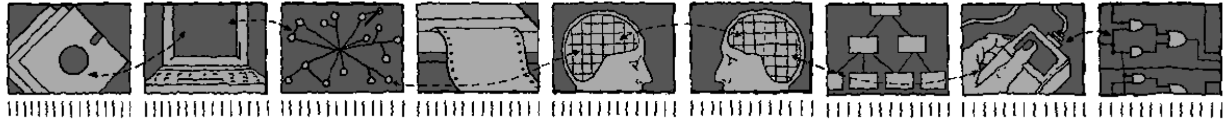
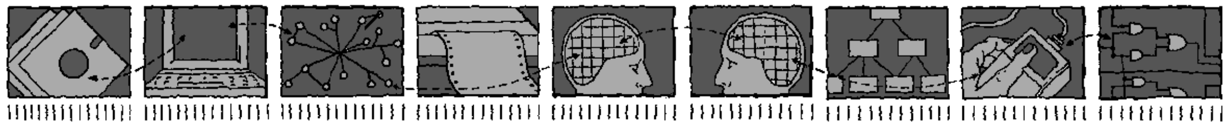


Department of Computing Science and Mathematics
University of Stirling



Extended DILL: Digital Logic in LOTOS

Ji He and Kenneth J. Turner

Technical Report CSM-142

November 1997

Department of Computing Science and Mathematics
University of Stirling

Extended DILL: Digital Logic in LOTOS

Ji He and Kenneth J. Turner

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email jih@cs.stir.ac.uk, kjt@cs.stir.ac.uk

Technical Report CSM-142

November 1997

Abstract

DILL (Digital Logic in LOTOS) is a language and an approach to specify digital logic in LOTOS; the initial version of DILL was developed in 1993 at Stirling. This technical report investigates further the possibilities for specifying and analysing digital systems in LOTOS. The new version of DILL contains more building blocks, including tri-state components and abstract descriptions. Strategies for specifying multi-bit signals are also supported in the extended DILL language. These extensions aim to provide a more flexible and powerful capability for specifying digital logic.

In the report, an example of designing a simple CPU is given to examine the new extensions and to give an overall feel for the DILL approach. The example indicates that DILL is suitable for specifying digital logic, not only at the gate level but also at an abstract level. Through the example we have gained more confidence in the suitability of DILL for specifying and analysing digital logic, especially for larger-scale digital circuits.

Keywords: Digital Logic, DILL (Digital Logic in LOTOS), Hardware Description Language, LOTOS (Language Of Temporal Ordering Specification)

Contents

1 Introduction **1**
1.1 Previous Work on DILL . . . 1
1.2 New Work on DILL . . . 2

2 Defining An Arbitrary State for Signals **3**
2.1 Arbitrary Signal State . . . 3
2.2 Impact on Models of Basic Gates . . . 4

3 Multi-Bit Signals and Buses **5**
3.1 Macros for Multi-Bit Wires And Signals . . . 5
3.2 LOTOS Data Type *BitArray* for Multi-Bit Signals . . . 6

4 Tri-State Components **7**

5 Specifying Digital Circuits at an Abstract Level **8**
5.1 Abstract Specifications . . . 8
5.2 Data Flow Style . . . 9
5.3 The Behaviour Style . . . 10
5.4 Specifying Circuits with Edge-Sensitive Signals . . . 11
5.5 Components with Non-Fixed Number of Signals . . . 12

6 Case Study: Designing a Sub-CPU in DILL **12**
6.1 Structure of a Typical CPU . . . 12
6.2 The Design Objectives of the Sub-CPU . . . 14
6.3 Behaviour Specification and Simulation of the Sub-CPU . . . 15
6.3.1 Behaviour Specification of the Sub-CPU . . . 15
6.3.2 Simulating the behaviour specification . . . 16
6.4 Structural Specification and Simulation of the Sub-CPU . . . 18
6.4.1 Structural Specification of the Sub-CPU . . . 18
6.4.2 Simulation of the Structural Specification . . . 19
6.5 Gate-Level Specification and Simulation . . . 19
6.6 Conclusion of the Case Study . . . 20

7 Conclusions and Future Work **20**
7.1 Conclusion . . . 20
7.2 Future work . . . 21

Acknowledgements **21**

References **21**

A LOTOS Syntax and *m4* Built-in Commands **23**

B The DILL Library **25**

C Gate-Level Component Specifications **32**
C.1 8-Bit Ripple Adder . . . 32
C.2 3-to-8 Line Decoder . . . 32
C.3 BCD-to-Decimal Decoder . . . 32
C.4 Excess-3-to Decimal Decoder . . . 34
C.5 Excess-3-Gray-to-Decimal Decoder . . . 35
C.6 1-Bit Comparator . . . 35
C.7 4-Bit Comparator . . . 36

C.8 8-Bit Comparator 36
C.9 4-Bit Binary Counter with Reset (74LS93) 36
C.10 Master Slave RS Flip-Flop with Preset and Preclear 37
C.11 Master Slave JK Flip-Flop With Preset and Preclear 38
C.12 Master Slave D Flip-Flop With Preset and Preclear 39
C.13 Edge-Triggered RS Flip-Flop 39
C.14 RS Latch with Preset and Preclear 39
C.15 4-Bit D Latch 40
C.16 4*4-Bit Memory with Read and Write (74LS170) 40
C.17 8-Bit Parity Checker/Generator 40
C.18 2-1 Multiplexer 42
C.19 8-bit 2-1 Multiplexer 42
C.20 8-bit 2-1 Multiplexer with Register Output 42
C.21 8-Bit Register 43
C.22 8-Bit Shift Register 43
C.23 4-Bit D-Type Register with Tri-State Output (74LS173) 43

D Signals in the Sub-CPU **46**
D.1 Signals in the Behaviour Specification 46
D.2 Signals in the Structural Specification 46

E Behaviour Specification of the Sub-CPU **48**

F Structural Specification of the Sub-CPU **55**

G Execution Path of An instruction Sequence of the Sub-CPU **59**
G.1 Execution Path for the Behaviour Specification 59
G.2 Execution Path for the Structural Specification 60

List of Figures

1 And-Inverter Circuit 1
2 Inverter-Inverter Circuit 3
3 Data Transfer through Bus 7
4 Tri-State Inverters and a Bus 8
5 RS latch 9
6 4-Bit Adder 13
7 D Flip-Flop 13
8 Block Diagram of a Simple CPU with Memory 13
9 The Structure of an Instruction 14
10 Timing of the Single-Cycle Sub-CPU 15
11 Block Diagram of the Sub-CPU 18
12 8-Bit Ripple Adder 32
13 3-to-8 Line Decoder 33
14 BCD-to-Decimal Decoder 34
15 1-Bit Comparator 36
16 4-Bit Comparator 36
17 4-Bit Binary Counter with Reset 37
18 Master-Slave RS Flip-Flop with Preset and Preclear 38
19 RS Latch with Preset and Preclear 39
20 4*4-Bit Memory with Read and Write 41
21 8-Bit Parity Checker/Generator 42
22 2-to-1 Multiplexer 42
23 4-Bit D-Type Register with Tri-State Output 45

List of Tables

1 Selected LOTOS Syntax 23
2 Selected *m4* Built-in Commands 24
3 Basic Logic Gates 25
4 Tri-State Components *(first part)* 26
5 Tri-State Components *(second part)* 27
6 Adders 27
7 Encoders and Decoders 27
8 Comparators 28
9 Counters 28
10 Flip-Flops 29
11 Latches 30
12 Memories 30
13 Parity Checkers/Generators 30
14 Multiplexers and Demultiplexers 30
15 Registers 31

1 Introduction

1.1 Previous Work on DILL

DILL (Digital Logic in LOTOS) is a language and an approach to specify digital logic in LOTOS. The inspiration of using LOTOS to specify digital system was pointed out in [9]. This original version of DILL includes:

- the necessary LOTOS data types for specification of hardware concepts
- an approach to modelling digital signals
- modelling basic logic gates
- building digital circuits by connecting the basic logic gates
- a library for the data types and some common digital components and circuit designs
- simulation capabilities, which were applied to the library

Digital signals are modelled as two-level voltages, expressed by constants of the LOTOS data type *Bit*. Constant 1 of *Bit* represents logic 1, constant 0 of *Bit* represents logic 0. There are also some extensions to the standard *Bit* type for logical operations on signals (*and*, *or*, *not*, ...). Note that DILL only deals with logic values and logic operations on physical signals. If DILL is used to describe a circuit that uses negative logic, the bit constant 1 represents a LOW voltage, and 0 represents a HIGH one.

Each basic logic gate (*Not*/*Inverter*, *And*, *Or* ...) is modelled as a LOTOS process. For example, *And3[Ip1,Ip2,Ip3,Op]* is a LOTOS specification of an *And* gate with 3 inputs. A LOTOS gate (e.g. *Ip1*) models a physical wire or pin. A LOTOS event (e.g. *Ip1!1*) models a signal change on the wire (here, from 0 to 1). Note that DILL describes only discrete signals and their changes (edges). Larger circuits can be built from basic logic gates using LOTOS parallel expressions. Thus an *And3* gate followed by an *Inverter* can be modelled as:

And3[Ip1,Ip2,Ip3,Op] |[Op]| Inverter[Op,IOp]

As illustrated in figure 1, connecting several wires and pins is modelled as synchronisation at LOTOS gates. In DILL we ignore the propagation delay of wires, If we have to take such delay into account in high speed circuits, we can add an additional component modelling delay in the appropriate place.

We always package the specification of a circuit into a LOTOS process in order to reuse it. When we do so, we give only the LOTOS gates for the inputs and outputs of the circuit, and hide all the other LOTOS gates. For example, if we want to reuse the circuit above, we simply write a new LOTOS process that can be used in building other larger circuits.

process And3Not[Ip1,Ip2,Ip3,IOp] : **noexit** :=
  hide Op **in**
    And3[Ip1,Ip2,Ip3,Op] |[Op]| Inverter[Op,IOp]
endproc (* And3Not *)

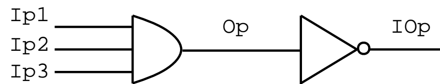


Figure 1: And-Inverter Circuit

The users of DILL are expected to be electronic engineers who may be not familiar with LOTOS at all. To be acceptable to the engineers, DILL provides a thin layer on top of LOTOS. This is implemented in a library called *dill.m4* – actually files of macros written in the *m4* [8] macro language. *dill.m4* also includes the pre-defined data types and some LOTOS specification of common small-scale digital circuits such as flip-flops and decoders. Of course, engineers can built their own component libraries to be reused by themselves or shared with others.

Here is the outline of a typical DILL circuit description:

```
divert(−1)
include(dill.m4)
include(mycomp.m4)
divert

circuit(‘MyCircuit[Ip1, ... IpN, Op1, ... OpN]’,‘
   Component1[...]
|[...]|
   Component2[...]
|[...]|
   ...
   ComponentN[...]

where
   process Component1 [...] : noexit :=
      ...
   endproc (* Component1 *)

   Component2_Decl
   ...
   ComponentN_Decl
’)
```

Actually this is an *m4* file, though it is not necessary for engineers to know much about *m4* other than the above. This circuit consists of several components connected in some way, among which *Component1* is a new component explicitly specified in the description, while *Component2* to *ComponentN* are pre-defined in *dill.m4* or *mycomp.m4*. This *m4* file can be translated into a LOTOS specification and then be simulated by LOTOS tools.

As pointed out in [4], the development of DILL has shown that LOTOS is applicable to the area of digital design, though it still has weaknesses such as simulation problems. One of the goals of using LOTOS for digital design was to exploit the well-developed range of LOTOS tools. But after using tools like LITE or HIPPO for simulation of digital circuits specified in DILL, we found that these tools handle some complex circuits poorly due to numerous internal events. One of our objectives in continuing the work on DILL is to find some method to reduce this complexity.

1.2 New Work on DILL

The extended DILL improves the initial version in the following ways:

- allowing an arbitrary unknown state for signals, specifically to deal with circuit initialisation problems
- modelling tri-state components, such as used in bus-based designs
- modelling multi-bit signals
- adding new building blocks to the DILL library, including gate-level and abstract descriptions

The objective of these extensions is to give DILL a more flexible and powerful ability to specify real-world digital systems. We also hope that by using abstract building blocks, the complexity of simulating large circuits can be reduced.

The following sections will describe the new extensions in detail. Section 2 introduces an arbitrary state for signals and explains its impact on the models of basic gates. Section 3 illustrates the two strategies for specifying multi-bit signals and buses. Section 4 deals with a new model for tri-state components. Section 5 covers the abstract specification of digital logic. These four sections include our main improvements on the DILL. In section 6 an example of designing a simple CPU is given to show how to use the extended DILL

to specify and simulate digital logic. Finally, section 7 concludes our work on extended DILL and proposes directions for future work.

Readers unfamiliar with LOTOS or *m4* may find the summaries in appendix A useful. Appendix B tabulates all the components in the DILL library. Appendix C gives the specification of the components that have been included in the extensions to DILL. Appendix D explains the signals used in the CPU example of section 6. Appendixes E and F give the behavioural and structural specifications of the CPU example. Appendix G shows typical simulation paths for these specifications.

2 Defining An Arbitrary State for Signals

2.1 Arbitrary Signal State

As noted in section 1.1, a signal in DILL is modelled as a two-level voltage, expressed by 0 and 1 in the LOTOS standard data type *Bit*. In the original DILL approach, all the input signals to a digital logic were assumed to be in the 0 state on start-up. This is clearly an approximation to the real world, in that the initial state of a digital system often cannot be predicted. The approximation also results in a temporal inconsistency in the digital logic. Suppose we wish to connect two Inverters in series as shown in figure 2. According to the previous version of DILL, the initial inputs of every components should be taken as 0, so wire A and wire B are both in the 0 state. The signal on wire C will thus initially be 1. This leads to a temporary inconsistency since the 0 state for A should lead to output 1 for B from the first inverter, and thus to output 0 for C from the second inverter.

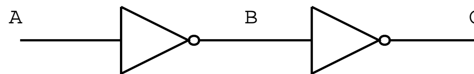


Figure 2: Inverter-Inverter Circuit

Although the 0 input assumption just influences the initial states of a circuit, it is still regarded as a shortcoming of the previous DILLIn order to model real digital circuits more precisely, we introduce a new state X for signals. We make X the initial state of every signal – both inputs and outputs. X can be interpreted as 'unknown', 'arbitrary' or 'do not care'. This removes inconsistencies such as that in figure 2.

