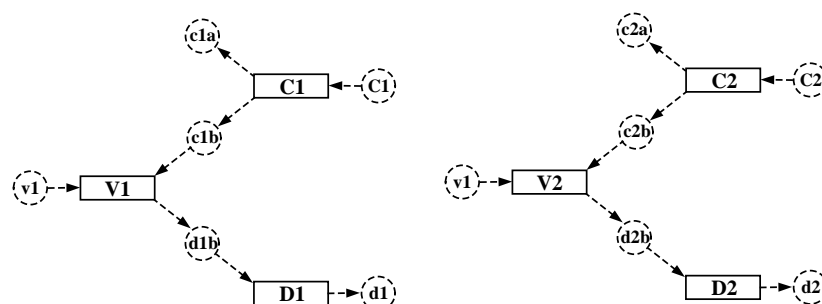


Figure 33: Seq3 local net

The global control net for the example is shown in Fig. 34. As in the last example three DC places and transitions are used again in a loop but in this case a dummy is used for the third transition.

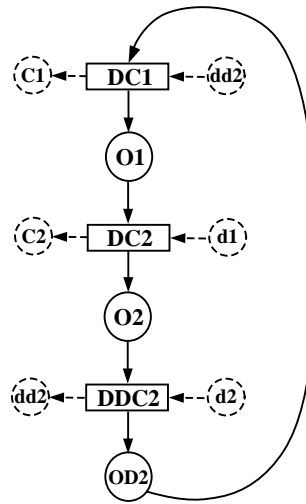


Figure 34: Seq3 global net

5.2 While loop nets

Below we show the net constuction for a while loop. The Verilog code for the example is shown in Fig. 35.

```

Module Wloop (V,R);
    input [7:0] V;
    output [7:0] R;
    reg [7:0] A;
    reg [7:0] X;
    reg [7:0] VV;
    reg [7:0] R;
    always
    begin
        A = 1;           // (1)
        X = 1;           // (1)
        VV = V;          // (1)
        while(100>=X)    // (2)
        begin
            VV = VV*2;    // (3)
            X = X+1;      // (3)
        end
        R <= VV+A;       // (4)
    end
endmodule

```

Figure 35: Verilog while loop example

The module declares one input, four variables and one variable output. Assignments are initially made to the three variables A, X and VV. When the while loop is called a multiplication and addition are repeatedly executed before the final result is output to R.

The data net generated for the while loop takes the form shown in Fig. 36.

At the top left of Fig. 36 a multiplexor net for the adder is used for passing the data required for the two

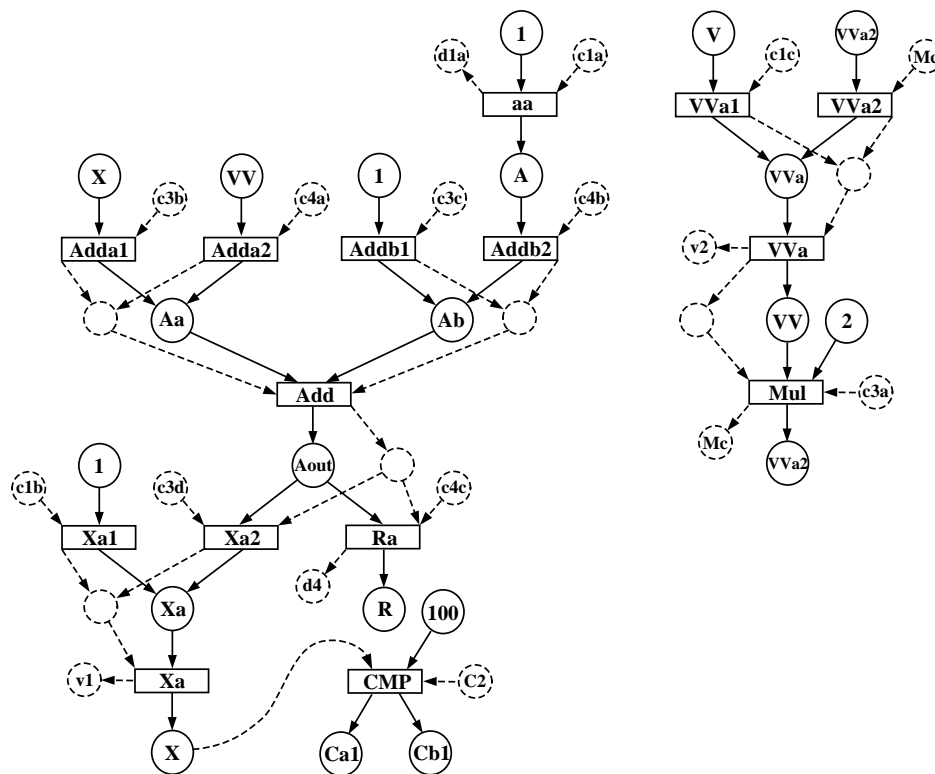


Figure 36: While loop data net

additions. At the top right a multiplexor net for the multiplier is used for initializing VV and for storing the output of the multiplication. A multiplexor net for assigning the variable X appears at the bottom. The subnet for the comparator appears next to this.

The local control nets for the example are shown in Fig. 37. Three nets are required for the control step stages 'C1', 'C3' and 'C4' respectively. 'C2' does not appear here as it is passed directly to the comparator net.

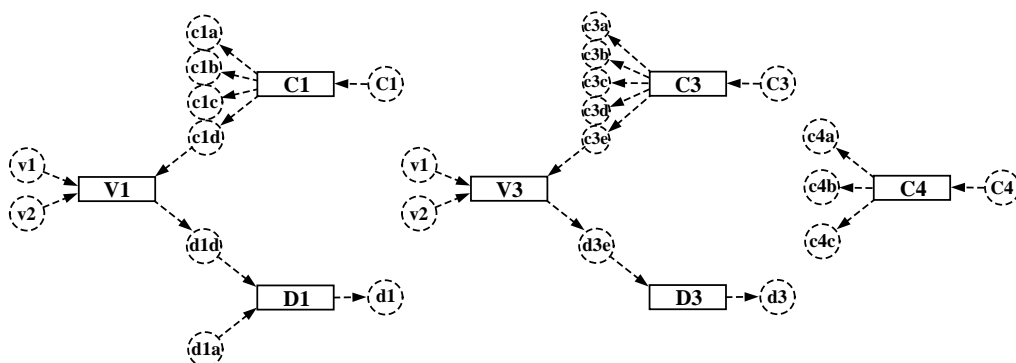


Figure 37: While loop local net

The global control net for the example is shown in Fig. 38. The 'C2' transition is used here for activating the comparator. A decision is made to stay in the loop governed by the feedback from comparator signal 'ca1' to transition 'DC3'. The loop requires the addition of dummy transition 'DDC3'. The exit from the loop is made via transition 'DC4' which uses feedback from comparator signal 'cb1'.

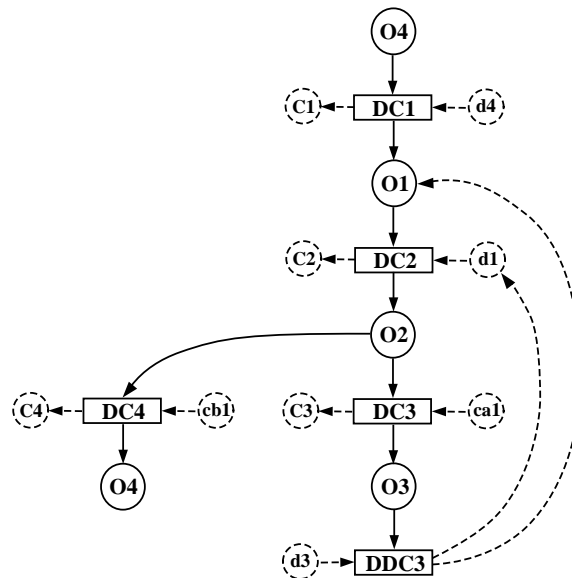


Figure 38: While loop global net

5.3 Repeat loop nets

Here we show the net construction for a repeat loop. The Verilog code for the example is shown in Fig. 39.

```
Module Rloop (V,R);
    input [7:0] V;
    output [7:0] R;
    reg [7:0] A;
    reg [7:0] X;
    reg [7:0] R;
    always
    begin
        A = V;                // (1)
        X = 1;                // (1)
        repeat(100)           // (2)
        begin
            X=X+1;            // (3)
        end
        R <= X+A;              // (5)
    end
endmodule
```

Figure 39: Verilog repeat loop example

The module declares one input, three variables and one variable output. Assignments are initially made to the two variables A and X. The repeat loop is called and one assignment is executed before outputting the result to R.

The data net generated for the repeat loop takes the form shown in Fig. 40.

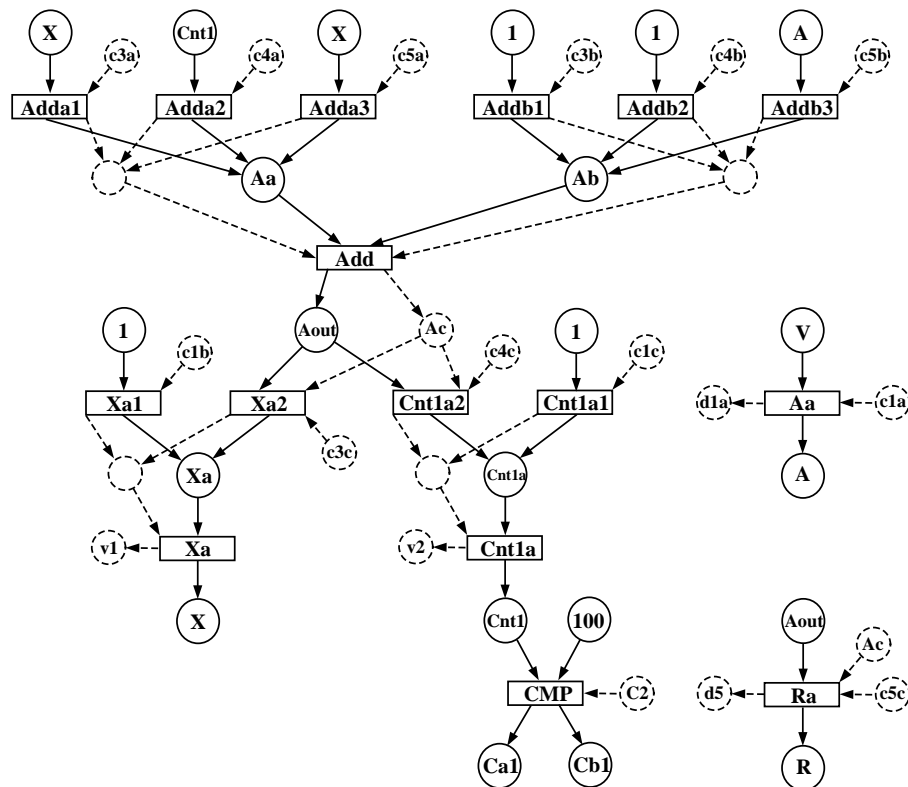


Figure 40: Repeat loop data net

At the top of Fig. 40 multiplexor nets for the adder are shown. An additional variable counter Cnt1 is automatically created by the synthesiser which keeps track of the loop iterations. This is done by checking if (Cnt1<100) and adding 1 to Cnt1 for each iteration. The net for this is shown connected to the comparator subnet at the bottom. The multiplexor net for the variable X is shown at the bottom left.

The local control nets for the example are shown in Fig. 41. Three nets are required for control step stages 'C1', 'C3', 'C4' and 'C5' respectively. Counting is done in stage 'C4'.

The global control net for the example is shown in Fig. 42. The DC2 transition is used for activating the comparator as for the previous example. This time the loop back does not require the addition of a dummy transition as there are already three stages in the loop.

5.4 For loop nets

Below we show the net constuction for a for loop. The Verilog code for the example is shown in Fig. 43.

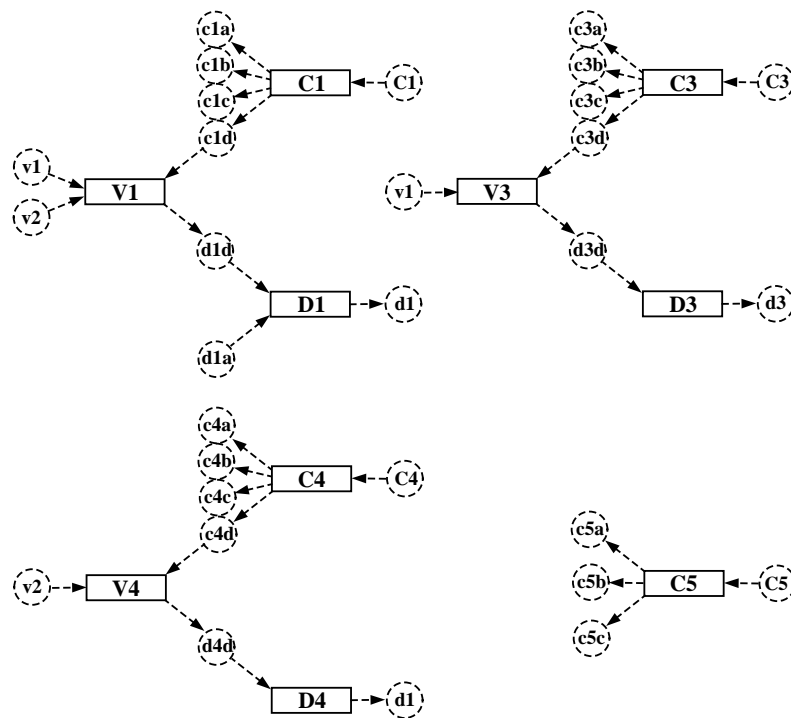


Figure 41: Repeat loop local net

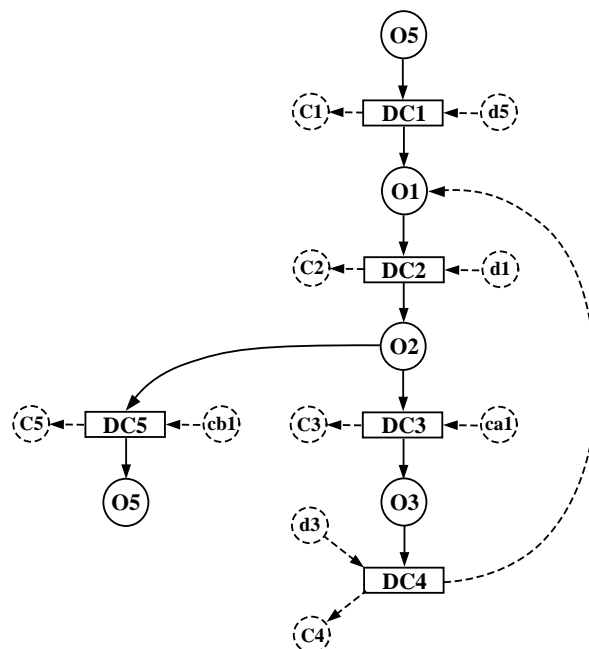


Figure 42: Repeat loop global net

```

Module Floop (V,R);
    input [7:0] V;
    output [7:0] R;
    reg [7:0] I;
    reg [7:0] X;
    reg [7:0] R;
    always
    begin
        X = V;                                // (1)
        for(I=0;I<100;I=I+1)                 // (2)
        begin
            X=X+1;                            // (3)
        end
        R <= X;                                // (5)
    end
endmodule

```

Figure 43: Verilog for loop example

The module declares one input, three variables and one variable output. Assignment is initially made to one variable X. The for loop is called and one addition is repeatedly executed before outputting the result to R.

The data net generated for the for loop takes the form shown in Fig. 44.

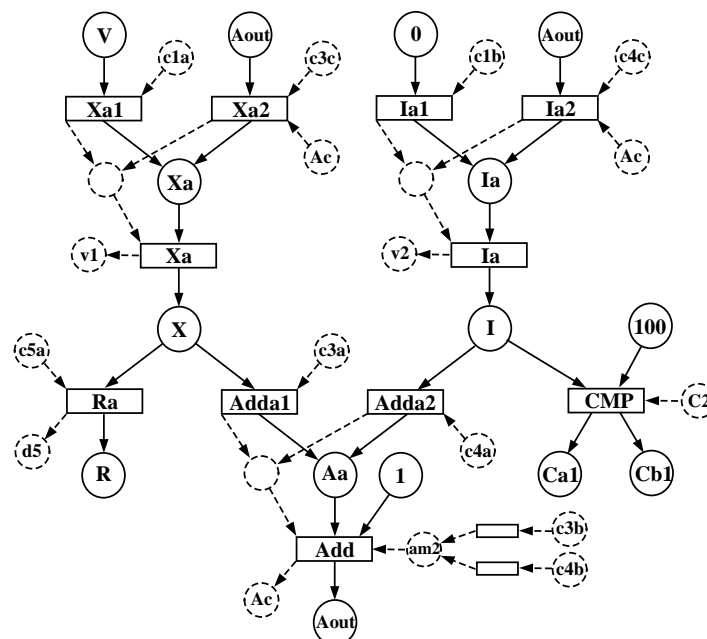


Figure 44: For loop data net

At the bottom of Fig. 44 a multiplexor net for the adder is shown. The multiplexor net for the variable X is shown at the top left. This time an additional variable counter I is created by the synthesiser which keeps track of the loop iterations. The net for this is shown connected to the comparator subnet.

The local control nets for the example are shown in Fig. 45. Three nets are required for control step stages 'C1', 'C3', 'C4' and 'C5' respectively. Counting is done in stage 'C4'.

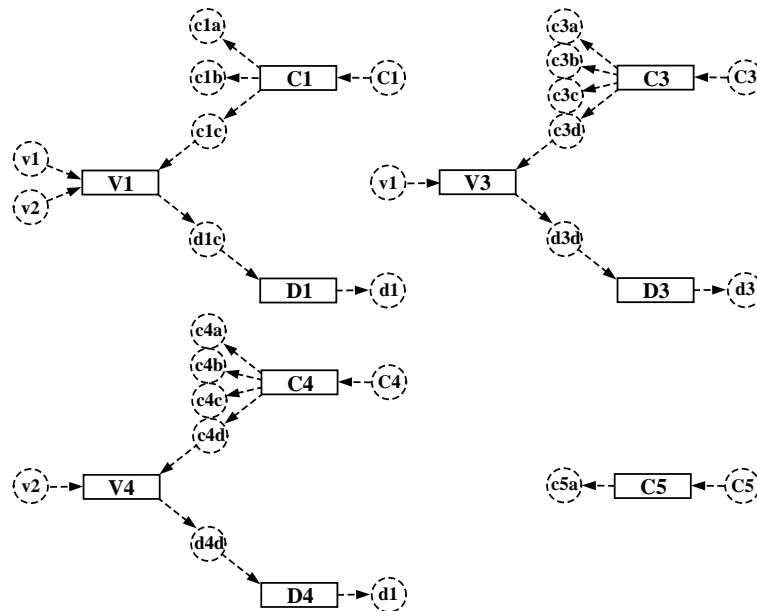


Figure 45: For loop local net

The global control net for the example is shown in Fig. 46. The 'DC2' transition is used for activating the comparator. As before the loop back does not require the addition of a dummy transition as there are three stages in the loop.

5.5 Conditional nets

5.5.1 If then else nets

Below is shown the net construction for an if then else example. The Verilog code for the example is shown in Fig. 47.

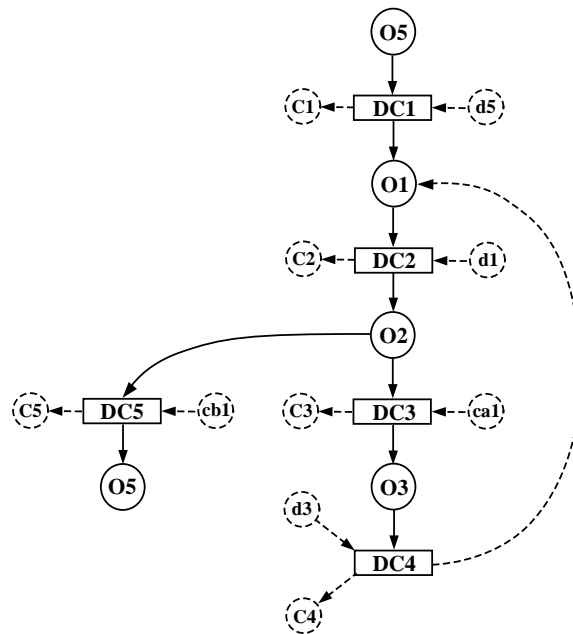


Figure 46: For loop global net

```

Module Ifcond (A,Z);
    input [7:0] A;
    output [7:0] Z;
    reg [7:0] B;
    reg [7:0] Z;
    always
    begin
        B = A;           // (1)
        if(B==0)         // (2)
        begin
            Z = 2;       // (3)
        end
        else
        begin
            Z = 3;       // (3)
        end
    end
endmodule

```

Figure 47: Verilog if then else example

The module declares one input A, one variable B and one variable output Z. The condition (B==0) is tested and then a conditional assignment is made to output Z.

The data net generated for the if then else construct takes the form shown in Fig. 48.

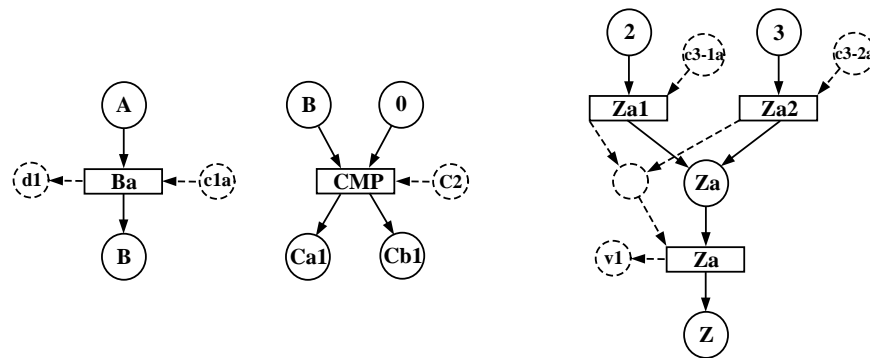


Figure 48: If then else data net

On the left of Fig. 48 the initial assignment to B is made. The comparator net in the centre of Fig. 48 is activated in stage 'C2' and either output 'Ca1' or 'Cb1' is fired depending on the test result for $(B==0)$. The multiplexor net on the right is used for selecting the appropriate conditional assignment to Z.

The local control nets for the example are shown in Fig. 49. Two nets for generating the path signals 'c3-1a' and 'c3-2a' are required respectively.

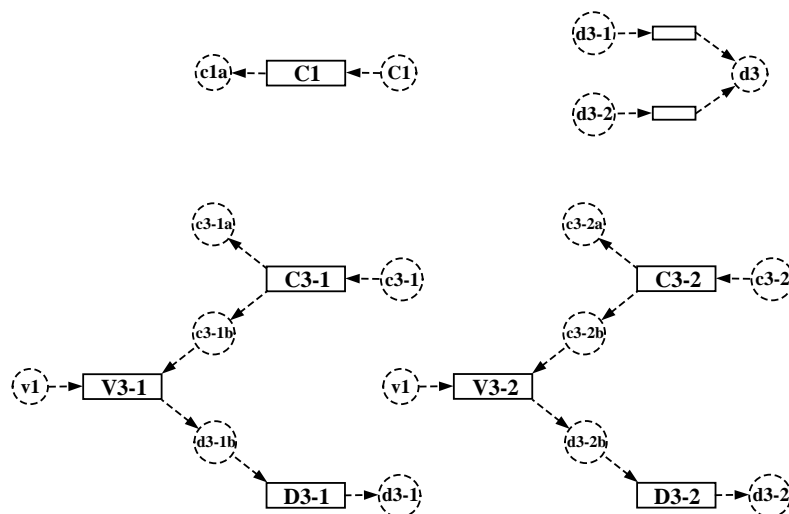


Figure 49: If then else local net

The global control net for the example is shown in Fig. 50. The 'DC2 transition' is used for activating the comparator. The path taken depends on the direct conditional signals from the datapath 'Ca1' or 'Cb1'.

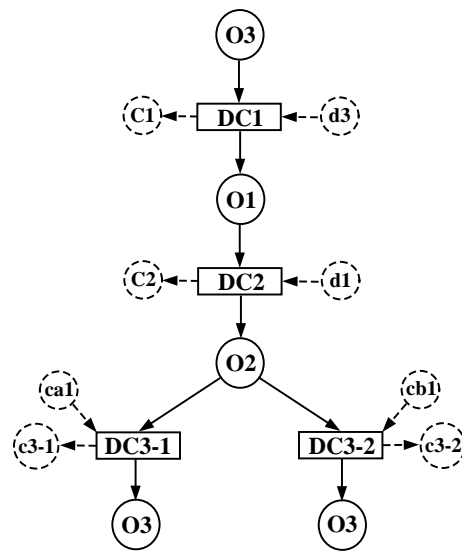


Figure 50: If then else global net

5.5.2 Nested if then else nets

Below we show the net construction for a nested if then else example. The Verilog code for the example is shown in Fig. 51.

```
Module Ifcondn (In1,In2,Y,Z);
    input [7:0] In1;
    input [7:0] In2;
    output [7:0] Y;
    output [7:0] Z;
    reg [7:0] A;
    reg [7:0] B;
    reg [7:0] Y;
    reg [7:0] Z;
    always
    begin
        A = In1;                // (1)
        B = In2;                // (1)
        if(A==0)                // (2)
        begin
            Y = 1;              // (3)
            if(B==0)
            begin
                Z = 3;          // (4)
            end
        else
        begin
            Z = 4;              // (4)
        end
        end
    else
    begin
        Y = 2;                  // (3)
    end
    end
endmodule
```

Figure 51: Verilog nested if then else example

The module declares two inputs A and B, and two variable outputs Y and Z. The condition (A==0) is first tested and if it is true an assignment is made to variable Y and the second comparison (B==0) is made. Otherwise the second assignment to Y is made.

The data net generated for the nested if then else construct is shown in Fig. 52.

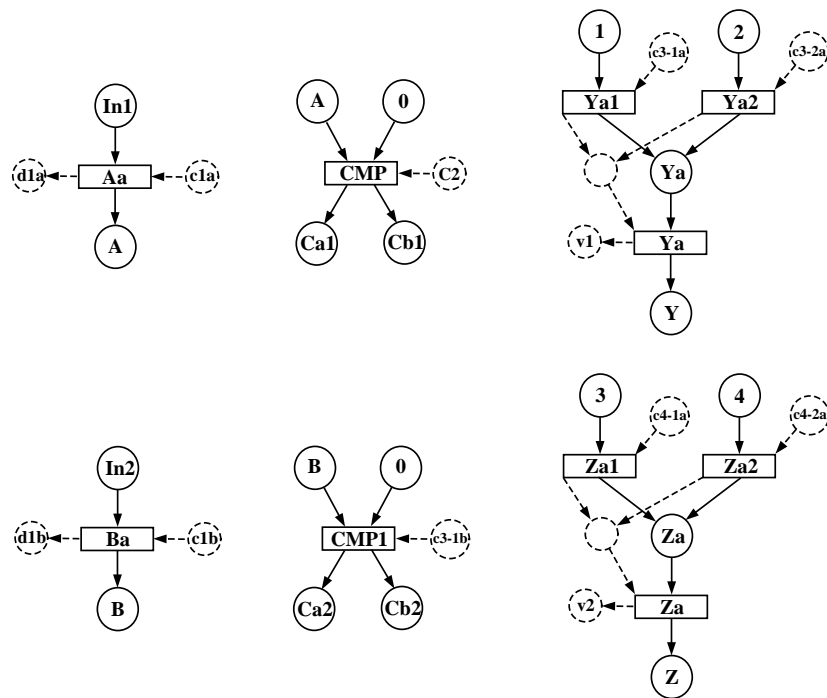


Figure 52: Nested if then else data net

The left hand side shows the assignment nets for A and B. The comparator nets for $(A==0)$ and $(B==0)$ are shown in the centre. The multiplexor assign nets for Y and Z are shown to the right.