A problem leading from the introduction of X is how to define boolean operations involving X. Can we decide the results of X *And* 0 or X *And* 1 for example? The answer determines how to write equations involving X in the *Bit* data type of LOTOS.

An important idea we have to make clear is that although we call X an 'arbitrary' state, we do not say that X is an arbitrary value between 0 and 1. DILL always assumes circuits are in a stable state, thus no signal could have an undecidable voltage, which is neither HIGH nor LOW. What we mean is that X is either 0 or 1, and there is a definite level on the wire, just we do not know its value and we do not care about it. It is now possible to evaluate bit operations using X. Because X *And* 0 is 0 whether X is 0 or 1, so X *And* 0 must be evaluated to 0. However, 1 *And* 1 is 1 but 0 *And* 1 is 0. The result of X *And* 1 is thus still unknown, and so must be evaluated to X. In the same way, we can define all the boolean operations on X:

```
forall B1, B2: Bit
  ofsort Bit
    not(X) = X;
    X and X = X;
    X and 0 = 0;
    X and 1 = X;
    B1 and B2 = B2 and B1;
    X or X = X;
    X or 0 = X;
    X or 1 = 1;
    B1 or B2 = B2 or B1;
```

Other operations can be derived from the *not*, *and*, *or* operations, just as for the usual two-value boolean operations:

a nand b = not(a and b);
a nor b = not(a or b);
a xor b = (a and not(b)) or (not(a) and b);

Normally we use X as the initial state of all signals, but it can also be used as the input state of a signal at any time. In the later case, X always means 'do not care'. Hardware engineers are familiar with this idea, and it makes DILL specification more flexible.

2.2 Impact on Models of Basic Gates

The models of basic logic gates need to be changed as a consequence of introducing the X. Perhaps surprisingly, the modified specifications are much simpler than the originals. In the previous version of DILL, every process modelling a basic logic gate had two auxiliary processes dealing respectively with initial outputs and later outputs. The new version only has one auxiliary process since initial states are handled by the bit operations. Take the one-input logic gate as example:

```
process Logic1 [Ip, Op] (BOp : BitOp) : noexit :=
    Logic1Aux [Ip, Op] (BOp, X of bit, X of bit)

    where

    process Logic1Aux [Ip, Op] (BOp : BitOp, BIn, BOut : Bit) : noexit :=
      Ip ? BInNew : Bit;
      Logic1Aux [Ip, Op] (BOp, BInNew, BOut)
    []
      (
        let BOutNew : Bit = Apply (BOp, BIn) in
          [(BOut eq X) and (BOutNew eq X)] ->
            Op ? BOutNew : Bit[BOutNew ne X];
            Logic1Aux [Ip, Op] (BOp, BIn, BOutNew)

        []
          [(BOutNew ne X) and (BOutNew ne BOut)] ->
            Op ! BOutNew;
            Logic1Aux [Ip, Op] (BOp, BIn, BOutNew)
      )
    endproc (* Logic1Aux *)
endproc (* Logic1 *)
```

The main points to note about this specification are:

- all the input and output signals are initialized to X
- if the current input values do not fully determine an output, the output can be an arbitrary but specific value (0 or 1)
- this is necessary to describe sequential circuits correctly (since they may have feedback)
- without this change, if an output of X were given it might be fed back to the input; the outputs might therefore stick at X (the equivalent of livelock) instead of settling on definite values. An example is our design of the master-slave JK flipflop (see [4] for circuit diagram), without such change, the output Q and Qbar can never settle down on 1 or 0 state no matter how J and K change.)

3 Multi-Bit Signals and Buses

In digital circuits, it is quite usual that some related signals are regarded as a whole. For example, in a computer system a group of 16 bit data signals could be seen as one signal carried by a data bus. We call such a signal consisting of more than one bit signals a *multi-bit signal*. Using the term from computer hardware, we refer to a wire carrying the multi-bit signal as a *bus*. Multi-bit signal and bus is an abstraction from the real physical world in which there exist only individual one-bit signals, each carried by a wire. Many hardware description languages provide data types to support this abstraction in order to simplify specification and also conform to what hardware engineers are used to. For example, VHDL (VLSI Hardware Design Language [3]) defines an 'array of bits' to represent multi-bit signals. The new version of DILL is also able to specify multi-bit wires (buses) and multi-bit signals, which are based on the different styles of specification.

3.1 Macros for Multi-Bit Wires And Signals

In the structural specification (see section 5), we have two macros *MWire* and *MComp* to specify multi-bit wires (buses) and components with such wires. As will be seen later, multi-bit wires are expanded to individual wires each carrying a one-bit signal. Similarly, multiple components are expanded to several individual components.

Macro *MWire(Count,'WireList')* is used for specifying a multi-bit wire of width 'Count'. For instance, *Mwire(8,'D')* represents an 8-bit data bus *D*. *MWire* is supported by the *m4* macro library, being ultimately translated into a LOTOS specification. For the example of *MWire(8,'D')*, the LOTOS specification will have gates *D7, D6, D5, D4, D3, D2, D1, D0*.

Macro *MComp(Count,ConnectingWire,'Component')* represents a group of components with similar architecture. Suppose that the LOTOS process *DFlipFlop[D,C,Q,Qbar]* models a D-type flip-flop. *MComp(4,C=,'DFlipFlop[D,C=,Q,Qbar]')* will correspond to a 4-bit D-type flip-flop with a clock signal common to its four one-bit flip-flops. Like *MWire*, macro *MComp* ends up generating LOTOS. The 4-bit D type flip-flop will appear as:

```
  DFlipFlop[D3,C,Q3,Qbar3]
|[C]|
  DFlipFlop[D2,C,Q2,Qbar2]
|[C]|
  DFlipFlop[D1,C,Q1,Qbar1]
|[C]|
  DFlipFlop[D0,C,Q0,Qbar0]
```

MComp and *MWire* are always used together. For the 4-bit D-type flip-flop, a DILL process representing it will be:

```
process DFlipFlop4 [MWire(4,D),C, MWire(4,'Q,Qbar')] : noexit :=
  MComp[4, C=,'DFlipFlop[D,C=,Q,Qbar]')
endproc
```

As we have seen, the combinational use of the two macros results in very compact DILL specifications. Although these two macros are just shorthands for the one-bit form of LOTOS specification, they provide a concise specification that incorporates conventional hardware ideas.

Besides the essential function illustrated above, *MWire* and *MComp* are designed for flexible interconnection of components in large circuits with repeated structure. Below we give some examples to explain how they are used.

MWire(Count,'WireList') produces multiple instances of wires:

MWire(4,'X,Y') yields *X3, Y3, X2, Y2, X1, Y1, X0, Y0*. *X* and *Y* can be regarded as 4-bit buses that are expanded into 4 one-bit wires.

If there is more than one bus in the bus list, single quotes must be put round the list. The index number after the wire names decreases by 1 from the index of MSB (Most Significant Bit) to LSB (least Significant Bit).

MWire(3,'A+,B,C++,D=')* yields *A3, B4, C4, D, A2, B2, C3, D, A1, B0, C2, D*.

MWire also allows some arithmetic operators to appear after a signal name. These refer to the current count, which ranges from the given multiplicity less one down to 0 (e.g. for 3 instances, the current count is sequentially set at 2, 1 and 0).

+ the index number is one more than the current count

- the index number is one less than the current count

* the index number is twice the current count

/ the index number is half the current count (rounded down to an integer)

= no index number is appended to wire name; the wire is a single wire. (The '=' means 'take as literally equal to the given name'.)

If necessary, the arithmetic operators may be repeated (e.g. *A++* for a double increment). A repeated '-' operator is unlikely to be useful since it could result in a negative index.

MComp(Count,ConnectingWire,'Component') produce multiple instances of a component process:

MComp(2,'And2[Ip1,Ip2,Op]') yields *And2[Ip11,Ip21,Op1]* ||| *And2[Ip10,Ip20,Op0]*.

The process (component) must be put in single quotes. The *MComp* actually takes three parameters: the multiplicity, the joining operator and the process. If the second parameter of the macro is omitted as here, there is no connection between each component and the ||| operator is used by default. An index number will be appended to each gate in the same process if not otherwise indicated (i.e. by '=').

MComp(4,'D, C=','DFlipFlop[D+,C=,Q,Qbar]') yields the following:

```
   DFlipFlop[D4,C, Q3,Qbar3]
|[D3,C]|
   DFlipFlop[D3,C, Q2,Qbar2]
|[D2,C]|
   DFlipFlop[D2,C,Q1,Qbar1]
|[D1,C]|
   DFlipFlop[D1,C,Q0,Qbar0]
```

If there is more than one connecting wire, the list must be given in single quotes. An arithmetic operator can also be appended to a wire name, with the same meaning as for the *MWire* Macro.

3.2 LOTOS Data Type *BitArray* for Multi-Bit Signals

For the behaviour style of specification (see section 5), DILL offers the data type *BitArray* to specify multi-bit signals. In such a case, some LOTOS gates are modeled as a group of wires (i.e. buses) carrying multi-bit signals. Because LOTOS gates in current standard LOTOS are not typed, we cannot distinguish these two kinds of gate from the syntax, however we can do so from the context.

The following are some basic operators of the *BitArray* type:

Bit is the constructor of the type

concatenates a bit array and a bit, or a bit and a bit array, to form a new bit array; for example, *Bit(0)#1#0#1* represents the 4-bit signal 0101

concatenates two bit arrays to form a new one; for example, *(Bit(0)#1#0#1) ## (Bit(1)#0#1#0)* forms the 8-bit signal 01011010

Length returns the width of a bit array

NatNum changes the binary value represented by a bit array to a natural number

+ adds the value of two bit arrays, or one bit array and a bit

not, and, nand, ... are logical operators

eq, ne, lt, ... are relational operators

. gets the value of a particular bit from a bit array

Set sets the value of a particular bit in a bit array

A multi-bit signal represented by *BitArray* can also be treated as a set of individual one-bit signals. Operators *Set* and '.' allow us to access any individual bit in a bit array. Thus data type *BitArray* can be used with both *n*-bit and one-bit wires.

We could continue to define a lot of operators of *BitArray* according to the needs of the circuits to be defined. For example, if we wished to define a BCD adder, we might want to define some new operators relating to BCD operations.

Introducing the type *BitArray* can result in more abstract specifications of digital systems, allowing us to ignore the details of how a logic design is implemented. It also makes it possible to use one process to model a group of components with the same behaviour except for the number of inputs and/or outputs. For instance *Reg_nBB[D,Ck,Q]* could represent a register of any data width.

4 Tri-State Components

In the discussion of previous sections, we referred only to two states of a signal 0 or 1. Although we introduced the X state, it just means either 0 or 1. For a tri-state component, its outputs are in one of three states: normal low-impedance Low, normal low-impedance High, and a high-impedance state. The advantage of tri-state components is that the outputs of several such components can be connected together without the worry of damage to the components. (The outputs of normal components cannot be connected since this would lead to a short circuit if one component were trying to output 0 and the other to output 1.) Tri-state output is especially useful when we want to transfer data among several components via a bus. As shown in figure 3, the registers A, B, C and D are connected to the register G through *Bus*, the outputs of the Decoder determine which register is allowed to transfer its data to the register G .

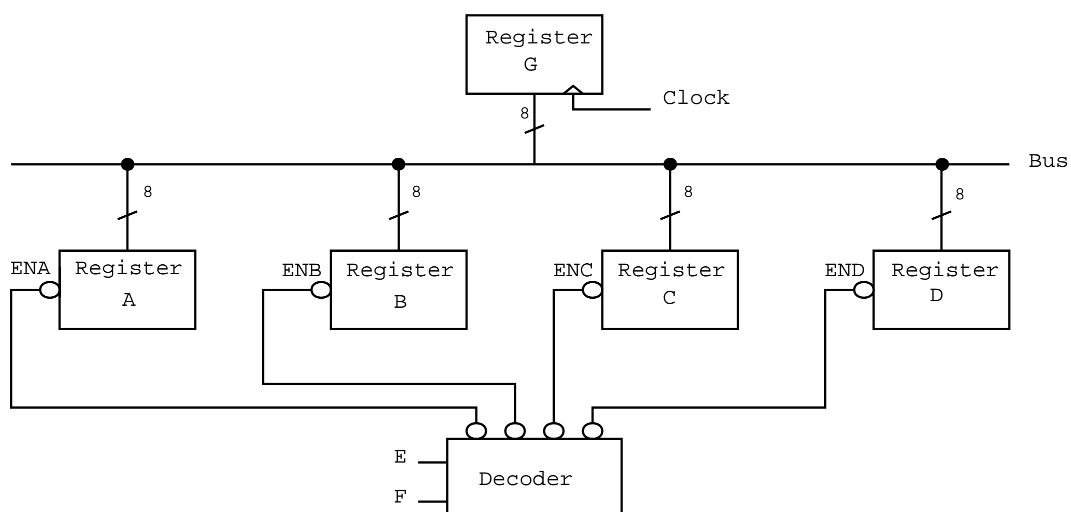


Figure 3: Data Transfer through Bus

Such kind of the bus structure is very common in digital design, so we decide to include tri-state components in the new version of DILL. Each tri-state gate corresponds to a basic logic gate but with tri-state output. On specifying the tri-state gates, we did not introduce a new bit value for the high impedance state, although this would appear to be the obvious solution. This approach would result in problems when several tri-state components were connected. For example, supposing three tri-state inverters were connected to a bus as shown in figure 4. In LOTOS this would be modelled as:

```
   Inverter_One[Ip1,En1,Bus]
|[Bus]|
   Inverter_One[Ip2,En2,Bus]
|[Bus]|
   Inverter_One[Ip3,En3,Bus]
```

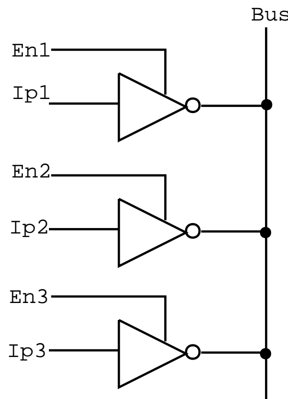


Figure 4: Tri-State Inverters and a Bus

Assume that two of the inverters are disabled (in the high impedance state) and one is in the 0 state. If we used a new bit value to indicate a high impedance 'output', then the event on LOTOS gate *Bus* could not synchronise. Thus no signal would appear on the bus. In reality, the bus would be in the 0 state.