The local control nets for the example are shown in Fig. 53. Four nets, 'c3-1', 'c3-2', 'c4-1' and 'c4-2' are shown for the four different paths taken. The nets at the top right are used for collecting the path end signals.

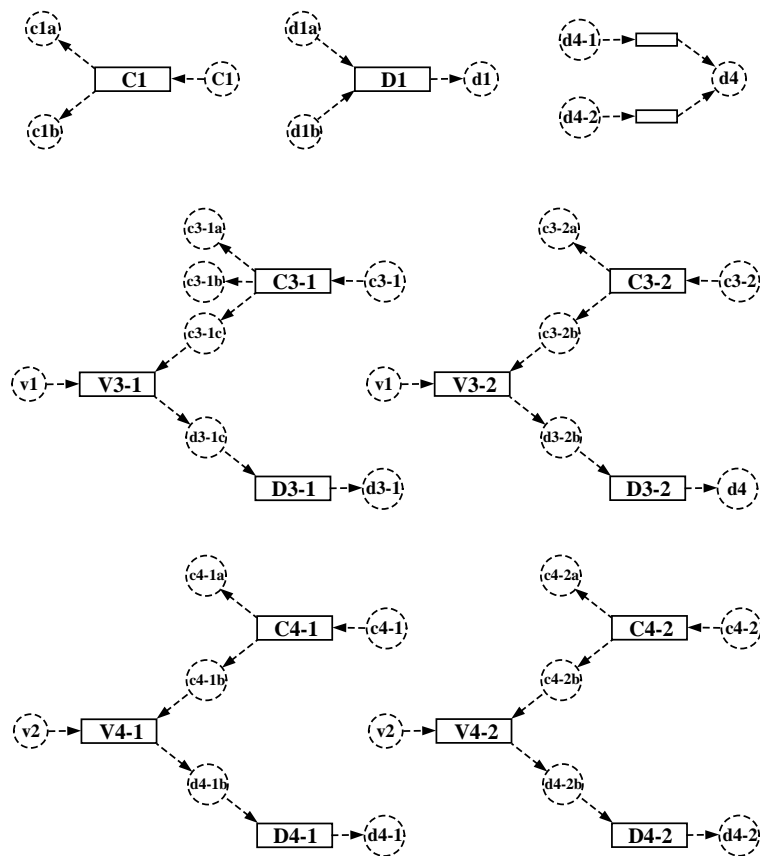


Figure 53: Nested if then else local net

The global control net for the example is shown in Fig. 54. The 'DC2' transition is used for activating the first comparator. Transition 'DC3-1' is used for activating the second comparator. All paths are rejoined at a common place.

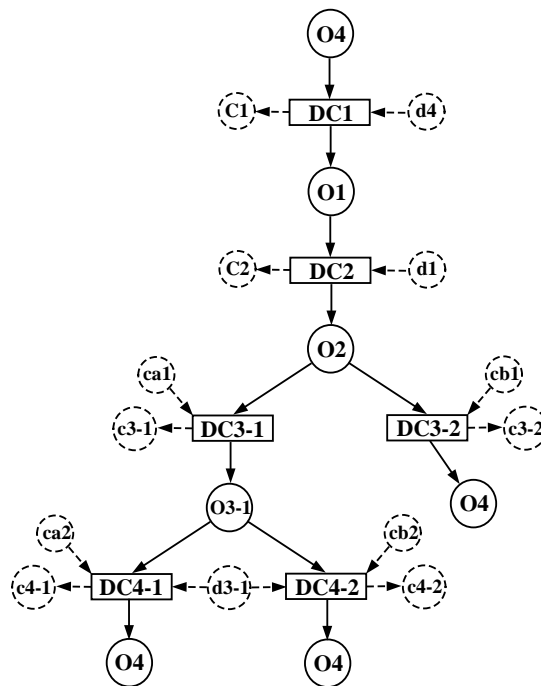


Figure 54: Nested if then else global net

5.5.3 Case nets

The net construction for a case construct is similar to an if then else construct. Similar net constructs are generated for the equivalent case statement. The Verilog code shown in Fig. 55. translates into similar net constructs as those for the if then else example of 5.5.1.

```
Module CaseA (A,Z);
    input [7:0] A;
    output [7:0] Z;
    reg [7:0] B;
    reg [7:0] Z;
    always
    begin
        B = A;           // (1)
        case(B==0)       // (2)
            1: Z = 2;     // (3)
            2: Z = 3;     // (3)
        end
    end
endmodule
```

Figure 55: Verilog case example

5.6 Fork nets

Below we show the net construction for a fork example. The Verilog code for the example is shown in Fig. 56.

```
Module ForkJoin (A,X,Y);
    input [7:0] A;
    output [7:0] X;
    output [7:0] Y;
    reg [7:0] X;
    reg [7:0] Y;
    always
        begin
            fork
                begin
                    X = A+5;        // (1)
                    X = X+2;        // (2)
                    X = X+1;        // (3)
                end
                begin
                    Y = A+6;        // (1)
                    Y = Y+3;        // (2)
                    Y = Y+2;        // (3)
                end
            join
        end
endmodule
```

Figure 56: Verilog fork example

The module declares one input A and two variable outputs X and Y. The body contains one fork join statement with two branches which are executed in parallel.

The data net generated for the fork construct takes the form shown in Fig. 57.

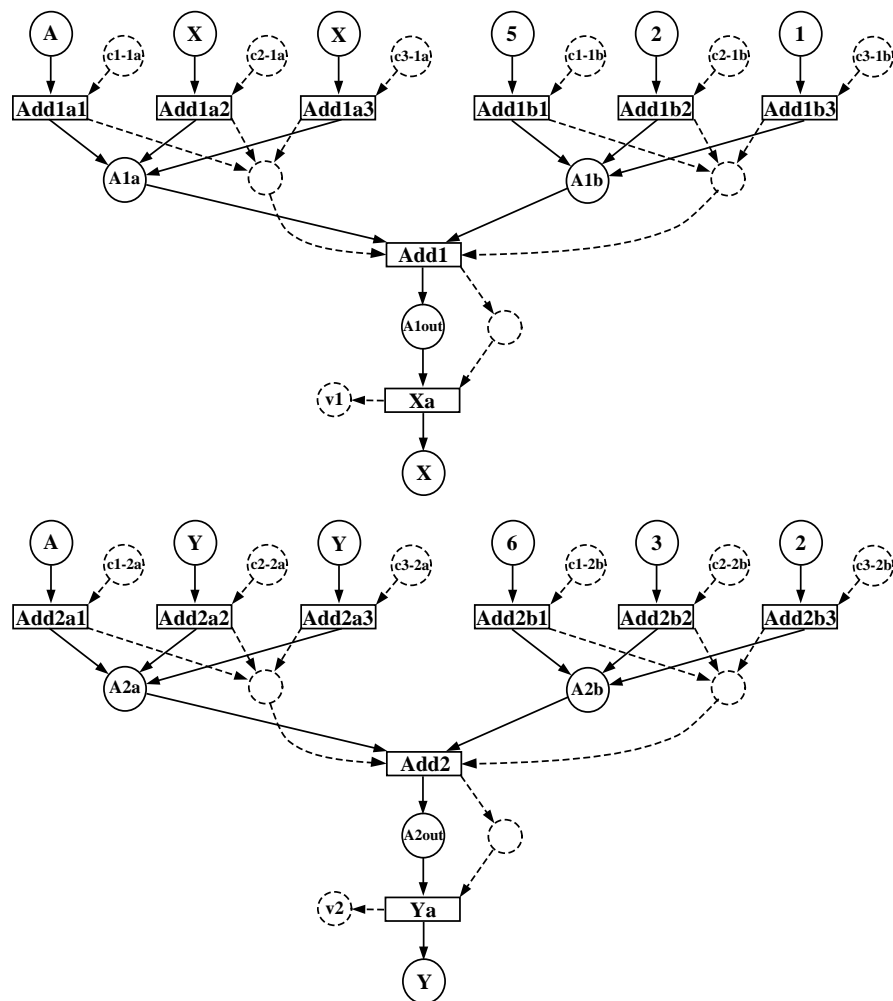


Figure 57: Fork data net

The fork generates two nets one for each fork branch and the bottom one represents the second. The multiplexor nets in the top fork branch are activated by signals of the form c1-1a, c2-1a, c3-1a etc. The second number to the right of the dash refers to the respective fork i.e. 1 in the above case.

The local control nets for the fork example which are similar are shown in Fig. 58. Three are shown on the left for the first thread and three are shown on the right for the other thread.

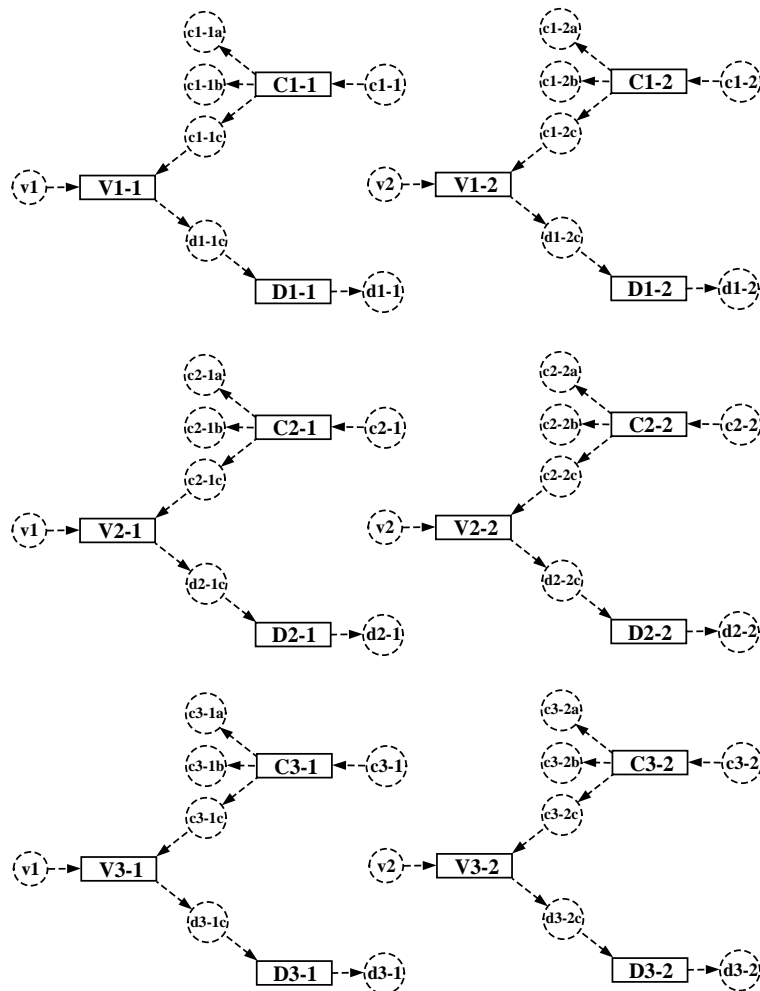


Figure 58: Fork local net

The global control net for the example is shown in Fig. 59. The 'DC1' transition is used for activating the initial assignments for both branches. The two separate paths generate the remaining control signals. A dummy cell at the bottom of the fork is used for joining the two branches back together.

5.7 Function nets

Here we present an example which uses statements with functions. The Verilog code for the example is shown in Fig. 60.

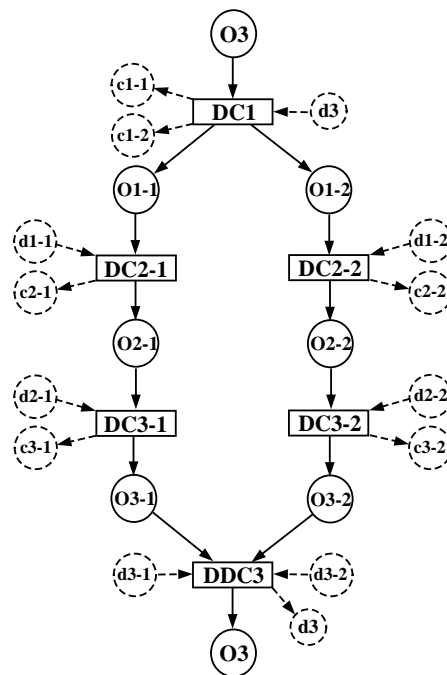


Figure 59: Fork global net

```

Module Func (A,Z);
    input [7:0] A;
    output [7:0] Z;
    reg [7:0] X;
    reg [7:0] Y;
    reg [7:0] Z;
    function [7:0] Fun
        input [7:0] X;
        input [7:0] Y;
    begin
        Fun = X+Y;
    end
endfunction

always
begin
    X = Fun(A,6);           // (1)
    Y = Fun(X,2);           // (2)
    Z = Fun(Y,1);           // (3)
end
endmodule

```

Figure 60: Verilog function example

The module declares one input, three variables and one variable output. Functional assignments are made in consecutive order to the three variables X, Y and Z.

The data net generated for functional assignments like this takes the form shown in Fig. 61.