We use a LOTOS event offer with '?' to mimic the behaviour of a tri-state component in the high impedance state. Such a state should allow one of the other outputs connecting to it to transfer data onto the bus. The state of the bus (and thus of all the connected outputs) would be determined by that data. A fragment of the LOTOS specification for tri-state output is:

```
[GIn eq not(Enable)] ->                              (* output is in high impedance state *)
   Op ? Other: Bit;
   Enable1Aux[Ip,G,Op](BOp,Enable,BIn,GIn,Other)
```

In our model of tri-state components, we do not provide a method to report the situation where either more than one tri-state components are enabled at the same time or none is enabled. In the former case, the electronic device may be damaged in the real world; a DILL description just ignores these enabled signals as if nothing had happened. In the later case, the bus will be in the high impedance state but in DILL, the bus will be in a random 0 or 1 state. It is therefore suggested to use the tri-state model carefully. Fortunately, as shown in figure 3, in real digital design the enable pins of tri-state components are always connected to the outputs of a decoder. This guarantees that at any time, one and only one enable signal is active. Our tri-state models are thus still quite useful.

5 Specifying Digital Circuits at an Abstract Level

5.1 Abstract Specifications

We have seen from previous sections how we construct digital circuits by connecting basic logic gates. In fact, all the components in the original DILL library except the basic logic gates were built in this way. We call this kind of specification style *structural* or *constructive*. It indicates how a digital circuit is designed.

Alternatively, we can use a more abstract style to specify our digital circuits. In such an abstract specification, we just specify what a component should do rather than how it is implemented. We have some good reasons to specify digital logic in this way:

- LOTOS supports specifications at a number of levels. It is quite natural for us to use it in higher-level specification.
- In some circumstances it is not necessary for us to know exactly how a component is built. For example, suppose we want to design an ALU (Arithmetic and Logic Unit) which interacts with a group of registers. We do not care about the construction of the register group at this stage; we only need to know its behaviour from the perspective of the outside environment. We should therefore specify the register group as a black box which just describes the relationship between inputs and outputs.
- There is well-developed hardware synthesis theory which provides the possibility to translate an abstract description into a concrete implementation automatically.
- The validation of digital logic designs in DILL is mainly by simulation at present. As we pointed out earlier, the current simulation tools handle complex circuits specified in the structural style poorly due to numerous internal events. One way to overcome this problem is to improve the LOTOS tools. The other way is through abstract specification. When we have confidence in our design for a component, we may replace it by its abstract counterpart and use the abstract component in larger circuits. So the simulation of the larger circuits will be only related to the abstract one. In this way, we avoid the internal events in the structural specification.

There are two styles for specifying components in an abstract way: *data flow* and *behavioural*. The abstract components in the DILL library are specified in either of these styles at the specifier's convenience. Several components have been specified in both styles to compare the differences between them.

5.2 Data Flow Style

In the data flow style, we use the LOTOS local definition operator **let**...**in** to describe how input signals (data) flow through the circuit to produce the outputs.

Suppose we are describing an RS latch (figure 5). The main part of its specification is:

```
    ...                                          (* inputs signal changed *)
  []
    let newdtQ : Bit = dtR nor dtQbar,
          newdtQbar: Bit = dtS nor dtQ in
      (...)                               (* output new value of output signal *)
```

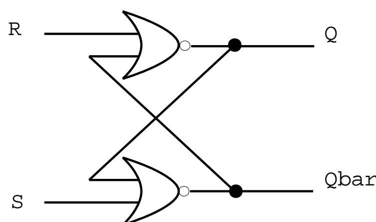


Figure 5: RS latch

A fragment of the specification for a 4-bit adder (figure 6) might be:

```
    ...                                          (* inputs signal changed *)
    let newdtS : BitArray = (dtA + dtB) + dtC0,
          newdtC4 : Bit = Carry(dtA,dtB) or Carry((dtA + dtB),dtC0) in
      (...)                               (* outputs changed according to new value *)
```

The difference between these two specifications in data flow style is that, in the case of the RS latch, the data flow expression can be changed to the connection of basic logic gates directly. However, in the case of the adder, there is no direct map between the data flow expression and its gate-level implementation. Sometimes, we call the specification like the RS latch above a *white box* specification. Although it is an abstract specification, it still needs some knowledge to implement it. In general, we need more or less constructive information to write the data flow specification. One extreme is mapping the structural design directly into a data flow expression. More often, we define operations on a LOTOS data type to aid the data flow specification. So the data flow style of specification relies heavily on the definition of such data types.

By the way, the data flow style should not seem strange because the basic logic gates are all defined in this way.

5.3 The Behaviour Style

The behaviour style of modelling produces a black box description of a component. It does not reflect how the circuit is built, but states what happens between the inputs and the outputs. What is inside the box is irrelevant. Again, consider the case of RS latch. The LOTOS specification is:

```
    ...                                                          (* inputs changed *)
  []
    [(dtR eq 1) and (dtS eq 0 of bit) and (dtQ ne 0 of bit)] ->           (*reset *)
      Q ! 0 of bit;
      RSLatchBBAux [R, S, Q, Qbar](dtR,dtS,0 of bit,dtQbar)
    []
      [(dtR eq 1) and (dtS eq 0 of bit) and (dtQbar ne 1)] ->
        Qbar! 1;
        RSLatchBBAux [R, S, Q, Qbar](dtR,dtS,dtQ,1)
    []                                                                (* set *)
      [(dtR eq 0 of bit) and (dtS eq 1) and (dtQ ne 1)] ->
        Q ! 1;
        RSLatchBBAux [R, S, Q, Qbar](dtR,dtS,1,dtQbar)
    []
      [(dtR eq 0 of bit) and (dtS eq 1) and (dtQbar ne 0 of bit)] ->
        Qbar ! 0 of bit;
        RSLatchBBAux [R, S, Q, Qbar](dtR,dtS,dtQ,0 of bit)
    []                                                          (* no change *)
      (* R,S return to their active level synchronously from the unstable state *)
      [(dtR eq 0 of bit) and (dtS eq 0 of bit) and
       (dtQ eq 0 of bit) and (dtQbar eq 0 of bit)] ->
        (
            i; Q ! 1;
            RSLatchBBAux [R,S,Q,Qbar] (dtR,dtS,1,dtQbar)
          []
            i; Qbar ! 1;
            RSLatchBBAux [R,S,Q,Qbar] (dtR,dtS,dtQ,1)
        )
      []                                                        (* R=1 S=1 *)
        [(dtR eq 1) and (dtS eq 1) and (dtQ ne 0 of bit)]->
          Q ! 0 of bit;
          RSLatchBBAux [R, S, Q, Qbar](dtR,dtS,0 of bit,dtQbar)
    []
      [(dtR eq 1) and (dtS eq 1) and (dtQbar ne 0 of bit)]->
        Qbar ! 0 of bit;
        RSLatchBBAux [R, S, Q, Qbar](dtR,dtS,dtQ,0 of bit)
```

The behaviour style for an RS latch gives the relationship between the inputs and the outputs. If *R=1, S=0* then the outputs are reset (*Q=0, Qbar=1*). If *R=0, S=1* then the outputs are set (*Q=1, Qbar=0*) and so on. In general, LOTOS guards are used to specify the relationship between inputs and outputs. Note the internal event **i** in the 'no change' state. If the two inputs *R, S* return from the inactive state (0) synchronously to the active state (1), the outputs cannot be decided by the environment; the result depends on the speed of the gates inside the circuit (i.e. there is a race condition). We use the internal event to model the non-deterministic behaviour.

One may argue that the behaviour style is not trivial even for a simple RS latch. This is true if we read the specifications of components in the DILL library. Almost every component specified in the behaviour style has a much longer description compared to its structural counterpart. But in fact, the idea of specifying a component in such style is very easy. What makes the description look so clumsy is aspects that are similar in all the component specifications. For example, a lot of guards are used to guarantee that a LOTOS event models a change of signal.

Reducing the complexity of the behaviour style specification is on the agenda of our future work on DILL. One possible solution maybe through adding more macros like *MWire* and *MComp* to give the 'core' part of the behaviour. The other, common part of the specification could be generated automatically during the expansion of the macros.

5.4 Specifying Circuits with Edge-Sensitive Signals

There are several components in the DILL library whose inputs or outputs are sensitive to signal *edges* rather than *values* of another signal. The most common of such signals are related to a clock, i.e. changes of these signals depend on the changing of a clock. Take the example of an edge-triggered D flip-flop (figure 7). The outputs *Q* and *Qbar* can be changed only at the negative-going transition of the clock.

The method of specifying such outputs is somewhat different from outputs that are not edge-sensitive. The method for specifying circuits without edge-sensitive signals takes the form:

```
   (* Input1 changed, go to next state *)
[]
   ...
[]
   (* InputI changed, go to next state *)
[]
   (* Output1 changed, go to next state *)
[]
   ...
[]
   (* OutputO changed, go to next state *)
```

In this model, every signal is independent of every other. But for circuits that are edge-sensitive, this model should be modified to guarantee all the outputs sensitive to the clock are correctly changed after the appropriate clock transition.

```
   (* Input1 changed, go to next state *)
[]
   ...
[]
   (* InputI changed, go to next state *)
[]
   (* Clock changed *)
   (
     (* Edge-sensitive Output1 changed, exit *)
   |||
     ...
   |||
     (* Edge-sensitive OutputE changed, exit *)
```

```
    )
[]
    (* Normal Output1 changed, go to next state *)
[]
    ...
[]
    (* Normal OutputN changed, go to next state *)
```

Writing specifications for edge-sensitive signals must always be done carefully to avoid deadlock, which results from incorrect guards after the clock transition.

5.5 Components with Non-Fixed Number of Signals

Sometimes we need a more flexible and more abstract model to represent a class of circuits. For example, perhaps we need an *n*-bit register to represent all registers no matter what the data widths are. Of course there is no such register in the real world. We abstract such a component in order to use it in every case where we need a register but do not care about the width, or where the width can be decided by the other components connected to it.

The LOTOS data type *BitArray* provides a convenient way to specify such a component. An abstract specification of a component with a one-bit signal can be easily changed to specification for *n* bits. What is needed is just changing the corresponding signal type (from *Bit* to *BitArray*) in the parameter list of the auxiliary process.

Such components cannot be connected with some of the other components since LOTOS gates in these components represent buses which will carry *N*-bit signals. However, an ordinary gate in other components can carry only one bit. *n*-bit components are always connected to provide a specification of digital logic at the system level.

6 Case Study: Designing a Sub-CPU in DILL

In this section we will give an example to show how to use DILL to design and simulate a digital system. Designing the digital system needs all the new features we added to DILL, such as multi-bit signal and bus, tri-state components, abstract components and so on. The example deals with designing part of a CPU (Central Processing Unit) of a simple computer system. We will refer to this part of the CPU as *sub-CPU* in the following text. Appendix D summarises the meaning of signals used in this example.

6.1 Structure of a Typical CPU

As we know, a typical CPU mainly consists of two parts: the control unit and the datapaths. The control unit provides all the necessary control signals that activate the various micro-operations in the datapaths to perform the specified data processing tasks. It also determines the sequence in which various actions are performed. Generally, the control unit consists of a program counter (PC) which provides the next instruction address to the instruction memory. An instruction memory contains several instructions to be executed in future steps, and a control logic which accepts an instruction and yields the control signals. The datapaths perform different kinds of data processing tasks according to the different instructions from the control unit. It is a combination of a set of registers (termed a *register file*), an arithmetic logic unit (ALU) and internal buses that provide pathways for the transfer of information between the registers, ALU and other computer components.

A CPU inevitably has connections with the other parts of a computer system, for example memory or I/O interface units. The CPU provides buses for transferring code, data and control information between itself and its connecting components. Figure 8 shows a block diagram for a simple CPU connecting with a memory. Note that every part in the figure, except instruction decoder and ALU, is controlled by the system clock, which we did not draw in the figure. Our example will be based on this structure.

Figure 6: 4-Bit Adder

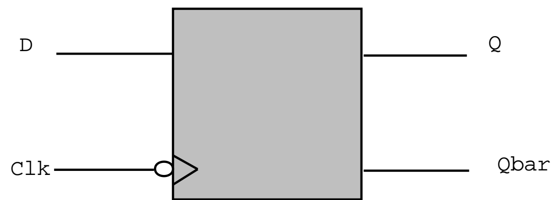


Figure 7: D Flip-Flop

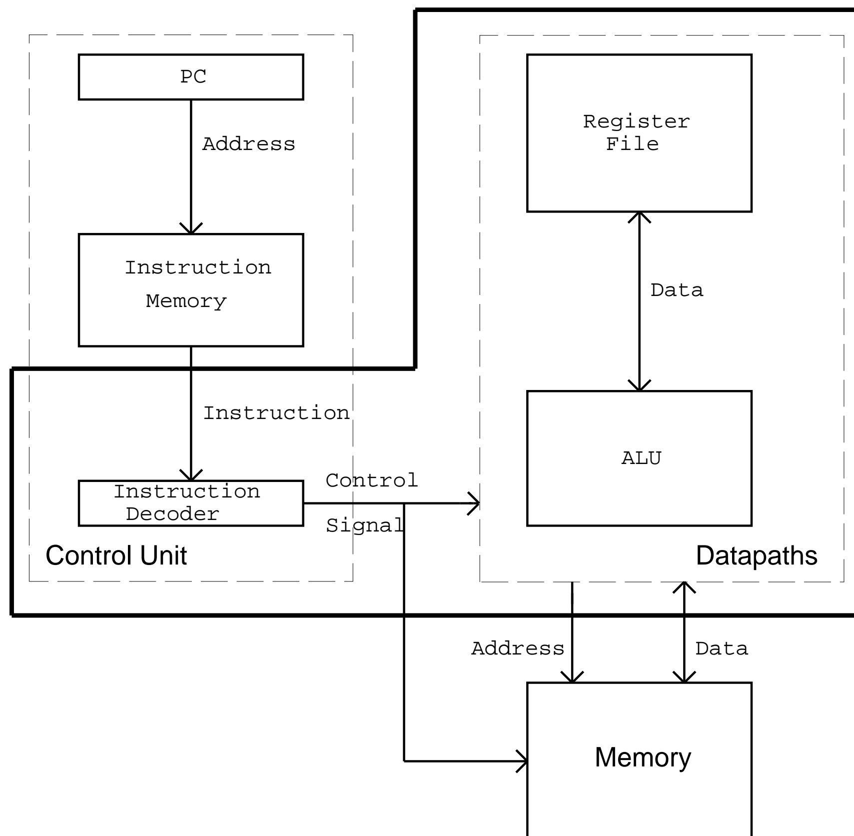


Figure 8: Block Diagram of a Simple CPU with Memory

6.2 The Design Objectives of the Sub-CPU

Our sub-CPU consists of the datapaths and the control logic of the control unit, which are included in the bold box in figure 8. This is a good start for us to define the objectives of designing the sub-CPU.

- The sub-CPU is a single-cycle CPU, which means fetching and executing an instruction should be finished in a single clock cycle.
- The sub-CPU is a part of a 4-bit computer, which means the widths of data registers, data buses and the memory units are all 4 bits.
- There are four data registers in the datapaths, named R0, R1, R2, R3, and two flag registers, named Flag0, Flag1.
- These registers load their inputs on a positive transition of the clock signal.
- The ALU performs two operations: addition of two values and comparison of two values.
- The instruction set of the sub-CPU includes 4 instructions:[1]
  - *Load DR, SA* Load the value residing in a memory unit whose address is specified by register SA into register DR.
  - *Store SA, SB* Store the value residing in register SB to a memory unit, whose address is specified in register SA.
  - *Add DR, SA, SB* Add two values residing in registers SA and SB,sending the sum to register DR and the carry to Flag1.
  - *Cmp SA,SB* Compare the two values in register SA, SB; if they are equal, the Flag0 is set to 1 otherwise set to 0.

  Note that the registers DR, SA, SB in the above instructions refer to any one of R3, R2, R1, R0.
- The structure of an instruction is shown in figure 9, in which:
  - DR is the Destination Register.
  - SA is the Source A register.
  - SB is the Source B register.
  - Opcode represents the operation to be executed.
    * 00 Compare two values in register SA and register SB.
    * 01 Store a value in register RB to a memory unit, whose address is in register SA.
    * 10 Load a value from a memory unit, whose address is in register SA, to register DR.
    * 11 Add the values in Source A register and Source B register and send the sum to the Destination Register DR.

Opcode	DR	SA	SB
7 6	5 4	3 2	1 0

Figure 9: The Structure of an Instruction

For an example the instruction *Add R3, R1, R2* will be in the following binary form: *11110110*. In DILL, it is represented by a value of type *BitArray bit(1)#1#1#1#0#1#1#0*.

If the number of operands is less than 3, the unused bits can be filled with either 1s or 0s. In the simulation discussed in this report, we feed each unused bit with 0. For instruction *Load R3,R2* the DILL representation will be *bit(1)#0#1#1#1#0#0#0*. Note the unused bits here are bit 1 and bit 0.

[1] The final instruction set will contain 5 instructions. See section 6.3.2.

- The connection between the sub-CPU and memory is achieved through address bus *BusAorMArr*, two data buses *DtIn* and *BusBorDtOut* and control signal *MW* (Memory Write).

Here, we will exploit a timing diagram shown in figure 10 and an *Add* instruction to help us understand better how our sub-CPU works. For an *Add* instruction *Add R3,R1,R0*, after the sub-CPU receives it, the control logic will produce control signals relating to the *Add* operation, then the values in register R1 and R0 are added. On the next positive transition of the clock signal, the sum is stored in register R3 and the carry is stored in Flag1.

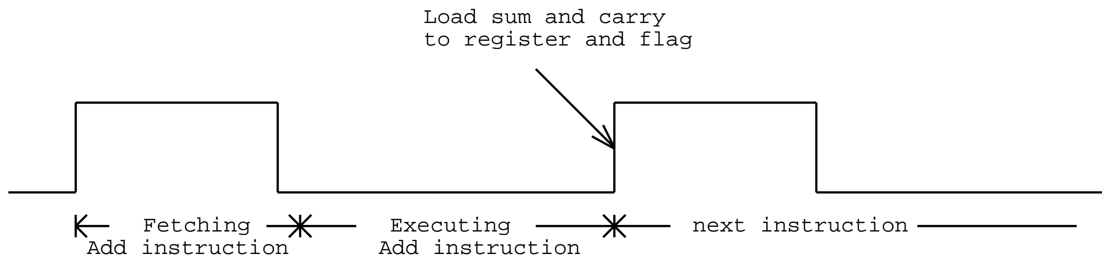


Figure 10: Timing of the Single-Cycle Sub-CPU

The rest of the section will explain how to use DILL to specify and simulate the design of the sub-CPU. We are going to use DILL at three levels, from high level behaviour specification to gate level structural specification. We will discuss them one by one.

6.3 Behaviour Specification and Simulation of the Sub-CPU

6.3.1 Behaviour Specification of the Sub-CPU

The objective of behaviour specification is to get a more precise specification than the description written in natural language.

As we have mentioned in section 5, a behaviour specification yields a black box of the system, with its inputs and outputs connecting with the outside environment. It just deals with how the system should behave rather than how to implement it. If we keep these two points in mind, writing a behaviour specification in DILL will not be too difficult.

Before we try to decide the inputs and outputs of the sub-CPU system, we review the block diagram of figure 8 and our design objectives in section 6.2. Obviously, the instruction (indicated as IR in the following text) which is offered by the instruction memory and the data from memory (indicated as *DtIn*) are inputs of the black box. The clock signal (indicated as *Clk*) is also an input because it determines when the register should load its value. With respect to sub-CPU's outputs, it is easy for us to find that the control signals to memory, memory write signal (*MW*), the memory address (*BusAorMarr*) and the data written to the memory (*BusBorDtout*) are counted as outputs of the black box. Besides these, the outputs of data registers and flag registers are also included in the outputs of the black box. Although these signals are not connected to any component outside the sub-CPU, their values are known to the outside world, say, the programmers of its assembly language .

So far, we have extracted the inputs and outputs of the sub-CPU. Thus the LOTOS process which represents the sub-CPU is like this:

SubCPU[IR, DtIn, Clk, R3, R2, R1, R0, Flag1, Flag0, BusAorMARR, BusBorDtout, MW]

We write the behaviour specification in the way of *inputs determining outputs*, which means for each input state, we need to specify its corresponding output state. Recall that section 5.4 says there are two kinds of output signals: one is edge-sensitive and the other is non edge-sensitive. In the sub-CPU system, *R3*, *R2*, *R1*, *R0*, *Flag1*, *Flag0* are sensitive to clock transitions. We take *BusAorMadd*, *BusBorDtOut*, *MW* as non edge-sensitive signals.

The outline of the specification is shown below, the complete DILL specification may be found in appendix E.

```
Signal on IR is changed; go to the next state
□
Signal on DtIn is changed; go to the next state
□
Signal on Clk is changed ;
(
  [ Clock signal is from 0 to 1 ] ->
    (
      [Is a Store Instruction] ->
          go to the next state
        (* store instruction does not yield any edge-sensitive signal*)
    □
      [Is a Load Instruction] ->
        Load the data on DtIn to the destination register;
        go to the next state
    □
      [Is an Add Instruction] ->
        (
          Load the sum of source A register and source B register to the
          destination register;
        |||
          Load the carry to the flag1 register
        )
        >>
          go to the next state
    □
      [Is a Cmp Instruction] ->
          Load the result of comparison of source A register and
          source B register to flag0 register;
          go to the next state
    )
  □
  [ Clock signal is not from 0 to 1] ->
      go to the next state
  )
□
[Is a Store instruction] ->
  Signal on MW changes to 1;
  go to the next state
□
[Is a Load or Add or Cmp instruction] ->
  Signal on MW changes to 0;
  go to the next state
□
  Data on BusAorMArr changes to the value of source A register;
  go to the next state
□
  Data on BusBorDtOut changes to the value of source B register;
  go to the next state
```

6.3.2 Simulating the behaviour specification

The simulation tool *hippo* is utilized in the simulation of the design of the sub-CPU. Inspecting the instruction set of the sub-CPU, it is evident that all the operations are based on registers, namely either the

actual operand or the address of a memory unit in which the actual operand resides is in a data register. So we choose a *Load* instruction as the first instruction to be simulated. For instruction *Load R0, R1*. Its binary form is 10001000, in DILL, represented by *bit(1)#0#0#0#1#0#0#0* of type *BitArray*.

The following is a fragment of the execution path of the instruction:

```
0     START
1     IR  !(((((((Bit(1) # 0) # 0) # 0) # 0) # 1) # 0) # 0)
2     MW  !0
3     Clk  !0
4     Clk  !1
```

The first step feeds the *Load* instruction to the IR. Steps 3 and 4 model the positive transition of clock signal. According to our specification, we intend that the *MW* signal should be changed to 0 because only the *Store* instruction can set it to 1. At the same time, the memory address which is contained in register R1 should appear on BusAorMArr. This simulation path shows that the signal on MW is just as expected but the memory address does not appear on the bus. This is because the value in R2 is an arbitrary value, so a definite memory unit can not be addressed.[2]

We know that in our instruction set, all values in registers can only come from memory. The problem is that we cannot load the memory address to register, because these addresses should also come from a memory unit. So after a couple of steps of simulation, we find ourselves in a dilemma. This indicates that there is a shortcoming in the original design of the instruction set. We need another instruction to load a value to register other than from memory.

We add a new load instruction which loads an immediate operand to a given register. For example, *Load R0,0000* loads 0 to register R0. The operand value comes from the instruction directly. Accordingly, the instruction set will contain five instructions and the instruction format is also changed a little. Now each instruction will have 9 bits and the highest bit indicates where the operand comes from, if it is 1, the operand will come from the instruction, otherwise from a memory unit. The other 8 bits have the same meaning as the previous definition.

After changing the specification to include the new instruction, we simulate the execution of the following instruction sequence:

```
Load  R0,0000
Load  R1,0011
Load  R2,1110
Load  R3,0011
Add   R0,R1,R2
Cmp   R0,R2
Cmp   R1,R3
Load  R3,0000
Store R3,R1
Load  R2,R3
```

The execution path for the first *Load* instruction looks like:

```
0     START
1     IR  !((((((((Bit(1) # 1) # 0) # 0) # 0) # 0) # 0) # 0) # 0)
2     MW  !0
3     Clk  !0
4     Clk  !1
5     R0  !(((Bit(0) # 0) # 0) # 0)
6     BusAorMArr  !(((Bit(0) # 0) # 0) # 0)
7     BusBorDtOut  !(((Bit(0) # 0) # 0) # 0)
```

The first step feeds the instruction *Load R0,0000* to IR. Since it is a *Load* instruction, the MW is changed to 0 (*MW !0* Memory write disable). After the positive transition of clock signal, the value in R0

[2]In the behaviour specification of the sub-CPU, we assume that if the value in register is an arbitrary number, the corresponding instruction will be neglected. See appendix E.

is changed to 0 (*R0 ! 0*). The data on BusAorMArr and BusBorDtOut are also changed according to the *Load* instruction. Because the changes appear between the execution of the current instruction and fetching the next instruction, they will not affect the execution of either the current instruction or the next one. By now, the first instruction has been executed correctly, we may continue to the second one.

The whole execution path of the above instruction sequence is in Appendix G.

6.4 Structural Specification and Simulation of the Sub-CPU

6.4.1 Structural Specification of the Sub-CPU

The structural specification specifies the structural design of the sub-CPU. We will not detail the design process but immediately give a block diagram (figure 11) here.

Figure 11: Block Diagram of the Sub-CPU

The structural specification is achieved by connecting components with the LOTOS parallel operator. We can first look for the components in the DILL library. If we cannot find an appropriate one, a new one should be written.

In the sub-CPU, the instruction decoder is a specific component in the design; we need to write a new process to specify it. It is written in a behaviour style, because we do not want to introduce internal events during the simulation. The decoders, multiplexers and demultiplexers in figure 11 can be easily found in DILL. Notice that all four registers are connected to both BusAorMArr and BusBorDtOut, so we need two

tri-state components, We also need 4-bit register components for data registers, and 1 bit register or flip-flop for flag registers of our sub-CPU. All these components are specified in the DILL library.

So what we need do is to write a new component for the instruction decoder in the behaviour style, and then connect all the components in an appropriate way.

The complete specification can be found in appendix F.

6.4.2 Simulation of the Structural Specification

The objective of simulating a structural specification is to look for design defects in a digital circuit. By comparing the execution paths of behaviour and structural specifications, we will know whether a design satisfies its behaviour requirements.

Actually, before we obtained the final version of the structural implementation of the sub-CPU, we had several designs which contained bugs, that were discovered during simulation. For example, originally, we used D flip-flops to represent flags. After executing the same instruction sequence as used in the simulation of the behaviour specification, we found the behaviour of flags are different. Here is the simulation path after execution the instruction *Load R0,0000*.

```
 0    START
 5    IR8  !1
 6    IR7  !1
 7    IR6  !0
 8    IR5  !0
 9    IR4  !0
10    IR3  !0
11    IR2  !0
12    IR1  !0
13    IR0  !0
18    MW  !0
52    Clk  !0
53    Clk  !1
54    Flag1  !0
55    R03  !0
56    R02  !0
57    R01  !0
58    R00  !0
59    BusAorMArr3  !0
60    BusAorMArr2  !0
61    BusAorMArr1  !0
62    BusAorMArr0  !0
63    BusBorDtOut3  !0
64    BusBorDtOut2  !0
65    BusBorDtOut1  !0
66    BusBorDtOut0  !0
```

Here we just list external events (corresponding to the inputs and outputs of the Sub-CPU) and omit all the internal events (corresponding to the internal connections between components), so the step numbers above are not continuous. Note that the change of flag1 in event 54 did not appear in the behaviour specification. Actually, in the structural design every instruction may influence the state of flags, but in the behaviour specification only *Add* and *Cmp* can change the flags. This suggests that the simple D flip-flop is not a suitable component to represent flags. Finally we replace it with a 1 bit register with load control.

After simulating the modified design and comparing its execution path (in appendix G)with that of the behaviour specification, we find they are same and satisfy ourselves that this time we get a proper design.

6.5 Gate-Level Specification and Simulation

The gate level specification is also a kind of structural specification, but replacing every component specified in an abstract way with a concrete one. It guarantees that a digital design can be implemented by connecting

ordinary basic logic gates, such as *AND, OR, NOT* gates.

In our design of the sub-CPU, except for the instruction decoder, the components we used in the previous level all have their gate level counterparts in our DILL library. Moreover, all these components have been simulated before they are put into the library. So what we should do at this level of specification is trivial: write a structural specification for the instruction decoder and simulate it, finally, replace all the abstract components with their gate level specification.