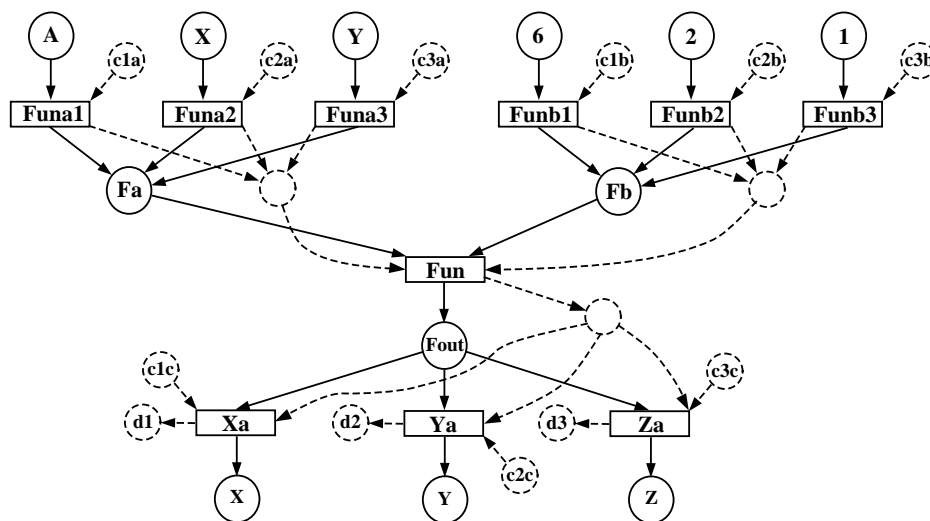


Figure 61: Function data net

At the top of Fig. 61 two three way multiplexor nets are used for multiplexing inputs to the function. A functional unit transition 'Fun', shown in the centre, executes all of the function calls and the results from this are passed to its output place. The corresponding assignment net is selected at the bottom.

The local control nets for the fuction example are shown in Fig. 62.

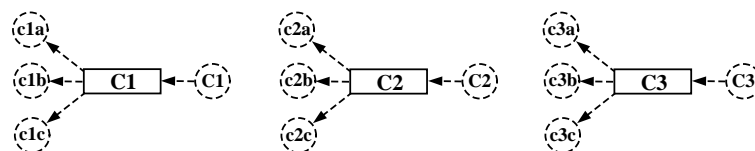


Figure 62: Function local nets

The global control net for the example is shown in Fig. 63. A sequence of three DC places and transitions are shown used in a loop.

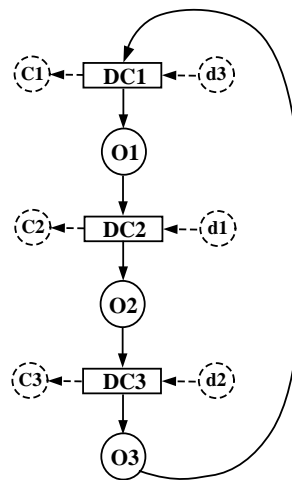


Figure 63: Function global net

5.8 Task nets

Below we show an example of a task and its calling statement in Verilog.

```

Module TaskA (A,Z);
    input [7:0] A;
    output [7:0] Z;
    reg [7:0] Z;

    Task TskA;
        input [7:0] A;
        input [7:0] B;
        output [7:0] Y;
    begin
        Y = A-B;
        Z = Y-1;
    end
endtask

always
begin
    TskA(A,6,Z);           // (1)
    Z = Z+A;               // (2)
end
endmodule

```

Figure 64: Verilog Task example

When translating Verilog Tasks into multinets we use the hierarchical properties of Petri nets. These are covered in detail in [11]. In hierarchical nets, hierarchical transitions are used for instancing subnets and fusion places are used for linking common places. Each task is translated into a subnet by translating the statements inside the task normally. The datapath net for the task TskA is shown in Fig. 65.

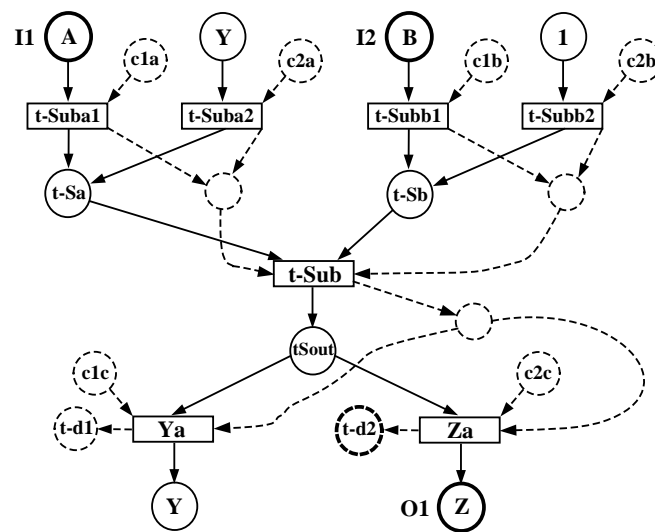


Figure 65: Task data net

Normally, each task is assigned an identifiable tag so that all its transitions and places are distinguishable from other tasks. In this example it is assumed that all transitions and places are tagged with the letter t (not all tags are shown in the diagram). Port places (places which are used as inputs and outputs) are shown highlighted. These act as inputs and outputs for the calling transition. The local nets for the task TskA are shown in Fig. 66. Here the transitions and input places are shown tagged by the letter t.

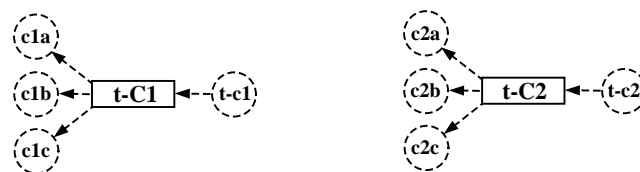


Figure 66: Task local net

The global nets for TskA are shown in Fig. 67. A sequence of two DCs is used. Input and output places are shown highlighted.

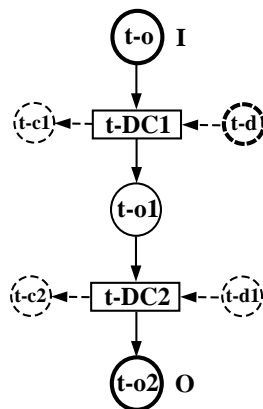


Figure 67: Task global net

The datapath net for the main code which calls TskA is shown in Fig. 68. Here a hierarchical transition 't-TskA' is used for calling the TskA subnet. For TskA to become active inputs 'A', '6', 't-o' and 't-d' must all become alive first. When the TskA task has been executed it returns an exit token in the fusion place 't-d2' (shown dashed) which enables the main datapath net to continue.

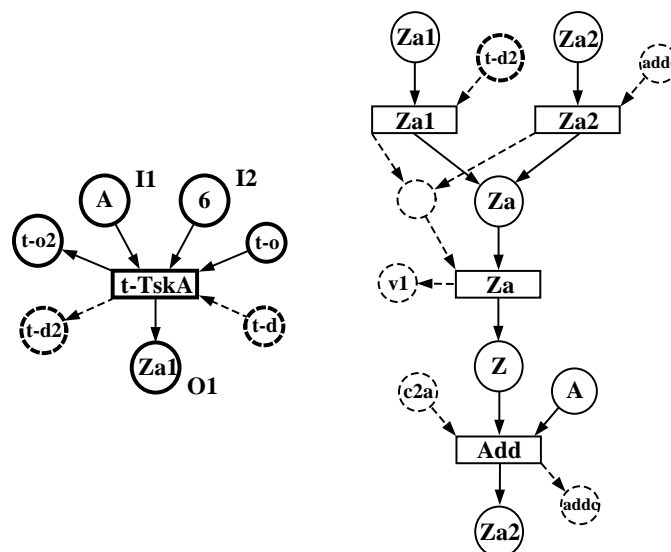


Figure 68: Task routine data net

The local nets for the main code are shown in Fig. 69.

The global nets for the main code are shown in Fig. 70. This consists of three DC transitions. The first DC transition fires the first part of the main net and simultaneously passes a token to the TskA net via 't-o' to activate the subnet. A return token is passed back upon completion of the TskA subnet to place 't-o2'. This enables the main net to continue firing.

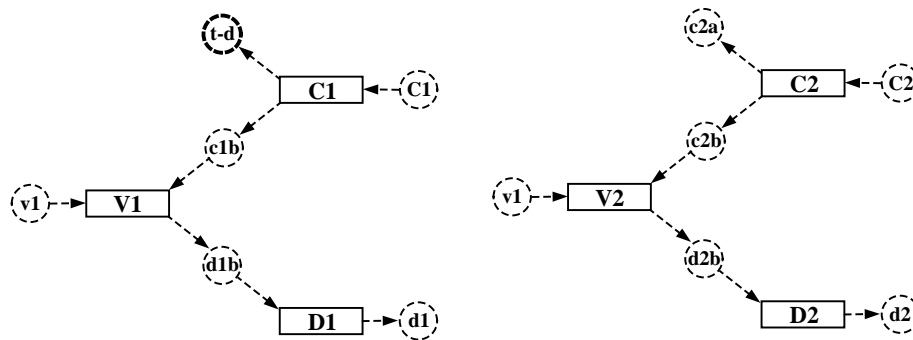


Figure 69: Task routine local net

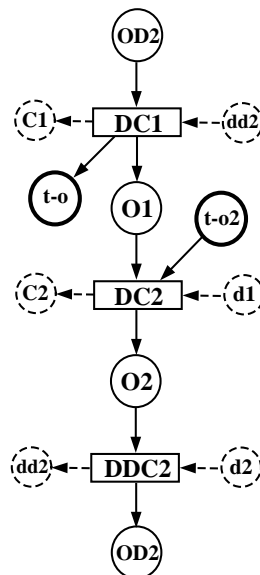


Figure 70: Task routine global net

Optimizations can be applied by selecting the appropriate options in the interface. If the option tag optimize is selected the synthesizer proceeds to reduce the number of cells in the global net. It does this by eliminating front end cells within the task net and replacing the firing signals for these from the main net.

6 CD decoder example

6.1 Specification

This section presents a larger example of an error decoder found in [18]. The error decoder implements error-detection on the audio information recorded on compact discs using a syndrome computation algorithm. The Verilog specification for the error decoder is shown in Figs. 71 and 72.

```
Module DEC (Reset,Start,Tw,Cw,Sw,Ew,Lw);
    input      Reset;
    input      Start;
    input      Tw;
    input [7:0] Cw;
    output     Sw;
    output [7:0] Ew;
    output [5:0] Lw;
    reg [31:0]  Syn;
    reg [7:0]   E, S;
    reg [5:0]   N;
    reg [1:0]   T;
    reg [1:0]   Stat;
    reg [1:0]   Sw;
    `include "ff_v.v"
    Task Frkeq1;
        input [1:0] T;
        output [5:0] N;
        output [31:0] Syn;
        reg [1:0] T;
        reg [5:0] N;
    begin
        syn = 0;
        if(T==0)
            begin
                n = 27;
            end
        else
            begin
                n = 32;
            end
    end
endtask
    Task Frkeq2;
        input [1:0] T;
        input [31:0] Syn;
        output [5:0] N;
        output [7:0] E;
        reg [1:0] T;
        reg [5:0] N;
        reg [7:0] E;
    begin
        e = syn[7:0];
        if(T==0)
            begin
                n = 27;
            end
        else
            begin
                n = 32;
            end
    end
```

Figure 71: Verilog DEC example


```

        end
    endtask
    Task Whlfrk;
        input [1:0] Tst;
        input [7:0] Cw;
        inout [5:0] N;
        inout [31:0] Syn;
        reg [5:0] N;
        reg [31:0] Syn;
    begin
        while(Chk(Tst,N,Syn))
            begin
                N = N - 1;
                if(Tst==1) S = Cw;
                else S = 0;
                Syn = Horner(S,Syn);
            end
        end
    endtask
    always
    begin
        Frkeq1(T,N,Syn);
        Whlfrk(1,Cw,N,Syn);
        Frkeq2(T,Syn,N,Ew);
        Whlfrk(0,Cw,N,Syn);
        Stat = n[5];
        Syn = Shffle(Syn);
        Stat = Chkstat(Stat,Syn);
        Syn = Shffle(Syn);
        Stat = Chkstat(Stat,Syn);
        Sw = Stat;
        Lw = n;
    end
endmodule

```

Figure 72: Verilog DEC example contn'd

6.2 Net generation

Firstly we show the net generation for the tasks. The datapath net for the task Frkeq1 is shown in Fig. 73. Tag T1 is the identifying task tag appended to all transitions and places.

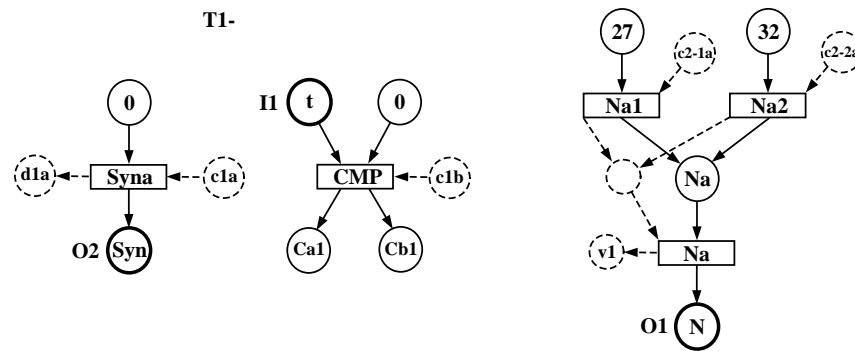


Figure 73: Frkeq1 task data net

The local nets for the task Frkeq1 are shown in Fig. 74.

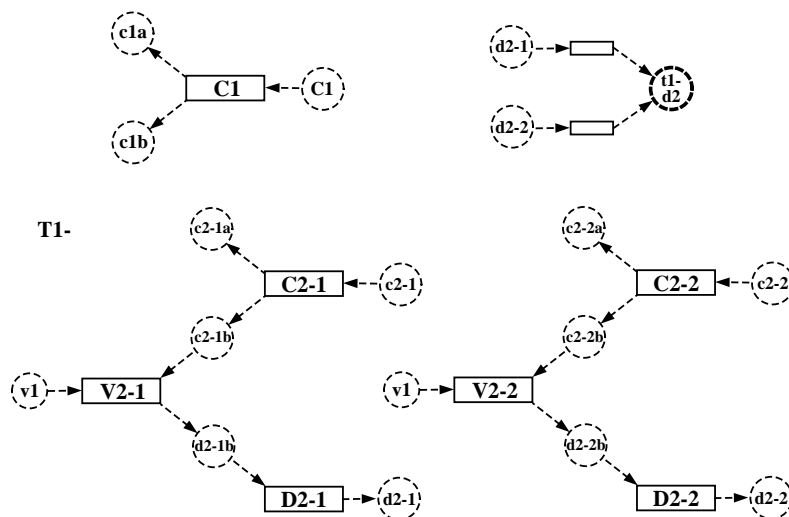


Figure 74: Frkeq1 local nets

The global net for the task Frkeq1 is shown in Fig. 75.