6.6 Conclusion of the Case Study

From above example, we may get some experience of designing digital logic in DILL. In general, the design process can be divided into several steps:

1. Specifying the design objectives of the digital logic in the behaviour style and simulating it. In this step, we aim at getting a precise specification of the digital system, and finding possible defects in the design requirements.
2. Specifying all the components which are selected to build the digital system in an abstract style and simulating them. This step may be omitted if these components are already in the DILL library.
3. Specifying the digital system in the structural style, this is achieved by connecting the components specified in the second step in an appropriate way. Simulating the specification to see if it satisfies the design objectives of the first step.
4. Specifying the components selected in the second step at the gate level and simulating them. If these components are already defined by DILL, the step may be omitted.
5. Replacing all the abstract components in the structural specification with their gate level components to get a gate level implementation.

7 Conclusions and Future Work

7.1 Conclusion

In this technical report we have described the improved version of DILL. The main contents of the new version can be summarised as:

- The necessary LOTOS data types for hardware specification are provided:
  - standard LOTOS data types and extensions for modelling one-bit signals
  - *BitArray* for modelling multi-bit signals
- Digital signals are modelled as two-level voltages, but specified using three signal states: 0 for logic 0, 1 for logic 1, and X for either 0 or 1.
- LOTOS gates use either one-bit wires or *n*-bit buses.
- The model of basic logic gates has been changed slightly from the original DILL because of the introduction of the X initial parameter. The new model has a simpler form than the old one.
- New *m4* macros have been defined for multi-bit wires.
- Tri-state components are now available in the library.
- Larger circuits can be specified in three styles:
  - structural style, connecting simpler components by LOTOS parallel expressions

– data flow style, using LOTOS local definitions to specify the relationship between inputs and outputs
– behaviour style, specifying the behavioural properties of a component

Through the work of the previous DILL version, we have acquired some experience of using LOTOS for gate-level specification. We have now specified some digital components at an abstract level, and find LOTOS also quite suitable for higher-level specification. Through the case study of the sub-CPU, we have gained more confidence in the suitability of LOTOS for specifying and analysing digital logic, especially for larger-scale digital circuits.

7.2 Future work

We plan future work on DILL to include the following:

- A more friendly interface for the user will be designed. As we mentioned before, there is only a thin layer above LOTOS to support the use of DILL. There is still a lot of work to be done on developing a better interface.
- A simpler approach for behaviour specification of digital circuits will be investigated. As seen in the case study of designing the sub-CPU, the behaviour specification has much longer codes than the structural one. Writing such specifications would be somewhat difficult for hardware engineers.
- The simulation tools ought to be improved so that internal events are transparent to DILL users. An ideal tool should have the capability to display the outputs of a digital circuit soon after its inputs are given.
- We will work on a model for specifying timing properties of digital logic. The current LOTOS standard has no time capabilities, and there are very limited tools to support timed specification in standard LOTOS. DILL currently refers only to the functional aspects of digital logic. But in future work on DILL, we hope to investigate timing properties because they are important in many logic designs.
- Equivalence checking between different levels of specification should also be taken into account in future work. As seen in our case study of the sub-CPU, we have not *proved* that the structural specification of the design satisfies the behaviour requirements.

Acknowledgements

Ji He gratefully acknowledges financial support for this work from the Sino-British Fellowship Trust and the University of Stirling. Mr. Frank R. Kelly kindly drew the diagrams in appendix C. Mr. Ian R. Wilson helped by carefully reading a draft of the report.

References

[1] George A.McCaskill and George J. Milne. Hardware description and verification using the circal-system. Technical Report HDV-24-92, Deparment of Computer Science,University of Strathclyde, UK, June 1992.

[2] G.J.Milne. Modeling digitla hardware in a process algebra. Technical Report CIS-95-010, University of South Australia, School of Computer and Information Science, May 1995.

[3] IEEE. *VLSI Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1992.

[4] Richard O. Sinnott. The development in LOTOS of digital logic specifications. Master's thesis, Department of Computing Science and Mathematics, University of Stirling, UK, March 1993.

[5] Kenneth J. Turner. LOTOS – A practical Formal Description Technique for OSI. In *International Open Systems 87*, volume 1, pages 265–279. Online Publications, London, March 1987.

[6] Kenneth J. Turner. An engineering approach to formal methods. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 357–380. North-Holland, Amsterdam, Netherlands, June 1993. Invited paper.

[7] Kenneth J. Turner, editor. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS and SDL.* Wiley, New York, January 1993.

[8] Kenneth J. Turner. Exploiting the *m4* macro language. Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, UK, September 1994.

[9] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

A LOTOS Syntax and *m4* Built-in Commands

This appendix gives selected LOTOS syntax in table 1 and *m4* built-in commands in table 2.

<table>
<tr><th>Notation</th><th>Meaning</th></tr>
<tr><td>(* text *)</td><td>a comment</td></tr>
<tr><td>stop</td><td>a behaviour that does nothing (no further action)</td></tr>
<tr><td>exit</td><td>a behaviour that immediately terminates successfully</td></tr>
<tr><td>exit (results)</td><td>successful termination with result values</td></tr>
<tr><td>gate</td><td>a ‘port’ at which event offers may synchronise</td></tr>
<tr><td>gate ! value</td><td>an offer to synchronise on a given value</td></tr>
<tr><td>gate ? variable : sort</td><td>an offer to synchronise on any value of the given sort, binding the actual value to the given variable name</td></tr>
<tr><td>gate ! ... ? ... [predicate]</td><td>an event offer with a predicate on values synchronised</td></tr>
<tr><td>process name [gates] (parameters) :
noexit := behaviour</td><td>a named process abstraction with given gates and value parameters, but no termination (e.g. it repeats indefinitely)</td></tr>
<tr><td>process name [gates] (parameters) :
exit (results) := behaviour</td><td>a process that terminates successfully with the given result sorts</td></tr>
<tr><td>name [gates] (parameters)</td><td>an instantiation of a named process</td></tr>
<tr><td>offer ; behaviour</td><td>prefixes an event offer to some behaviour (‘followed by’)</td></tr>
<tr><td>[guard] $\rightarrow$ behaviour</td><td>offers behaviour only if the guard condition is satisfied (‘if’)</td></tr>
<tr><td>behaviour1 [] behaviour2</td><td>offers a choice between two behaviours (‘or’)</td></tr>
<tr><td>behaviour1 $\gg$ behaviour2</td><td>allows the second behaviour to occur if the first behaviour terminates successfully (‘enables’)</td></tr>
<tr><td>exit (results) $\gg$
accept declarations in behaviour</td><td>successful termination with export of result values</td></tr>
<tr><td>behaviour1 $\rhd$ behaviour2</td><td>allows the second behaviour to disrupt the first behaviour unless this terminates successfully first (‘disabled by’)</td></tr>
<tr><td>behaviour1 || behaviour2</td><td>allows two behaviours to run in parallel, but fully synchronised on their events (‘synchronised with’)</td></tr>
<tr><td>behaviour1 ||| behaviour2</td><td>allows two behaviours to run in parallel, but with independent occurrence of their events (‘interleaved with’)</td></tr>
<tr><td>behaviour1 |[gates]| behaviour2</td><td>allows two behaviours to run in parallel, synchronising on all events at the given gates (‘synchronised on gates with’)</td></tr>
</table>

Table 1: Selected LOTOS Syntax

Build-in Command	Description
#...	ignore comment text up to and including the new line
changequote(*left*,*right*)	define quote characters as *left* and *right*(instead of '...')
define(*name*,*text*)	define macro *name* to be *text*
divert(*stream_number*)	divert future output to stream *stream_number* (1 to 99), the default of 0 is standard output
divnum(*stream_number*)	expand to currently active diversion number
dnl	delete up to newline, generally to prevent extra white space in macro
eval(*expression*)	expand to result of numeric expression; the arithmetic and logical operators are essentially those of C
ifdef(*'name',def_text,undef_text*)	if macro *name* (quoted to prevent expansion) is defined, expand to *def_text* otherwise *undef_next*
ifelse(*text1,text2,eq_text,ne_text*)	If *text1* and *text2* are equal strings,expand to *eq_text* otherwise *ne_text*; conditions may be repeated
include(*file_name*)	expand to contents of *file_name*
incr(*number*)	expand to *number*+1
index(*string1,string2*)	expand to position in *string1* where *string2* occurs (0 is start,-1 means not found)
len(*string*)	expand to length of *string*
substr(*string,position,number*)	expand to *string* from *position* (0 is start) for *number* characters (default to end of string)
translit(*string,from,to*)	expand to *string* with *from* characters replaced by corresponding *to* characters (or deleted if there is no corresponding)
undefine(*'name'*)	delete definition of *name* (quoted to prevent expansion)
undivert(*stream_number,...*)	retrieve text from diversion *stream_number* (all diversions as default) and append to current diversion (usually standard output)

Table 2: Selected *m4* Built-in Commands

B The DILL Library

Tables 3 to 15 summarise the components available in the DILL library. In the tables, 'WB' in the name of components refers to a White Box (extensional) style of specification, while 'BB' means a Black Box (intensional) style. In all other cases, an explicit structural style has been followed. The differences among these three styles are discussed in section 5.

Library Component	**Meaning**
And2[Ip1,Ip2,Op]	2-input *And*
And3[Ip1,Ip2,Ip3,Op]	3-input *And*
And4[Ip1,Ip2,Ip3,Ip4,Op]	4-input *And*
And8[MWire(8,Ip),Op]	8-input *And*
Inverter[Ip,Op]	*Not*, Inverter
Nand2[Ip1,Ip2,Op]	2-input *Not And*
Nand3[Ip1,Ip2,Ip3,Op]	3-input *Not And*
Nand4[Ip1,Ip2,Ip3,Ip4,Op]	4-input *Not And*
Nand8[MWire(8,Ip),Op]	8-input *Not And*
Nor2[Ip1,Ip2,Op]	2-input *Not Or*
Nor3[Ip1,Ip2,Ip3,Op]	3-input *Not Or*
Nor4[Ip1,Ip2,Ip3,Ip4,Op]	4-input *Not Or*
Nor8[MWire(8,Ip),Op]	8-input *Not Or*
One[S]	Source of logic 1
Or2[Ip1,Ip2,Op]	2-input *Or*
Or3[Ip1,Ip2,Ip3,Op]	3-input *Or*
Or4[Ip1,Ip2,Ip3,Ip4,Op]	4-input *Or*
Or8[MWire(8,Ip),Op]	8-input *Or*
Repeater[Ip,Op]	*Same*, Repeater, Delay
Sink[S]	Absorb signal
Xnor2[Ip1,Ip2,Op]	2-input *Exclusive Nor*
Xnor3[Ip1,Ip2,Ip3,Op]	3-input *Exclusive Nor*
Xnor4[Ip1,Ip2,Ip3,Ip4,Op]	4-input *Exclusive Nor*
Xnor8[MWire(8,Ip),Op]	8-input *Exclusive Nor*
Xor2[Ip1,Ip2,Op]	2-input *Exclusive Or*
Xor3[Ip1,Ip2,Ip3,Op]	3-input *Exclusive Or*
Xor4[Ip1,Ip2,Ip3,Ip4,Op]	4-input *Exclusive Or*
Xor8[MWire(8,Ip),Op]	8-input *Exclusive Or*
Zero[S]	Source of logic 0

Table 3: Basic Logic Gates

Library Component	**Meaning**
And2_One[Ip1,Ip2,G,Op]	2-input *And*, 1 active enable, tri-state output
And2_Zero[Ip1,Ip2,G,Op]	2-input *And*, 0 active enable, tri-state output
And3_One[Ip1,Ip2,Ip3,G,Op]	3-input *And*, 1 active enable, tri-state output
And3_Zero[Ip1,Ip2,Ip3,G,Op]	3-input *And*, 0 active enable, tri-state output
And4_One[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *And*, 1 active enable, tri-state output
And4_Zero[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *And*, 0 active enable, tri-state output
And8_One[MWire(8,Ip),G,Op]	8-input *And*, 1 active enable, tri-state output
And8_Zero[MWire(8,Ip),G,Op]	8-input *And*, 0 active enable, tri-state output
Inverter_One[Ip,G,Op]	1-bit Inverter, 1 active enable, tri-state output
Inverter_One_4[MWire(4,Inp),En,MWire(4,Oup)]	4-bit Inverter, 1 active enable, tri-state output
Inverter_Zero[Ip,G,Op]	1-bit Inverter, 0 active enable, tri-state output
Inverter_One_8[MWire(8,Inp),En,MWire(8,Oup)]	8-bit Inverter, 1 active enable, tri-state output
Inverter_Zero_4[MWire(4,Inp),En,MWire(4,Oup)]	4-bit Inverter, 0 active enable, tri-state output
Inverter_Zero_8[MWire(8,Inp),En,MWire(8,Oup)]	8-bit Inverter, 0 active enable, tri-state output
Nand2_One[Ip1,Ip2,G,Op]	2-input *Nand*, 1 active enable, tri-state output
Nand2_Zero[Ip1,Ip2,G,Op]	2-input *Nand*, 0 active enable, tri-state output
Nand3_One[Ip1,Ip2,Ip3,G,Op]	3-input *Nand*, 1 active enable, tri-state output
Nand3_Zero[Ip1,Ip2,Ip3,G,Op]	3-input *Nand*, 0 active enable, tri-state output
Nand4_One[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Nand*, 1 active enable, tri-state output
Nand4_Zero[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Nand*, 0 active enable, tri-state output
Nand8_One[MWire(8,Ip),G,Op]	8-input *Nand*, 1 active enable, tri-state output
Nand8_Zero[MWire(8,Ip),G,Op]	8-input *Nand*, 0 active enable, tri-state output
Nor2_One[Ip1,Ip2,G,Op]	2-input *Nor*, 1 active enable, tri-state output
Nor2_Zero[Ip1,Ip2,G,Op]	2-input *Nor*, 0 active enable, tri-state output
Nor3_One[Ip1,Ip2,Ip3,G,Op]	3-input *Nor*, 1 active enable, tri-state output
Nor3_Zero[Ip1,Ip2,Ip3,G,Op]	3-input *Nor*, 0 active enable, tri-state output
Nor4_One[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Nor*, 1 active enable, tri-state output
Nor4_Zero[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Nor*, 0 active enable, tri-state output
Nor8_One[MWire(8,Ip),G,Op]	8-input *Nor*, 1 active enable, tri-state output
Nor8_Zero[MWire(8,Ip),G,Op]	8-input *Nor*, 0 active enable, tri-state output
Or2_One[Ip1,Ip2,G,Op]	2-input *Or*, 1 active enable, tri-state output
Or2_Zero[Ip1,Ip2,G,Op]	2-input *Or*, 0 active enable, tri-state output
Or3_One[Ip1,Ip2,Ip3,G,Op]	3-input *Or*, 1 active enable, tri-state output
Or3_Zero[Ip1,Ip2,Ip3,G,Op]	3-input *Or*, 0 active enable, tri-state output
Or4_One[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Or*, 1 active enable, tri-state output
Or4_Zero[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Or*, 0 active enable, tri-state output
Or8_One[MWire(8,Ip),G,Op]	8-input *Or*, 1 active enable, tri-state output
Or8_Zero[MWire(8,Ip),G,Op]	8-input *Or*, 0 active enable, tri-state output
Repeater_One[Ip,G,Op]	1-bit Repeater, 1 active enable, tri-state output
Repeater_One_4[MWire(4,Ip),En,MWire(4,Op)]	4-bit Repeater, 1 active enable, tri-state output
Repeater_One_8[MWire(8,Ip),En,MWire(8,Op)]	8-bit Repeater, 1 active enable, tri-state output
Repeater_Zero[Ip,G,Op]	1-bit Repeater, 0 active enable, tri-state output
Repeater_Zero_4[MWire(4,Ip),En,MWire(4,Op)]	4-bit Repeater,0 active enable, tri-state output
Repeater_Zero_8[MWire(8,Ip),En,MWire(8,Op)]	8-bit Repeater, 0 active enable, tri-state output