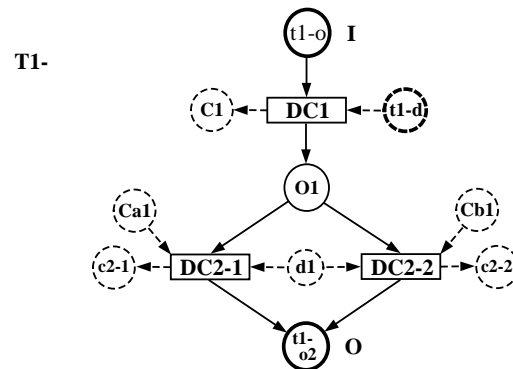


Figure 75: Frkeq1 global net

The nets for task Frkeq2 are similar to Frkeq1 and are not shown here.

The datapath net for the task Whlfrk is shown in Fig. 76. Tag T3 is the identifying task tag appended to all transitions and places.

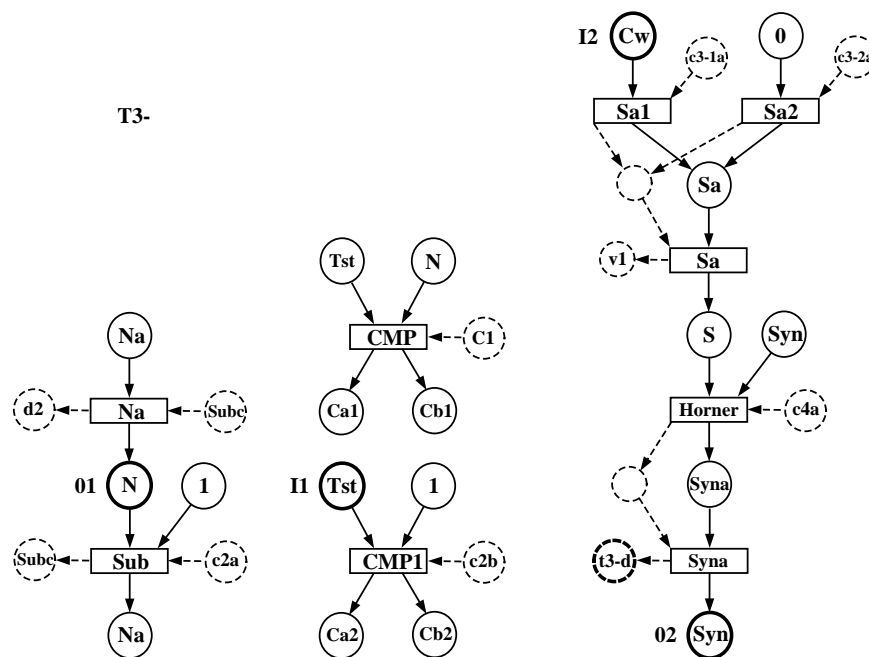


Figure 76: Whlfrk task data net

The local nets for the task Whlfrk are shown in Fig. 77.

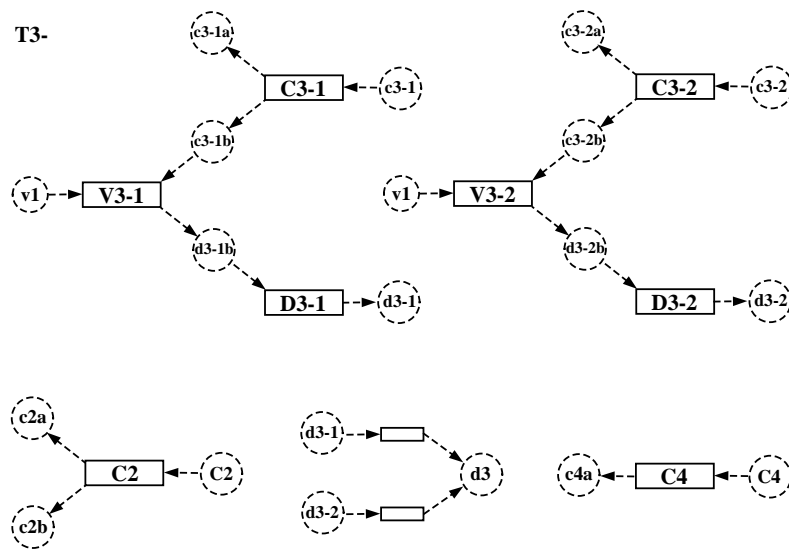


Figure 77: Whlfrk local nets

The global net for the task Whlfrk is shown in Fig. 78. This first forks and then splits to take into account the condition.

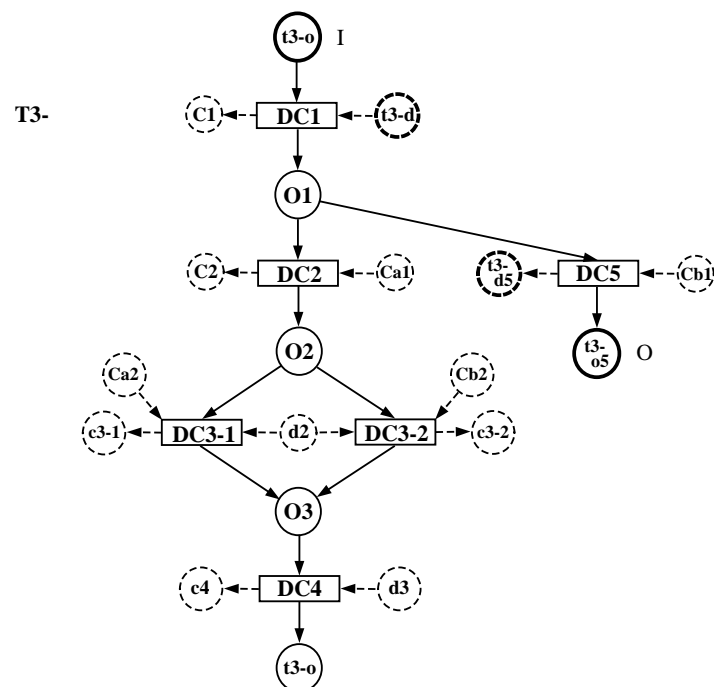


Figure 78: Whlfrk global net

The datapath nets for the decoder are shown in Figs. 79 and 80. The task calling nets for the decoder are shown in Fig. 79. Here hierarchical transitions t1-Fk1, t3-Wf and t2-Fk2 are used for instantiating the

task subnets Frkeq1, Whlfrk and Frkeq2. The multiplexor assignment nets for N and Syn are shown to the right.

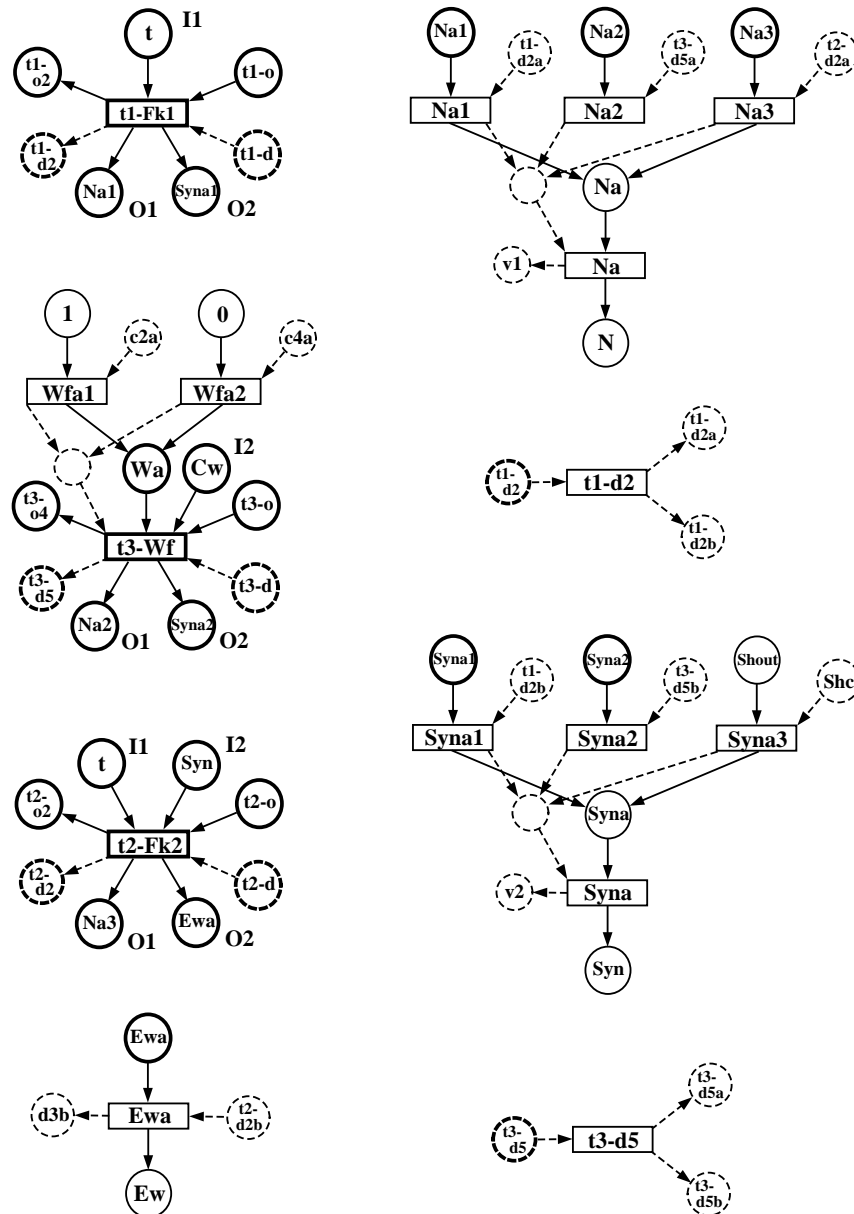


Figure 79: Decoder data net 1

The remaining function and assignment nets for the decoder are shown in Fig. 80.

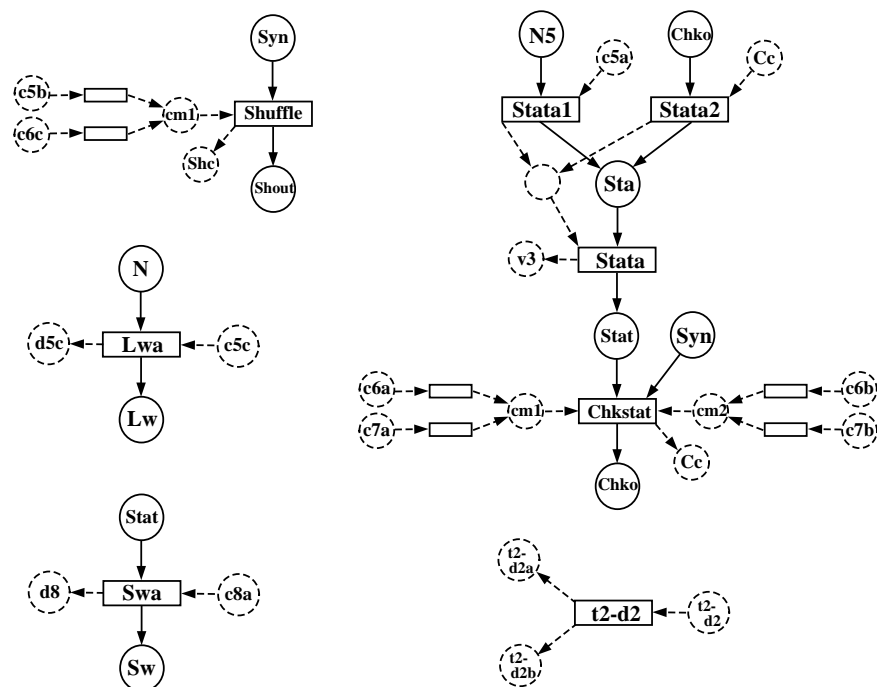


Figure 80: Decoder data net 2

The local nets for the decoder are shown in Fig. 81.