Table 4: Tri-State Components *(first part)*

Library Component	Meaning
Xnor2_One[Ip1,Ip2,G,Op]	2-input *Xnor*, 1 active enable, tri-state output
Xnor2_Zero[Ip1,Ip2,G,Op]	2-input *Xnor*, 0 active enable, tri-state output
Xnor3_One[Ip1,Ip2,Ip3,G,Op]	3-input *Xnor*, 1 active enable, tri-state output
Xnor3_Zero[Ip1,Ip2,Ip3,G,Op]	3-input *Xnor*, 0 active enable, tri-state output
Xnor4_One[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Xnor*, 1 active enable, tri-state output
Xnor4_Zero[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Xnor*, 0 active enable, tri-state output
Xnor8_One[MWire(8,Ip),G,Op]	8-input *Xnor*, 1 active enable, tri-state output
Xnor8_Zero[MWire(8,Ip),G,Op]	8-input *Xnor*, 1 active enable, tri-state output
Xor2_One[Ip1,Ip2,G,Op]	2-input *Xor*, 1 active enable, tri-state output
Xor2_Zero[Ip1,Ip2,G,Op]	2-input *Xor*, 0 active enable, tri-state output
Xor3_One[Ip1,Ip2,Ip3,G,Op]	3-input *Xor*, 1 active enable, tri-state output
Xor3_Zero[Ip1,Ip2,Ip3,G,Op]	3-input *Xor*, 0 active enable, tri-state output
Xor4_One[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Xor*, 1 active enable, tri-state output
Xor4_Zero[Ip1,Ip2,Ip3,Ip4,G,Op]	4-input *Xor*, 0 active enable, tri-state output
Xor8_One[MWire(8,Ip),G,Op]	8-input *Xor*, 1 active enable, tri-state output
Xor8_Zero[MWire(8,Ip),G,Op]	8-input *Xor*, 1 active enable, tri-state output

Table 5: Tri-State Components *(second part)*

Library Component	Meaning
Adder2_BB[A1, A0, B1, B0, S1, S0, Cout]	2-bit adder
Adder4_BB[MWire(4,‘A,B’),C0,MWire(4,S),C4]	4-bit parallel adder
AdderN_BB[A,B,S,Cout]	*n*-bit adder
HalfAdder[A,B,S,C]	Half adder
HalfAdder_BB[A,B,S,C]	Half adder
FullAdder[A,B,Cin,S,Cout]	Full adder
FullAdder_BB[A,B,Cin,S,Cout]	Full adder
RippleAdder2[A1, A0, B1, B0, S1, S0, Cout]	2-bit ripple-through adder
RippleAdder4[MWire(4,‘A,B’),C0,MWire(4,S),C4]	4-bit ripple-through adder
RippleAdder8 [MWire(8,‘A,B’),C0, MWire(8,S), C8]	8-bit ripple-through adder

Table 6: Adders

Library Component	Meaning
BCDtoDec[MWire(4,D),MWire(10,Y)]	BCD-to-Decimal decoder
BCDtoDec_WB_One[MWire(4,D),MWire(10,Y)]	BCD-to-Decimal decoder, outputs 1 active
BCDtoDec_WB_Zero[MWire(4,D),MWire(10,Y)]	BCD-to-Decimal decoder, outputs 0 active
Decoder2[D1,D0,Q3,Q2,Q1,Q0]	2-to-4 line decoder
Decoder2_WB_One[D1,D0,Q3,Q2,Q1,Q0]	2-to-4 line decoder, outputs 1 active
Decoder2_WB_Zero[D1,D0,Q3,Q2,Q1,Q0]	2-to-4 line decoder, outputs 0 active
Decoder3To8[MWire(3,D),MWire(8,Y)]	3-to-8 line decoder
Decoder3To8_WB_One[MWire(3,D),MWire(8,Y)]	3-to-8 line decoder, outputs 1 active
Decoder3To8_WB_Zero[MWire(3,D),MWire(8,Y)]	3-to-8 line decoder, outputs 0 active
Encoder4to2[D3, D2, D1, D0, Q1, Q0]	4-to-2 encoder
Encoder4to2_WB_One[D3, D2, D1, D0, Q1, Q0]	4-to-2 encoder, inputs 1 active
Encoder4to2_WB_Zero[D0,D1,D2,D3,Q0,Q1]	4-to-2 encoder, inputs 0 active
Excess3toDec[MWire(4,D),MWire(10,Y)]	Excess-3-to Decimal decoder
Excess3toDec_WB_One[MWire(4,D),MWire(10,Y)]	Excess-3-to Decimal decoder, outputs 1 active
Excess3toDec_WB_Zero[MWire(4,D),MWire(10,Y)]	Excess-3-to Decimal decoder, outputs 0 active
Excess3GrtoDec[MWire(4,D),MWire(10,Y)]	Excess-3-Gray-to-Decimal decoder
Excess3GrtoDec_WB_One[MWire(4,D),MWire(10,Y)]	Excess-3-Gray-to-Decimal decoder, outputs 1 active
Excess3GrtoDec_WB_Zero[MWire(4,D),MWire(10,Y)]	Excess-3-Gray-to-Decimal decoder, outputs 0 active

Table 7: Encoders and Decoders

Library Component	Meaning
Comparator1[Ai,Bi,Xi,Yi,Ap,Bp]	1-bit comparator
Comparator1_WB[Ai,Bi,Xi,Yi,Ap,Bp]	1-bit comparator
Comparator4[Ls4,Gr4,MWire(4,‘X,Y’),Ls0,Gr0]	4-bit comparator
Comparator8[Ls8,Gr8,MWire(8,‘X,Y’),Ls0,Gr0]	8-bit comparator
Comparator4_BB[Ls4,Gr4,MWire(4,‘X,Y’),Ls0,Gr0]	4-bit comparator
ComparatorN_BB[Ls,Gr,X,Y,Ls0,Gr0]	*n*-bit comparator

Table 8: Comparators

Library Component	Meaning
Bi_Counter4_Reset[Q4,R1,R2,MWire(4,Q)]	4-bit binary counter with reset
Bi_Counter4_BB_Reset[Q4,R1,R2,MWire(4,Q)]	4-bit binary counter with reset
Clock[C]	Clock
Divider2[C,Q]	Divide-by-2 counter
Divider2_BB_Neg[C,Q]	Divide-by-2 counter, negative clock
Divider2_BB_Pos[C,Q]	Divide-by-2 counter, positive clock
Divider4[C,Q1,Q0]	Divide-by-4 counter
Divider4_BB_Neg[C,Q1,Q0]	Divide-by-4 counter, negative clock
Divider4_BB_Pos[C,Q1,Q0]	Divide-by-4 counter, positive clock
Divider8[C,Q2,Q1,Q0]	Divide-by-8 counter
Divider8_BB_Neg[C,Q2,Q1,Q0]	Divide-by-8 counter, negative clock
Divider8_BB_Pos[C,Q2,Q1,Q0]	Divide-by-8 counter, positive clock

Table 9: Counters

Library Component	**Meaning**
DFlipFlop[D,C,Q,Qbar]	Master-Slave D flip-flop
DFlipFlop_BB_Neg[D,C,Q,Qbar]	Negative edge-triggered D flip-flop
DFlipFlop_BB_Pos[D,C,Q,Qbar]	Positive edge-triggered D flip-flop
DFlipFlop_BB_PreClr_Neg[D,Preset,Clear,C,Q,Qbar]	Negative edge-triggered D flip-flop, preset and preclear
DFlipFlop_BB_PreClr_Pos[D,Preset,Clear,C,Q,Qbar]	Positive edge-triggered D flip-flop, preset and preclear
DFlipFlopLockOut_BB_Neg[D,C,Q,Qbar]	Master-Slave D flip-flop, lockout, data output on negative clock
DFlipFlopLockOut_BB_Pos[D,C,Q,Qbar]	Master-Slave D flip-flop, lockout, data output on positive clock
DFlipFlop_PreClr[D,Preset,Clear,Ck,Q,Qbar]	Master-Slave D flip-flop, preset and preclear
JKFlipFlop[J,K,C,Q,Qbar]	Master-Slave JK flip-flop
JKFlipFlop_BB_Neg[J,K,C,Q,Qbar]	Negative edge-triggered JK flip-flop
JKFlipFlop_BB_Pos[J,K,C,Q,Qbar]	Positive edge-triggered JK flip-flop
JKFlipFlop_BB_PreClr_Neg[J,K,Preset,Clear,C,Q,Qbar]	Negative edge-triggered JK flip-flop, preset and preclear
JKFlipFlop_BB_PreClr_Pos[J,K,Preset,Clear,C,Q,Qbar]	Positive edge-triggered JK flip-flop, preset and preclear
JKFlipFlopLockOut_BB_Neg[J,K,C,Q,Qbar]	Master-Slave JK flip-flop, lockout, data output on negative clock
JKFlipFlopLockOut_BB_Pos[J,K,C,Q,Qbar]	Master-Slave JK flip-flop, lockout, data output on positive clock
JKFlipFlop_PreClr[J,K,Preset,Clear,Ck,Q,Qbar]	Master-Slave JK flip-flop, preset and preclear
MSFlipFlop[R,S,C,Q,Qbar]	Master-Slave RS flip-flop
MSFlipFlop_PreClr[R,S,Preset,Clear,Ck,Q,Qbar]	Master-Slave RS flip-flop, preset and preclear
RSFlipFlopEdge[R,S,C,Q,Qbar]	Edge-triggered RS flip-flop
RSFlipFlop_BB_Neg[R,S,C,Q,Qbar]	Negative edge-triggered RS flip-flop
RSFlipFlop_BB_Pos[R,S,C,Q,Qbar]	Positive edge-triggered RS flip-flop
RSFlipFlop_BB_PreClr_Neg[R,S,Preset,Clear,C,Q,Qbar]	Negative edge-triggered RS flip-flop, preset and preclear
RSFlipFlop_BB_PreClr_Pos[R,S,Preset,Clear,C,Q,Qbar]	Positive edge-triggered RS flip-flop, preset and preclear
TFlipFlop[C,Q,Qbar]	T flip-flop
TFlipFlop_BB_Neg[D,C,Q,Qbar]	Negative edge-triggered T flip-flop, preset and preclear
TFlipFlop_BB_Pos[D,C,Q,Qbar]	Positive edge-triggered T flip-flop, preset and preclear

Table 10: Flip-Flops

Library Component	Meaning
CDRSLatch[R,S,D0,D1,C,Q,Qbar]	Clocked Reset-Set latch, data input
CDRSLatch_WB[R,S,D0,D1,C,Q,Qbar]	Clocked Reset-Set latch, data input
CRSLatch[R,S,C,Q,Qbar]	Clocked Reset-Set latch
CRSLatch_WB[R,S,C,Q,Qbar]	Clocked Reset-Set latch
DLatch[D,C,Q,Qbar]	Delay latch
DLatch_WB[D,C,Q,Qbar]	Delay latch
DLatch4[MWire(4,D),G,MWire(4,Q)]	4-bit D latch
DLatch4_BB[MWire(4,D),G,MWire(4,Q)]	4-bit D latch
DLatch8[MWire(8,D),G,MWire(8,Q)]	8-bit D latch
DLatch8_BB[MWire(8,D),G,MWire(8,Q)]	8-bit D latch
DRSLatch[R,S,D0,D1,Q,Qbar]	Reset-Set latch, data input
DRSLatch_WB[R,S,D0,D1,Q,Qbar]	Reset-Set latch, data input
Latch_WB_PreClr[R,S,Preset,Clear,Ck,Q,Qbar]	RS latch, preset and preclear
Latch_PreClr[R,S,Preset,Clear,Ck,Q,Qbar]	RS latch, preset and preclear
Latch_WB_PreClr[R,S,Preset,Clear,Ck,Q,Qbar]	RS latch, preset and preclear
RSLatch[R,S,Q,Qbar]	Reset-Set latch
RSLatch_BB[R,S,Q,Qbar]	Reset-Set latch
RSLatch_WB[R,S,Q,Qbar]	Reset-Set latch

Table 11: Latches

Library Component	Meaning
Reg_4T4_RW[MWire(4,D),Gw,Wb,Wa,Gr,Rb,Ra,MWire(4,Q)]	4*4-bit memory, read and write
Reg_4T4_BB_RW[D,Gw,Wb,Wa,Gr,Rb,Ra,Q]	4*4-bit memory, read and write

Table 12: Memories

Library Component	Meaning
Parity8[MWire(8,D),P]	8-bit parity checker/generator
Parity8_WB[MWire(8,D),P]	8-bit parity checker/generator