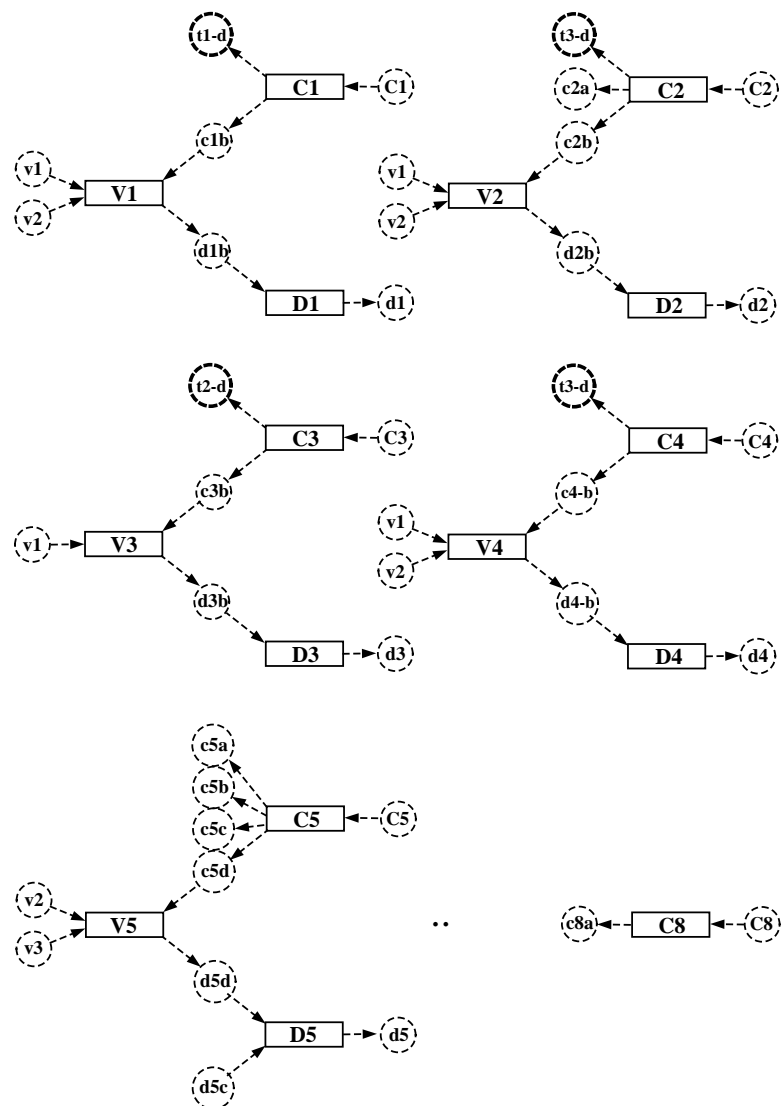


Figure 81: Decoder local nets

The global net for the decoder is shown in Fig. 82. The tasks are initiated in the first steps. Some assignment steps are missed out for brevity.

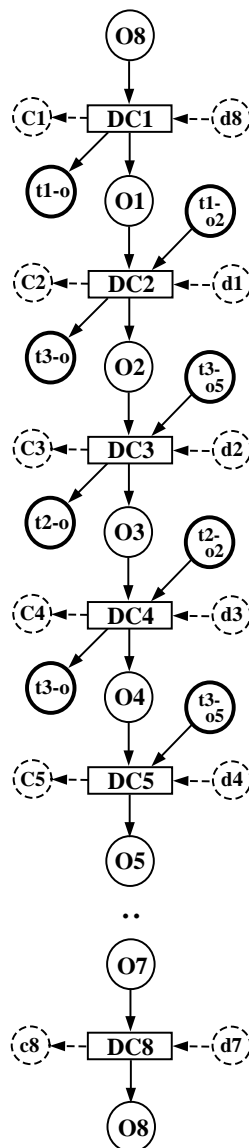


Figure 82: Decoder global net

7 Pipelining

7.1 Pipelining

The throughput of a digital system is the rate at which data is supplied to and produced by the system. This is improved by pipelining. In a pipelined system pipeline registers are used to store data at intermediate stages in the design so that each stage can process its data at the same time as the other stages. Two types of pipelining exist: structural and functional. In structural pipelining internal pipeline registers are used inside functional units for storing intermediate data at specific intervals. In functional pipelining pipeline registers are positioned between functional units so that the functional units can be fired simultaneously.

Synchronous systems ordinarily make use of concurrent register transfers in order to achieve pipelining. In asynchronous systems the concurrent firing of registers cannot be guaranteed in the same way as for synchronous as only an approximation to concurrent timing can be made due to feedback constraints.

High level synthesis systems target both structural and functional pipelining. In structural pipelining a functional unit with a set of pipestages is selected. For functional pipelining a cutset in the datapath can be worked out by the system from a specified user throughput rate. For our high level synthesis approach we take a lower level view of pipelining where the cutsets are specified more directly by the user. We concern ourselves only with functional pipelining here. To create a functional pipeline the user initially enters a sequential specification but the timing behaviour may be subsequently altered by specifying the outputs of various functions as pipestages. This is demonstrated in the following example.

7.2 Pipeline Example

Here we present a pipeline example. The Verilog code for the example is shown in Fig. 83.

```
Module Pipe (A,B,Z);
    input [7:0] A;
    input [7:0] B;
    output [7:0] Z;
    reg [7:0] X;
    reg [7:0] Y;
    reg [7:0] Z;
    function [7:0] FunA
        input [7:0] X;
        input [7:0] Y;
    begin
        FunA = X+Y;
    end
endfunction
function [7:0] FunB
    input [7:0] X;
    input [7:0] Y;
begin
    FunB = X-Y;
end
endfunction
always
begin
    Y = FunA(A,1);           // (1)
    Z = FunB(Y,1);           // (2)
    Y = FunA(B,6);           // (2)
    Z = FunB(Y,2);           // (3)
end
endmodule
```

Figure 83: Verilog pipeline example

The module declares two inputs, three variables and one variable output. Functional assignments are made in consecutive order to the variables Y and Z.

To specify a pipeline for the functions the user must first compile the specification and then select the options menu in the tool interface followed by pipeline. A window then appears presenting the user with

a choice of functions to pipeline. The functions to be pipelined must be selected from this list, in this case FunA and FunB, and then the OK button should be clicked. The pipeline nets can now be synthesized.

The data net generated for a two stage pipeline like this takes the form shown in Fig. 84. All the transitions and places are tagged with a unique p number representing the pipeline stage.

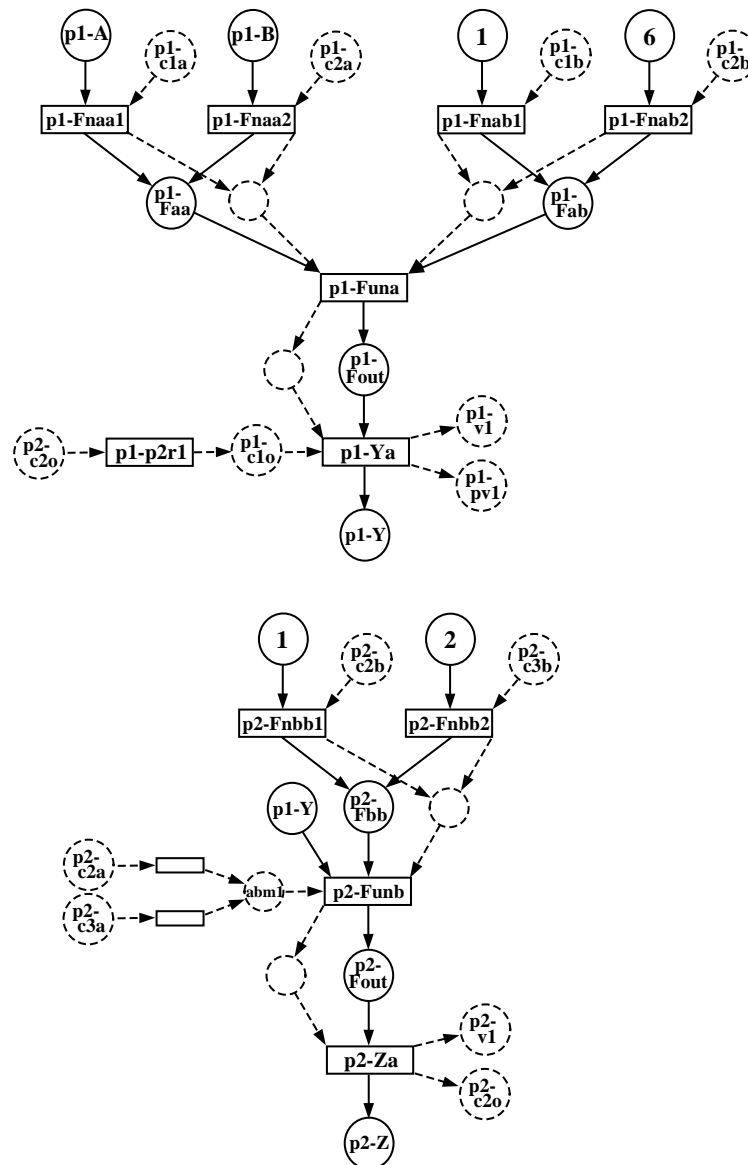


Figure 84: Pipeline data net

The data net for the first stage of the pipeline is shown at the top of Fig. 84. This consists of a normal multiplexor function net at the top and a register assignment net at the bottom. The register assignment net inputs a feedback signal from the 2nd pipestage via 'p2-c2o' which is initialised for firing. The register control output comprises a feedback signal 'p1-v1' and a feedforward signal 'p1-vp1' which is piped forward to the global control nets for the second stage.

The second stage of the pipeline is shown at the bottom below the first stage. It also comprises a multiplexor function net at the top and a register assignment net at the bottom. The function inputs the register output from the first pipestage. The second stage uses its own set of control inputs.

The local control nets for the pipeline example are shown in Fig. 85. These are split into two halves. The top half shows the local control nets for the first pipestage. The bottom half shows the local control nets for the second pipestage.

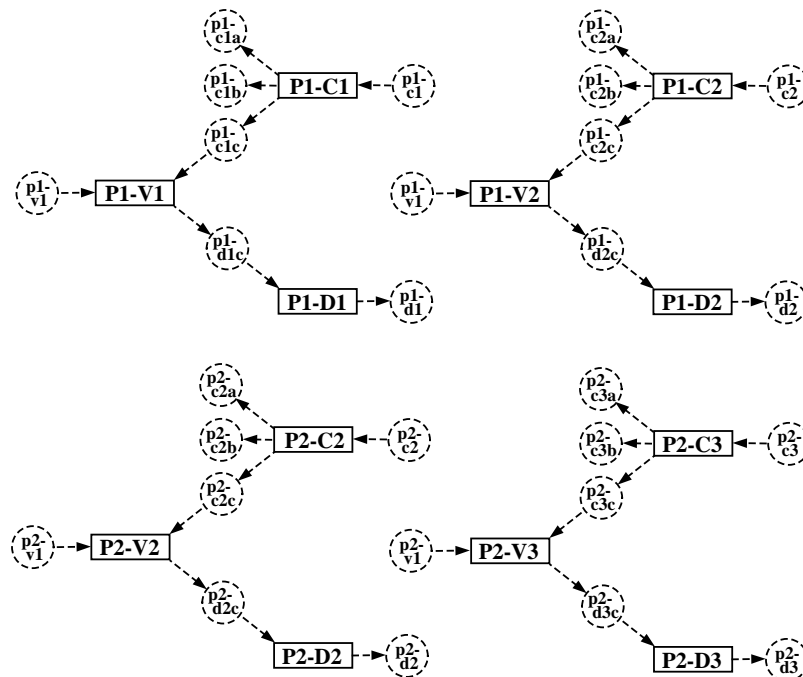


Figure 85: Pipeline local nets

Each local net transfers signals only to the pipestage specified by its pipeline tag.

The global control nets for the pipeline example are shown in Fig. 86. A sequence of three DC places and transitions are shown used in a loop for each pipestage. The global control net for the first stage is shown on the left. It consists of an ordinary sequential DC net. The global control net for the second stage is shown on the right. It consists of an ordinary sequential DC net with the added pipeline control signal 'p1-vp1' fed forward from the previous data stage. Each pipeline stage is initialised for starting.

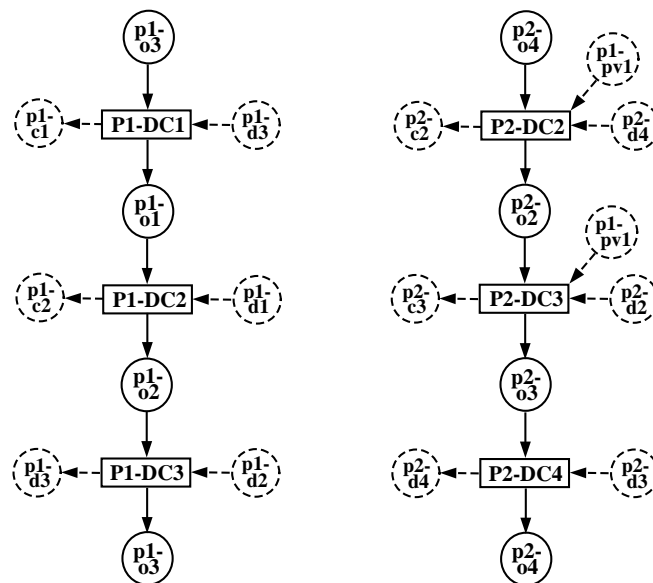


Figure 86: Pipeline global net

8 AES example

The example presented in this section is that of Rijndael's AES block cipher encryption security algorithm [19]. The symmetric block cipher Rijndael was standardized by the National Institute of Standards and Technology as Advanced Encryption Standard (AES) in 2001. The AES is the successor of the Data Encryption Standard (DES).

8.1 Specification

The block diagram for the encryption is shown in Fig. 87.

The diagram shows the basic steps that are used in the data encryption algorithm. It is split into a number of different stages. After data is entered (128 bits in this example) various transformations are performed. In each normal round a round key is first added. This is followed by a byte substitution stage. Then the data in different rows are shifted. Finally the data in similar columns is mixed. In each round (n iterations) that the data is processed a new key is generated. In the final round the mix columns stage is missed out before the encrypted data is output.

The Verilog code for the AES algorithm is shown in Fig. 88 and Fig. 89.

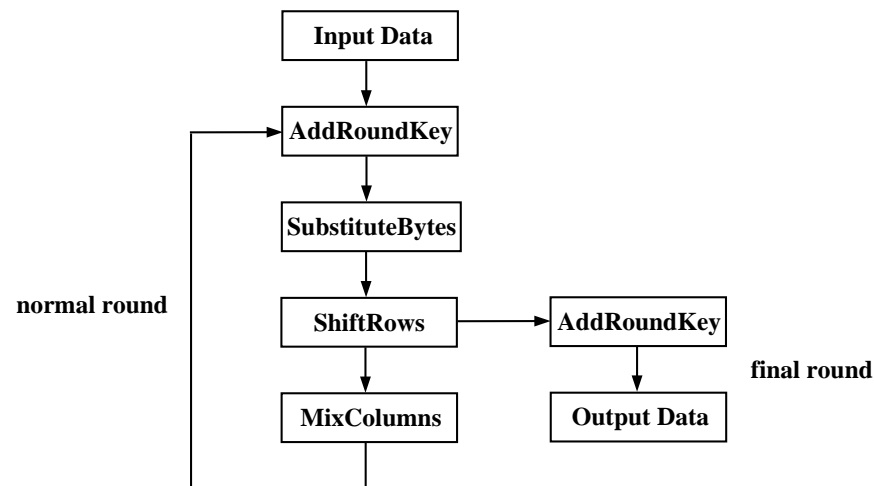


Figure 87: Block diagram of AES

```

Module AES (Fstart,Sel,Rst,Ld1,Key_in,Text_in,Text_out);
    input      Fstart;
    input      Sel, Rst;
    input      Ld1;
    input [127:0] Key_in;
    input [127:0] Text_in;
    output [127:0] Text_out;

    reg        Ld;
    reg        Tst;
    reg [127:0] Key;
    reg [127:0] Tw;
    reg [31:0]  W0, W1, W2, W3;
    reg [31:0]  Sbout;
    reg [31:0]  SrouT;
    reg [127:0] Mout;
    reg [31:0]  C0, C1, C2, C3;
    reg [31:0]  M0, M1, M2, M3;
    reg [31:0]  T0, T1, T2, T3;
    reg [127:0] Mem [10:0];
    reg [127:0] Text_out;
    reg        Dn;
    reg        N;
    `include "fv_v.v"
    always
    begin
        begin
            if(fstart==1)
            begin
                Dn = 0;
                Ld = 1;
                T0 = Text_in[127:96];
                T1 = Text_in[95:64];
                T2 = Text_in[63:32];
                T3 = Text_in[31:00];
            end
        end
    end

```

Figure 88: Verilog AES example

```

Key = Key_in;
W0 = Key[127:96];
W1 = Key[95:64];
W2 = Key[63:32];
W3 = Key[31:00];
N = 0;
while(10>=N);
begin
  if(N==10) Tst = 0;
  else Tst = 1;
  fork
    begin
      Sbout = sbox((C0[31:24],C1[31:24],C2[31:24],C3[31:24],Sel,Ld);
      Srou = srow(Sbout,1);
      Mout = mix(0,Srou,Sel,Tst,Mout);
      Sbout = sbox((C0[23:16],C1[23:16],C2[23:16],C3[23:16],Sel,Ld);
      Srou = srow(Sbout,2);
      Mout = mix(1,Srou,Sel,Tst,Mout);
      Sbout = sbox((C0[15:8],C1[15:8],C2[15:8],C3[15:8],Sel,Ld);
      Srou = srow(Sbout,3);
      Mout = mix(2,Srou,Sel,Tst,Mout);
      Sbout = sbox((C0[7:0],C1[7:0],C2[7:0],C3[7:0],Sel,Ld);
      Srou = srow(Sbout,4);
      Mout = mix(3,Srou,Sel,Tst,Mout);
      C0 = Exor1(Tst,Sel,Ld,Mout[127:96], T0,W0);
      C1 = Exor2(Tst,Sel,Ld,Mout[95:64], T1,W1);
      C2 = Exor3(Tst,Sel,Ld,Mout[63:32], T2,W2);
      C3 = Exor4(Tst,Sel,Ld,Mout[31:0], T3,W3);
    end
  begin
    Tw = key_exp(N,1,0,Key);
  end
  join
  N = N+1;
  W0 = Tw[127:96];
  W1 = Tw[95:64];
  W2 = Tw[63:32];
  W3 = Tw[31:0];
  Ld = 0;
end
Text_out = {C0,C1,C2,C3};
end
endmodule

```

Figure 89: Verilog AES example contn'd

8.2 Net generation

To specify a pipeline for the functions the user must first compile the specification and then select the options menu in the tool interface followed by pipeline. A window then appears presenting the user with a choice of functions to pipeline. The functions to be pipelined must be selected, in this case sbox, srow and mix, and then the OK button should be clicked. The data nets generated for the datapath of the AES example are shown in Figs 90-94. Fig. 90 shows the data nets for the inputting of the text data 'T0..T3',

the inputting of the key 'W0..W3' and the nets associated with the start and loop tests.

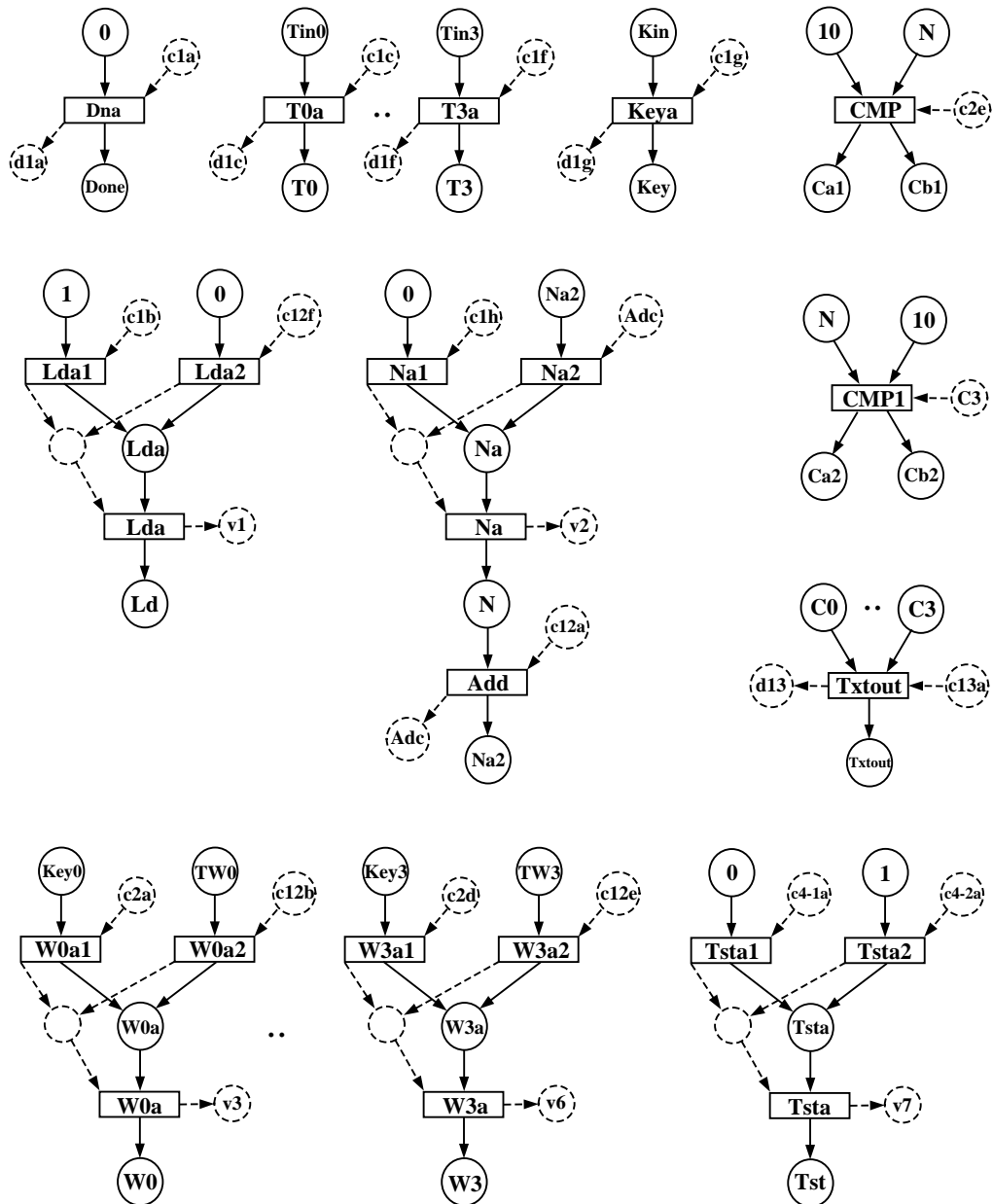


Figure 90: AES data input and test nets

Double dots .. between nets are used to save repetitive drawing of nets and imply that the nets include those numbered from left .. right or those numbered from top .. bottom. For example, the transitions for inputting text at the top range from T0a .. T3a. Fig. 91 shows the pipeline net for the sbox. The P1- label at the left implies that, unless already tagged, any transitions and places are tagged with a P1-.

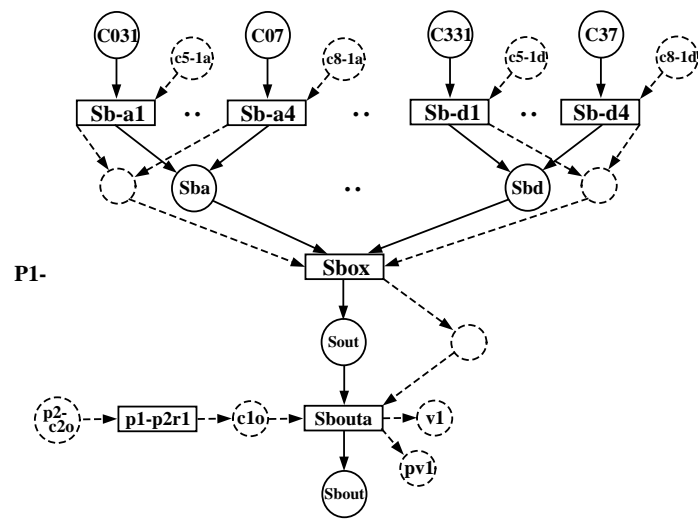


Figure 91: AES sbox pipeline data net

Fig. 92 shows the pipeline net for the shift rows function. The tag used for this pipestage is P2-.

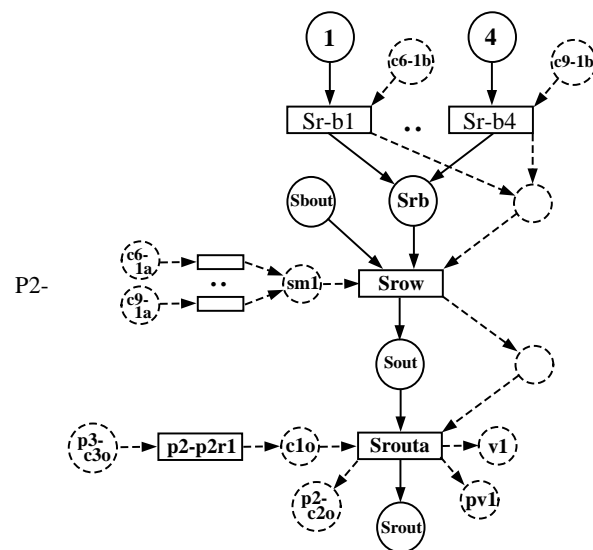


Figure 92: AES shift row pipeline data net

Fig. 93 shows the pipeline net for the mix columns function. The tag used for the mix pipestage is P3-.

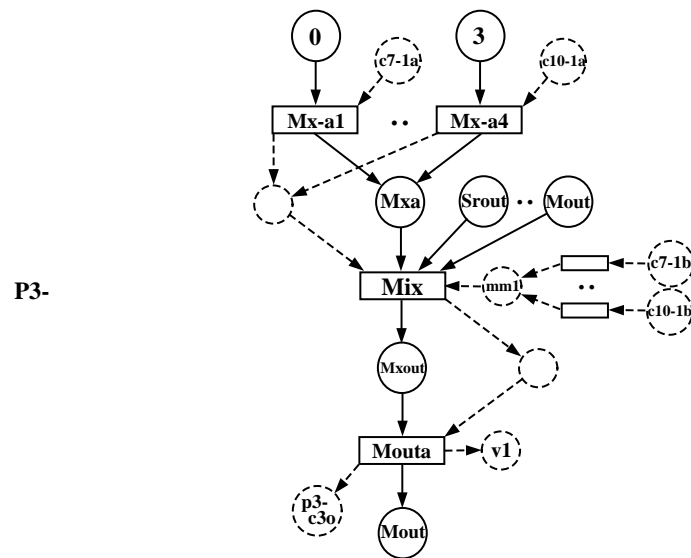


Figure 93: AES mix column pipeline data net

Fig. 94 shows the net for the exor functions. These are not pipelined and as they are really a part of the main data net they are not tagged.

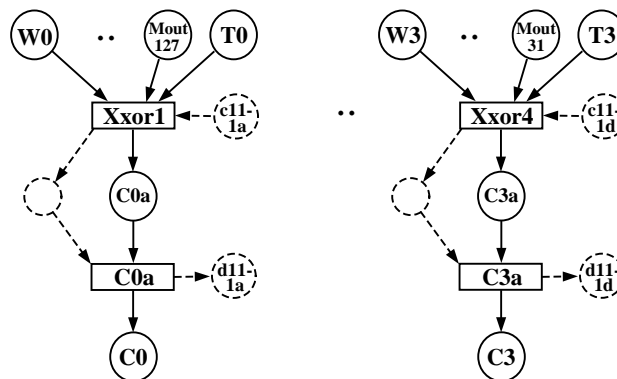


Figure 94: AES exor key data net

Fig. 95 shows the local nets for the main datapath.