Table 13: Parity Checkers/Generators

Library Component	Meaning
Demultiplexer1to2[D,S,Q1,Q0]	1-to-2 demultiplexer
Demultiplexer1to2_WB[D,S,Q1,Q0]	1-to-2 demultiplexer
Demultiplexer2to4[D,S1,S0,Q3,Q2,Q1,Q0]	2-to-4 demultiplexer
Demultiplexer2to4_WB[D,S1,S0,Q3,Q2,Q1,Q0]	2-to-4 demultiplexer
Multiplexer4to1[D3,D2,D1,D0,S1,S0,Q]	1-bit 4-to-2 multiplexer
Multiplexer4to1_WB[D3,D2,D1,D0,S1,S0,Q]	1-bit 4-to-2 multiplexer
Multiplexer2to1[A,B,S,C]	1-bit 2-to-1 multiplexer
Multiplexer2to1_BB[A,B,S,C]	1-bit 2-to-1 multiplexer
Multiplexer2to1_WB[A,B,S,C]	1-bit 2-to-1 multiplexer
Multiplexer2to1_8[MWire(8,'A,B'),S,MWire(8,C)]	8-bit 2-to-1 multiplexer
Multiplexer2to1_8_BB[MWire(8,A),MWire(8,B),S, MWire(8,C)]	8-bit 2-to-1 multiplexer
Multiplexer2to1Reg_8[MWire(8,'A,B'),S,Ck, MWire(8,Q)]	8-bit 2-to-1 multiplexer, register output
Multiplexer2to1_RegN_BB_Pos[A,B,S,Ck,C]	*n*-bit 2-to-1 multiplexer, register output

Table 14: Multiplexers and Demultiplexers

Library Component	Meaning
BucketBrigade2[J0,J1,K0,C,Q0,Q1]	2-bit bucket brigade
PassOn2[D,C0,C1,Q]	2-bit sequential pass-on
Register4[MWire(4, D), Clk, MWire(4, Q)]	4-bit D-type register
Register4_BB_Neg [MWire(4, D), C, MWire(4, Q)]	4-bit D-type register, data output on negative clock
Register4_BB_Pos [MWire(4, D), C, MWire(4, Q)]	4-bit D-type register,data output on positive clock
Register8[MWire(8,D),C,MWire(8,'Q,Qbar')]	8-bit register
RegisterN_BB_Neg[D,C,Q,Qbar]	*n*-bit register, data output on negative clock
RegisterN_BB_Pos[D,C,Q,Qbar]	*n*-bit register, data output on positive clock
RegisterLoadClr[D,G,Clr,Clk,Q]	1-bit register, data load enable, preclear,data output on positive clock
RegisterLoadClr4[MWire(4,D),G,Clr,Clk,MWire(4,Q)]	4-bit register, data load enable, preclear, data output on positive clock
RegisterLoadClr4_Tri[M,N,MWire(4,D),G1,G2,Clr,Clk, MWire(4,Q)]	4-bit register, data load enable, preclear, tri-state output, data output on positive clock
RegisterLoadClr_BB_Neg[D,G,Clr,Clk,Q]	1-bit register, data load enable, preclear,data output on negative clock
RegisterLoadClr_BB_Pos[D,G,Clr,Clk,Q]	1-bit register, data load enable, preclear,data output on positive clock
RegisterLoadClr4_BB_Neg[D,G,Clr,Clk,Q]	4-bit register, data load enable, preclear,data output on negative clock
RegisterLoadClr4_BB_Pos[D,G,Clr,Clk,Q]	4-bit register, data load enable, preclear,data output on positive clock
ShiftRegister2[D,C,Q0,Q1]	2-bit shift register
ShiftRegister2_BB_Neg[D,C,Q1,Q0]	2-bit shift register, data output on negative clock
ShiftRegister2_BB_Pos[D,C,Q1,Q0]	2-bit shift register, data output on positive clock
ShiftRegister8[D0,C,D8]	8-bit shift register
ShiftRegister8_BB_Neg[D,C,MWire(8,Q)]	8-bit shift register, data output on negative clock
ShiftRegister8_BB_Pos[D,C,MWire(8,Q)]	8-bit shift register, data output on positive clock

Table 15: Registers

C Gate-Level Component Specifications

This appendix gives gate-level circuit specifications for the new DILL components. Specifications for the original DILL components can be found in [4].

C.1 8-Bit Ripple Adder

See figure 12.

```
process RippleAdder8[MWire(8,'A,B'),c0, MWire(8,S), c8] :noexit :=
   hide MWire(7,c+) in
   MComp(8,c,'FullAdder[A,B,C,S,C+]')
  |[c0]|
   zero[c0]
endproc (* RippleAdder8 *)
```

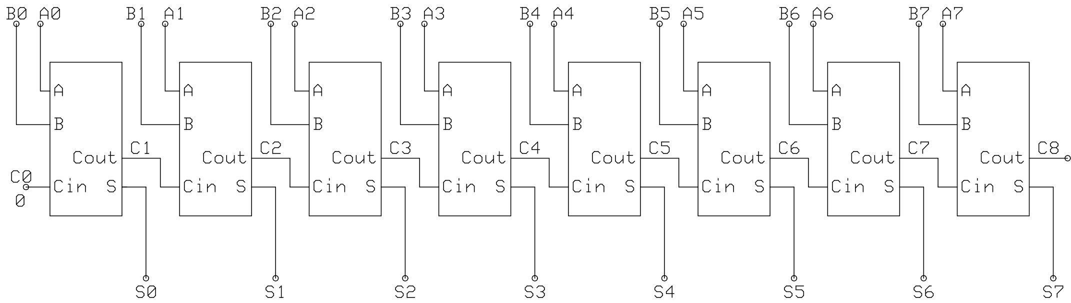


Figure 12: 8-Bit Ripple Adder

C.2 3-to-8 Line Decoder

See figure 13.

```
process Decoder3To8[MWire(3,D),MWire(8,Y)] : noexit:=
    hide MWire(3,Ind) in
       (MComp(3,'Inverter[D,Ind]'))
   |[D2,D1,D0,Ind2,Ind1,Ind0]|
       (And3[D2,D1,D0,Y7] ||| And3[Ind2,Ind1,Ind0,Y0] )
   |[D2,D1,D0,Ind2,Ind1,Ind0]|
       (And3[D2,D1,Ind0,Y6] ||| And3[Ind2,Ind1,D0,Y1] )
   |[D2,D1,D0,Ind2,Ind1,Ind0]|
       (And3[D2,Ind1,D0,Y5] ||| And3[Ind2,D1,Ind0,Y2] )
   |[D2,D1,D0,Ind2,Ind1,Ind0]|
       (And3[D2,Ind1,Ind0,Y4] ||| And3[Ind2,D1,D0,Y3] )
endproc (* Decoder3To8 *)
```

C.3 BCD-to-Decimal Decoder

See figure 14. Outputs are 0 active.

```
process BCDtoDec[MWire(4,D),MWire(10,Y)] : noexit:=
  hide MWire(4,Ind),MWire(10,Ybar) in
   MComp(4,'Inverter[D,Ind]')
  |[MWire(4,'D,Ind')]|
   (
```

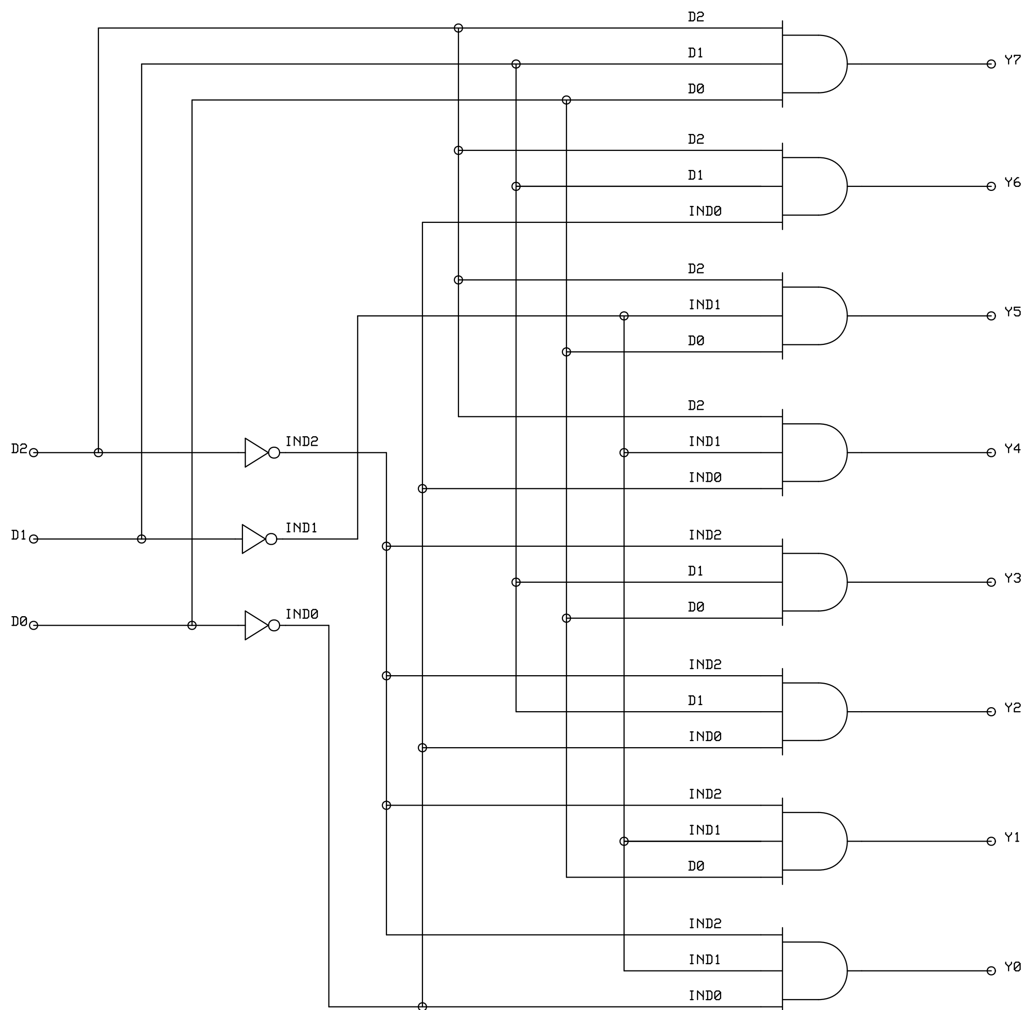


Figure 13: 3-to-8 Line Decoder

```
    ((((((((Nand4[Ind3,Ind2,Ind1,Ind0,Y0]        |[Ind3,Ind2,Ind1]|
    Nand4[Ind3,Ind2,Ind1,D0,Y1])                 |[Ind3,Ind2,Ind0]|
    Nand4[Ind3,Ind2,D1,Ind0,Y2])                 |[Ind3,Ind2,D1,D0]|
    Nand4[Ind3,Ind2,D1,D0,Y3])                   |[Ind3,Ind1,Ind0]|
    Nand4[Ind3,D2,Ind1,Ind0,Y4])                 |[Ind3,D2,Ind1,D0]|
    Nand4[Ind3,D2,Ind1,D0,Y5])                   |[Ind3,D2,D1,Ind0]|
    Nand4[Ind3,D2,D1,Ind0,Y6])                   |[Ind3,D2,D1,D0]|
    Nand4[Ind3,D2,D1,D0,Y7])                     |[Ind2,Ind1,Ind0]|
    Nand4[D3,Ind2,Ind1,Ind0,Y8])                 |[D3,Ind2,Ind1,D0]|
    Nand4[D3,Ind2,Ind1,D0,Y9]
  )
endproc (* BCDtoDec *)
```

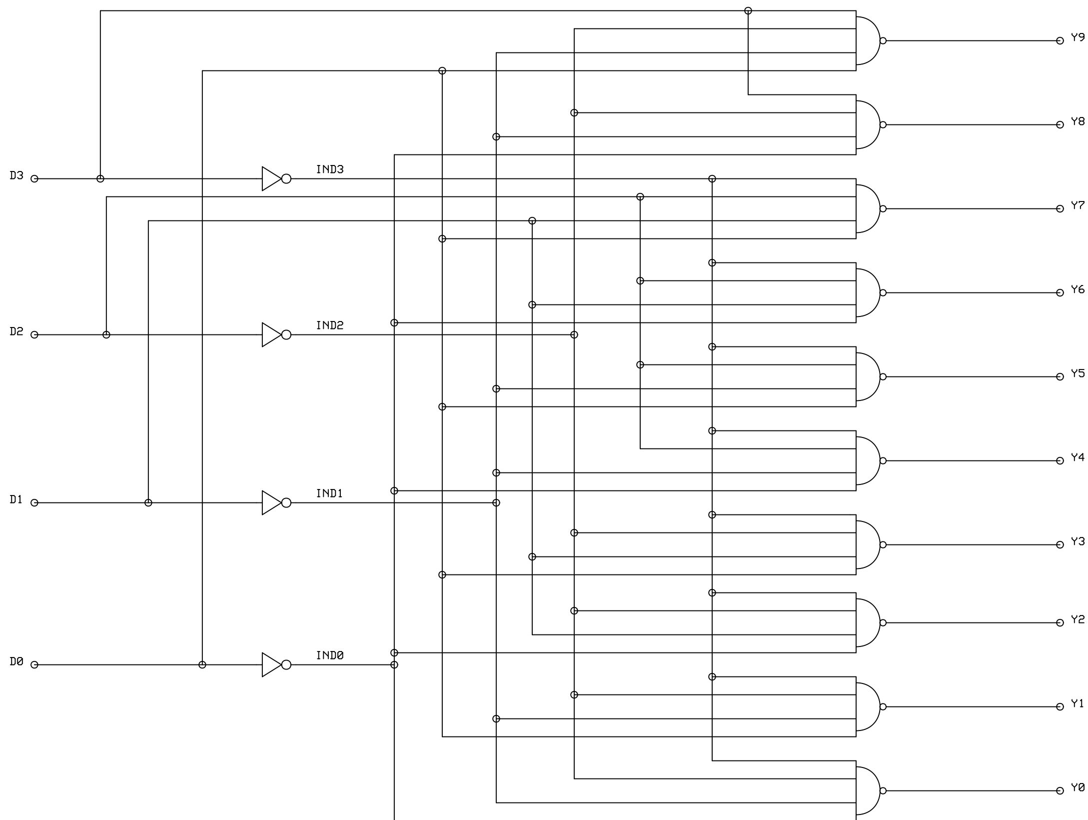


Figure 14: BCD-to-Decimal Decoder

C.4 Excess-3-to Decimal Decoder

Outputs are 0 active.

```
process Excess3toDec[MWire(4,D),MWire(10,Y)] : noexit:=
 hide MWire(4,Ind),MWire(10,Ybar) in
  MComp(4,‘Inverter[D,Ind]’)
 |[MWire(4,‘D,Ind’)]|
  (
    ((((((((Nand4[Ind3,Ind2,D1,D0,Y0]     |[Ind3]|
    Nand4[Ind3,D2,Ind1,Ind0,Y1])          |[Ind3,D2,Ind1,Ind0]|
```

```
    Nand4[Ind3,D2,Ind1,Ind0,Y2])            |[Ind3,D2,D1,Ind0]|
    Nand4[Ind3,D2,D1,Ind0,Y3])              |[Ind3,D2,D1,D0]|
    Nand4[Ind3,D2,D1,D0,Y4])                |[Ind2,Ind1,Ind0]|
    Nand4[D3,Ind2,Ind1,Ind0,Y5])            |[D3,Ind2,Ind1,D0]|
    Nand4[D3,Ind2,Ind1,D0,Y6])              |[D3,Ind2,D1,Ind0]|
    Nand4[D3,Ind2,D1,Ind0,Y7])              |[D3,Ind2,D1,D0]|
    Nand4[D3,Ind2,D1,D0,Y8])                |[D3,D2,Ind1,Ind0]|
    Nand4[D3,D2,Ind1,Ind0,Y9]
  )
endproc (* Excess3toDec *)
```