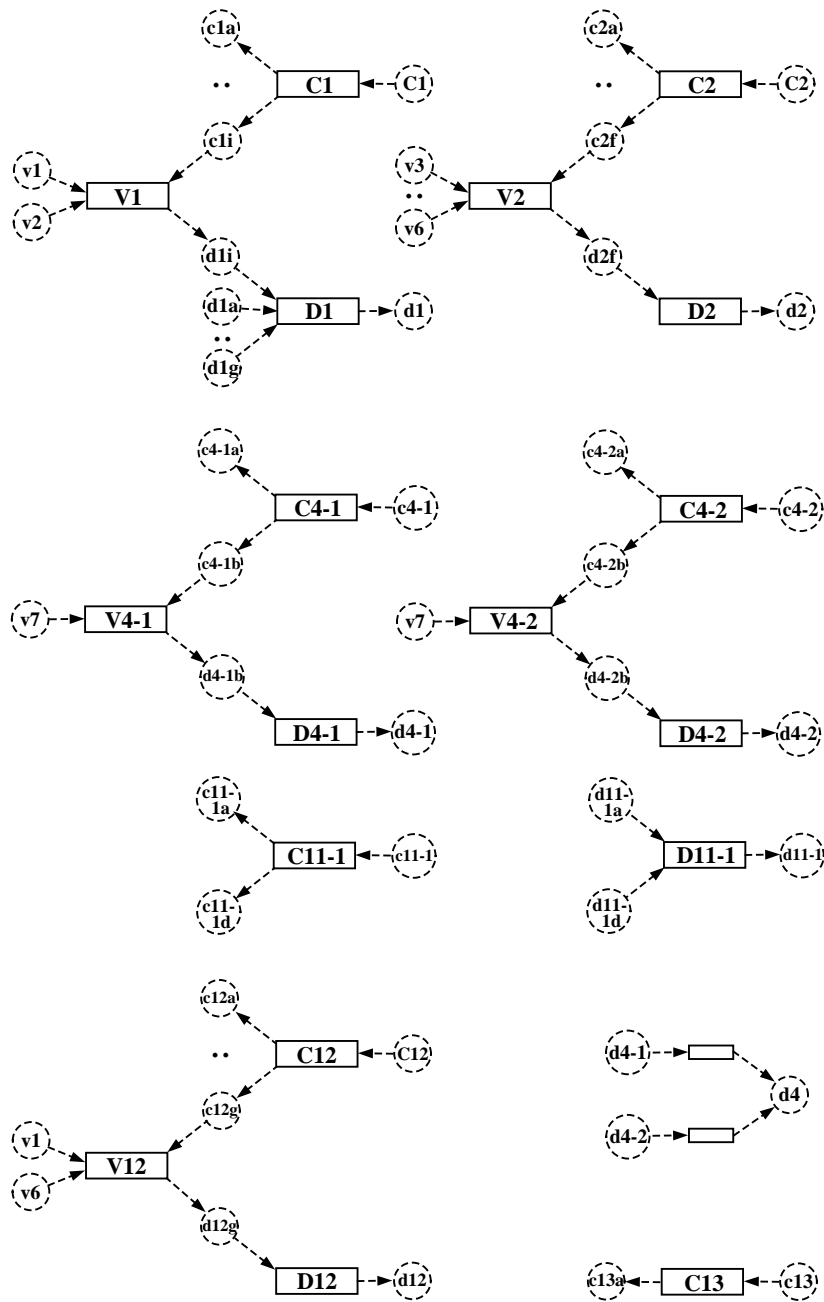


Figure 95: AES main local nets

These are split into two parts those numbered 1..4 and 11+. The gap in numbers ranging from 5 to 10 is used by the pipeline local nets.

Fig. 96 shows the local nets for the pipeline functions. The three pipeline stages from P1- to P3- are covered by the stages ranging from 5..8, 6..9 and 7..10 respectively. The local nets numbered from 5..8 at the top are used for processing the four lots of 32-bit data that pass through the sbbox.

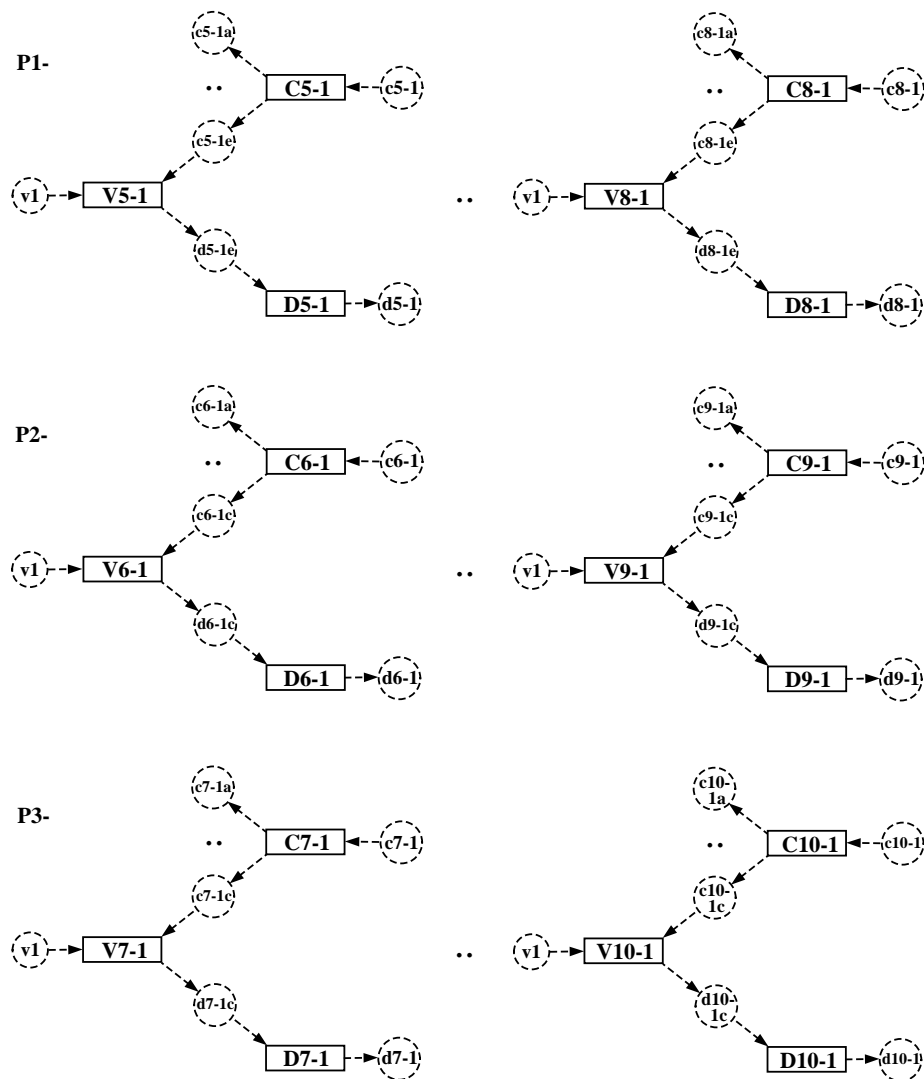
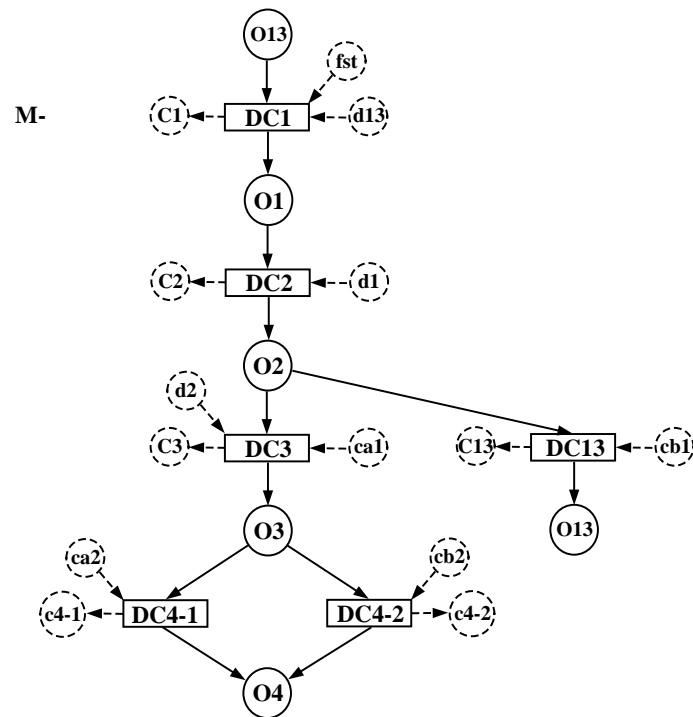


Figure 96: AES function pipeline local nets

Fig. 97 shows the main global control net. This is split in two parts with a gap appearing for the pipeline nets. Place 'O4' is used for connecting from the main nets to the pipeline nets.



Pipeline nets appear here ..

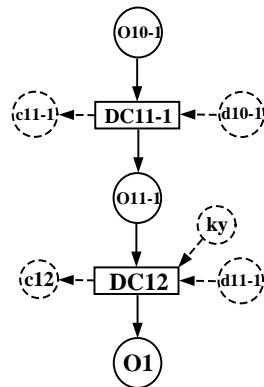


Figure 97: AES main global net

Fig. 98 shows the pipeline global control nets. The three pipestages appear from left to right. Within each pipestage there are four transition stages one for each 32 bit set of data.

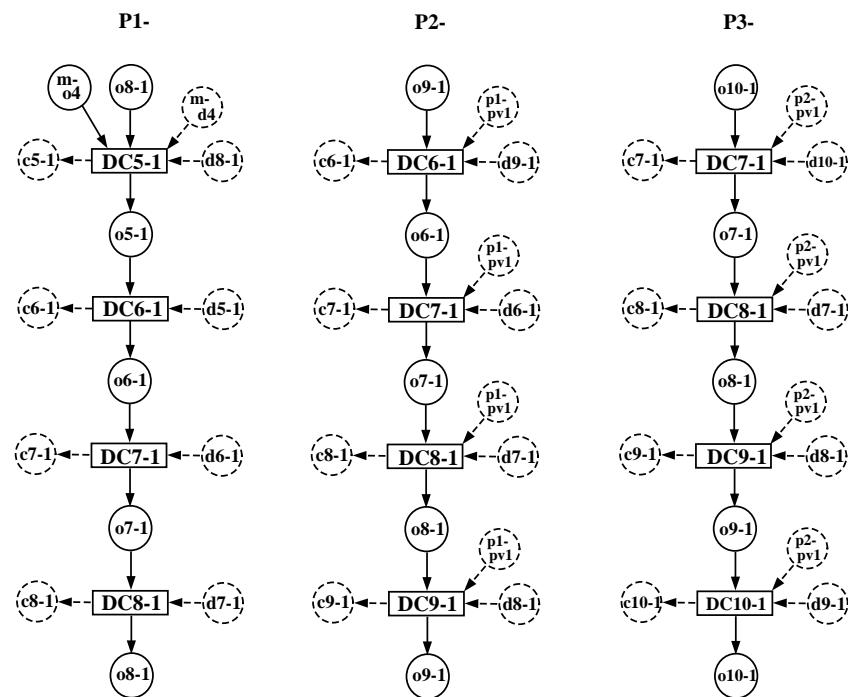


Figure 98: AES pipeline global nets

References

- [1] A. I. Davis, S. M. Nowick, "An Introduction to Asynchronous Circuit Design," The Encyclopaedia of Computer Science and Technology, Vol. 38, New York, Feb. 1998.
- [2] C. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming Language Tangram and its Translation into Handshake Circuits," In Proc. European Conference on Design Automation (EDAC), pp. 384-389, 1991.
- [3] A. Bardsley and D. Edwards, "Compiling the language Balsa to delay-insensitive hardware, Hardware Description Languages and their Applications (CHDL)," pages 89-91, 1997.
- [4] I. Blunno and L. Lavagno, "Automated synthesis of micro-pipelines from behavioral Verilog HDL," Proc. of IEEE Symp. on Adv. Res. in Async. Cir. and Syst. (ASYNC'2000), pp. 84-92.
- [5] The Petrify tool, <http://www.lsi.upc.es/~jordic/petrify/>
- [6] H. Jacobson, E. Brunvand, G. Gopalakrishnan and P. Kudva, "High-Level Asynchronous System Design using the ACK Framework," Proceedings ASYNC, pp. 2000.
- [7] V. Varshavsky et al, "Self-Timed Control of Concurrent Processes," Kluwer Academic Publishers, P.O. Box 17,3300 AA Dordrecht, The Netherlands, 1990 (Russian Edition: Nauka, Moscow, 1986).

- [8] P. Eles, K. Kuchcinski and Z. Peng, "System Synthesis with VHDL," Kluwer Academic Publishers, P.O. Box 17,3300 AA Dordrecht, The Netherlands, 1998.
- [9] F. Burns, D. Shang, A. Koelmans, and A. Yakovlev, "Synthesis of Asynchronous Data Paths and Controllers using PNs," Proceedings of 12th UK Asynchronous Forum, South Bank University, London, June 17-18, 2002.
- [10] D. Shang, F. Burns, A. Koelmans, and A. Yakovlev, "Asynchronous System Synthesis based on Direct Mapping using VHDL and Petri nets," IEE Proceedings: Computers and Digital Techniques, May, 200.
- [11] K. Jensen, Coloured Petri nets, "Basic Concepts, Analysis Methods and Practical use," Volume 1, Basic concepts. EATCS Monographs in Theoretical Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, 1997.
- [12] James L. Peterson, "Petri net Theory and the Modelling of Systems," Prentice-Hall, 1981.
- [13] A. Yakovlev, L. Gomes, and L. Lavagno, Editors, "Hardware Design and Petri Nets," Kluwer Academic Publishers, March 2000.
- [14] R. David, "Modular Design of Asynchronous Circuits Defined by Graphs," IEEE Transactions on Computers, Vol. 26(8), pp. 727-737, Aug. 1977.
- [15] D. Shang, F. Xia and A. Yakovlev, "Asynchronous Circuit Synthesis via Direct Translation," ISCAS 2002, IEEE International Symposium on Circuits and Systems, Scottsdale, Arizona. May 2002.
- [16] A. Bystrov, D. Sokolov, A. Yakovlev, A. Koelmans, "Balancing Power Signature in Secure Systems," Proc. 14th Asynchronous Forum, Newcastle, July 2003.
- [17] <http://www.icarus.com/eda/verilog/>
- [18] J. Kessels, K. van Berkel, R. Burgess, M. Ronken and F. Schalij, "An error decoder for the compact disc player as an example of VLSI programming," Technical report, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.
- [19] J. Daemen and V. Rijmen, "The design of Rijndael," The design of Rijndael, 2002.