C.5 Excess-3-Gray-to-Decimal Decoder

Outputs are 0 active.

```
process Excess3GrtoDec[MWire(4,D),MWire(10,Y)] : noexit:=
 hide MWire(4,Ind),MWire(10,Ybar) in
  MComp(4,'Inverter[D,Ind]')
 |[MWire(4,'D,Ind')]|
  (
    ((((((((Nand4[Ind3,Ind2,D1,Ind0,Y0]           |[Ind3,D1,Ind0]|
    Nand4[Ind3,D2,D1,Ind0,Y1])                   |[Ind3,D2,D1]|
    Nand4[Ind3,D2,D1,D0,Y2])                     |[Ind3,D2,D0]|
    Nand4[Ind3,D2,Ind1,D0,Y3])                   |[Ind3,D2,Ind1,Ind0]|
    Nand4[Ind3,D2,Ind1,Ind0,Y4])                 |[D2,Ind1,Ind0]|
    Nand4[D3,D2,Ind1,Ind0, Y5])                  |[D3,D2,Ind1,D0]|
    Nand4[D3,D2,Ind1,D0,Y6])                     |[D3,D2,D1,D0]|
    Nand4[D3,D2,D1,D0,Y7])                       |[D3,D2,D1,Ind0]|
    Nand4[D3,D2,D1,Ind0,Y8])                     |[D3,Ind2,D1,Ind0]|
    Nand4[D3,Ind2,D1,Ind0,Y9]
  )
endproc (* Excess3GrtoDec *)
```

C.6 1-Bit Comparator

See figure 15. This is the cell from which multi-bit comparators are built. The signals are:

Ai, Bi: *i*th bits to be compared

Xi, Yi: compared results from the *i+1*th bits; 0, 0 for equal, 1, 0 for $A < B$, 0, 1 for $A > B$

Ap, Bp: compared results of the *i*th bits

```
process Comparator1[Ai,Bi,Xi,Yi,Ap,Bp] :noexit :=
  hide Inx,Iny,Ina,Inb,A3,B3 in
   (Inverter[Xi,Inx] ||| Inverter[Yi,Iny]
  |||
    Inverter[Ai,Ina] ||| Inverter[Bi,Inb]
   )
  |[Xi,Yi,Ina,Inb,Inx,Iny]|
   ((Nand3[Inx,Yi,Inb,A3] |[a3]| Nand2[A3,Ina,Ap])
    |[ina,inb]|
    (Nand3[Xi,Iny,Ina,B3] |[b3]| Nand2[B3,Inb,Bp])
   )
 endproc (* Comparator1 *)
```

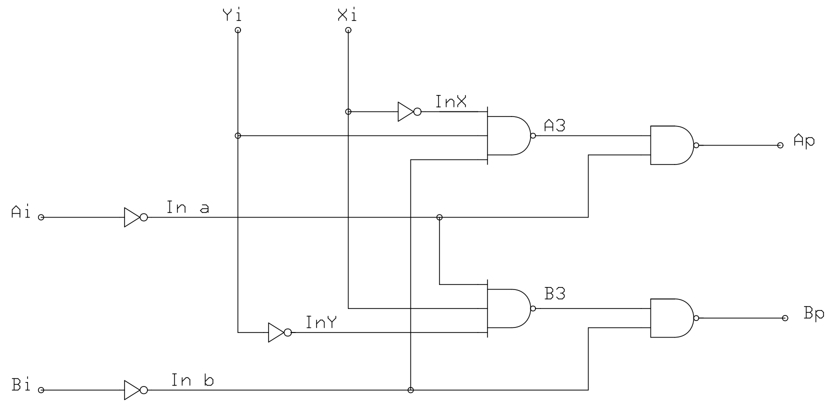


Figure 15: 1-Bit Comparator

C.7 4-Bit Comparator

See figure 16.

process Comparator4[Ls4,Gr4,MWire(4,'X,Y'),Ls0,Gr0] :**noexit** :=
  hide MWire(3,'Ls+,Gr+') **in**
    MComp(4,'Ls,Gr','Comparator1[Ls+,Gr+,X,Y,Ls,Gr]')
endproc (* Comparator4 *)

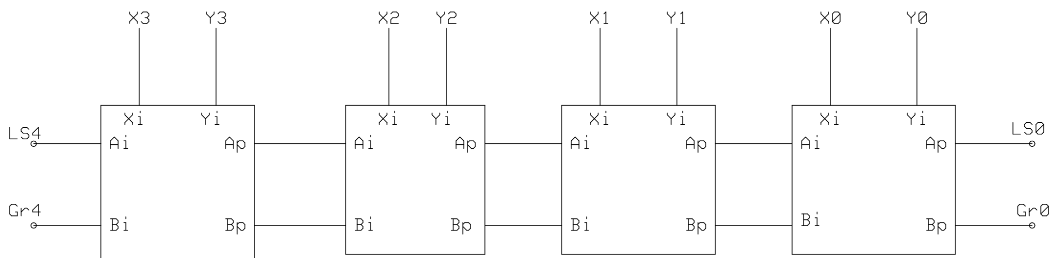


Figure 16: 4-Bit Comparator

C.8 8-Bit Comparator

process Comparator8[Ls8,Gr8,MWire(8,'X,Y'),Ls0,Gr0] :**noexit** :=
  hide Ls4,Gr4 **in**
    (Zero[Ls8] ||| Zero[Gr8])
  |[Ls8,Gr8]|
      Comparator4[Ls8,Gr8,MWire(4,'X+++,Y+++'),Ls4,Gr4]
  |[Ls4,Gr4]|
      Comparator4[Ls4,Gr4,MWire(4,'X,Y'),Ls0,Gr0]
endproc (* Comparator8 *)

C.9 4-Bit Binary Counter with Reset (74LS93)

See figure 17.

R1, R2: Asynchronous Reset inputs, 1 active

Q4: clock signal

Q3–Q0: counter outputs, change at the negative transition of the clock signal if R1, R2 are not active.

Asynchronous Reset inputs are 1 active. Asynchronous reset means the counter may be reset to 0 independently of the clock input. In most cases, it happens prior to the count operation.

```
process Bi_Counter4_Reset[Q4,R1,R2,MWire(4,Q)] :noexit :=
 hide R1R2,MWire(4,‘J,K,Pre,Qbar’) in
  Nand2[R1,R2,R1R2]
  |[R1R2]|
  MComp(4,‘R1R2=,Q’,‘JKFlipFlop_PreClr[J,K,Pre,R1R2=,Q+,Q,Qbar]’)
  |[MWire(4,‘J,K,Pre’)]|
  ( MComp(4,One[J]) ||| MComp(4,One[K]) ||| MComp(4,One[Pre]) )
endproc (* Bi_Counter4_Reset *)
```

Figure 17: 4-Bit Binary Counter with Reset

C.10 Master Slave RS Flip-Flop with Preset and Preclear

See figure 18. Asynchronous preset and preclear inputs are 0 active. Asynchronous preset and clear means the flip-flop may be set to 1 or reset to 0 independently of the clock input. Outputs change at the negative transition of the clock signal if the preset and preclear are not active.

```
process MSFlipFlop_PreClr[R,S,Preset,Clear,Ck,Q,Qbar] :noexit :=
 hide Ckint,Rint,Sint in
   Inverter[Ck,Ckint]
```

```
  |[Ck,Ckint]|
   ( CRSLatch[R,S,Ck,Rint,Sint]
   |[Rint,Sint]|
      Latch_PreClr[Sint,Rint,Preset,Clear,Ckint,Q,Qbar]
   )
endproc (* MSFlipFlop_PreClr *)
```

Figure 18: Master-Slave RS Flip-Flop with Preset and Preclear

C.11 Master Slave JK Flip-Flop With Preset and Preclear

Asynchronous preset and preclear inputs are 0 active. Outputs change at the negative transition of the clock signal if the preset and preclear are not active.

```
process JKFlipFlop_PreClr[J,K,Preset,Clear,Ck,Q,Qbar] :noexit :=
```

```
    hide Rint,Sint in
      (And2[J,Qbar,Sint] ||| And2[K,Q,Rint])
    |[Rint,Sint,Q,Qbar]|
      MSFlipFlop_PreClr[Rint,Sint,Preset,Clear,Ck,Q,Qbar]
endproc (* JKFlipFlop_PreClr *)
```

C.12 Master Slave D Flip-Flop With Preset and Preclear

Asynchronous preset and preclear inputs are 0 active. Outputs change at the negative transition of the clock signal if the preset and preclear are not active.

```
process DFlipFlop_PreClr[D,Preset,Clear,Ck,Q,Qbar] :noexit :=
   hide Dint in
     Inverter[D,Dint]
  |[D,Dint]|
     JKFlipFlop_PreClr[D,Dint,Preset,Clear,Ck,Q,Qbar]
endproc (* DFlipFlop_PreClr *)
```

C.13 Edge-Triggered RS Flip-Flop

Outputs change at the negative transition of the clock signal.

```
process RSFlipFlopEdge [R,S,Ck,Q,Qbar] : noexit :=
  hide A,B,C,D in
    ((Nand2[S,Ck,A] |[A,Ck]| Or2[Ck,A,C]) |[C]| Nand2[C,Qbar,Q])
  |[Ck,Q,Qbar]|
    ((Nand2[R,Ck,B] |[B,Ck]| Or2[Ck,B,D]) |[D]| Nand2[D,Q,Qbar])
endproc (* RSFlipFlopEdge *)
```

C.14 RS Latch with Preset and Preclear

See figure 19. Asynchronous preset and preclear are 0 active. Outputs change when CK is 1 if the preset and preclear are not active.

```
process Latch_PreClr[R,S,Preset,Clear,Ck,Q,Qbar] :noexit :=
  hide Rint,Sint in
    (Nand3[S,Clear,Ck,Sint] |[Ck]| Nand3[R,Preset,Ck,Rint])
  |[Rint,Sint,Preset,Clear]|
    (Nand3[Sint,Preset,Qbar,Q] |[Q,Qbar]| Nand3[Rint,Clear,Q,Qbar])
endproc (* Latch_PreClr *)
```

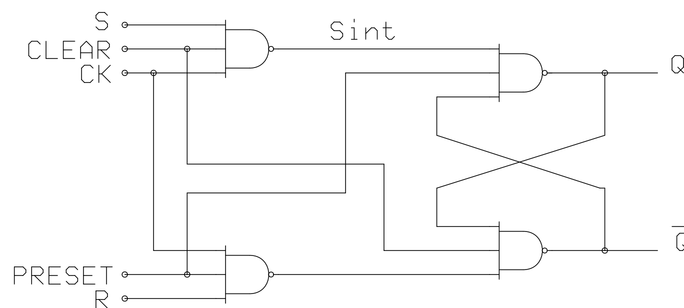


Figure 19: RS Latch with Preset and Preclear

C.15 4-Bit D Latch

```
process DLatch4[MWire(4,D),G,MWire(4,Q)] :noexit :=
  hide MWire(4,Qbar) in
  MComp(4,G=,‘DLatch[D,G=,Q,Qbar]’)
endproc (* DLatch4 *)
```

C.16 4*4-Bit Memory with Read and Write (74LS170)

See figure 20.

D3–D0: data inputs

Gw/Gr: write/read enable (0 active)

Wb, Wa/Rb, Ra: write/read address

Q3–Q0: data outputs

```
process Reg_4T4_RW[MWire(4,D),Gw,Wb,Wa,Gr,Rb,Ra,MWire(4,Q)]:noexit :=
 hide MWire(4,‘W,Gwin,Q0,Q1,Q2,Q3,Qin’) in
 (( Decoder2[Wb,Wa,MWire(4,W)]
    |[MWire(4,W)]|
     MComp(4,Gw=,‘Nor2[Gw=,W,Gwin]’)
   )
  |[MWire(4,Gwin)]|
   ( DLatch_4[MWire(4,D),Gwin3,MWire(4,Q3)]
    |[MWire(4,D)]|
     DLatch_4[MWire(4,D),Gwin2,MWire(4,Q2)]
    |[MWire(4,D)]|
     DLatch_4[MWire(4,D),Gwin1,MWire(4,Q1)]
    |[MWire(4,D)]|
     DLatch_4[MWire(4,D),Gwin0,MWire(4,Q0)]
    )
  )
|[MWire(4,‘Q0,Q1,Q2,Q3’)]|
 (MComp(4,‘Rb=,Ra=’, ‘Multiplexer2[Q3,Q2,Q1,Q0,Rb=,Ra=,Qin]’)
  |[MWire(4,Qin)]|
   MComp(4,Gr=,‘Or2[Qin,Gr=,Q]’)
  )
endproc (* Reg_4T4_RW *)
```

C.17 8-Bit Parity Checker/Generator

See figure 21.

```
process Parity8 [MWire(8, D), P] : noexit :=
  hide MWire(4, A), MWire(2, B) in
MComp(4, ‘Xor2 [D+*, D*, A]’)
|[MWire(4, A)]|
MComp(2, ‘Xor2 [A+*, A*, B]’)
|[MWire(2, B)]|
  Xor2 [MWire(2, B), P]
endproc (* Parity8 *)
```

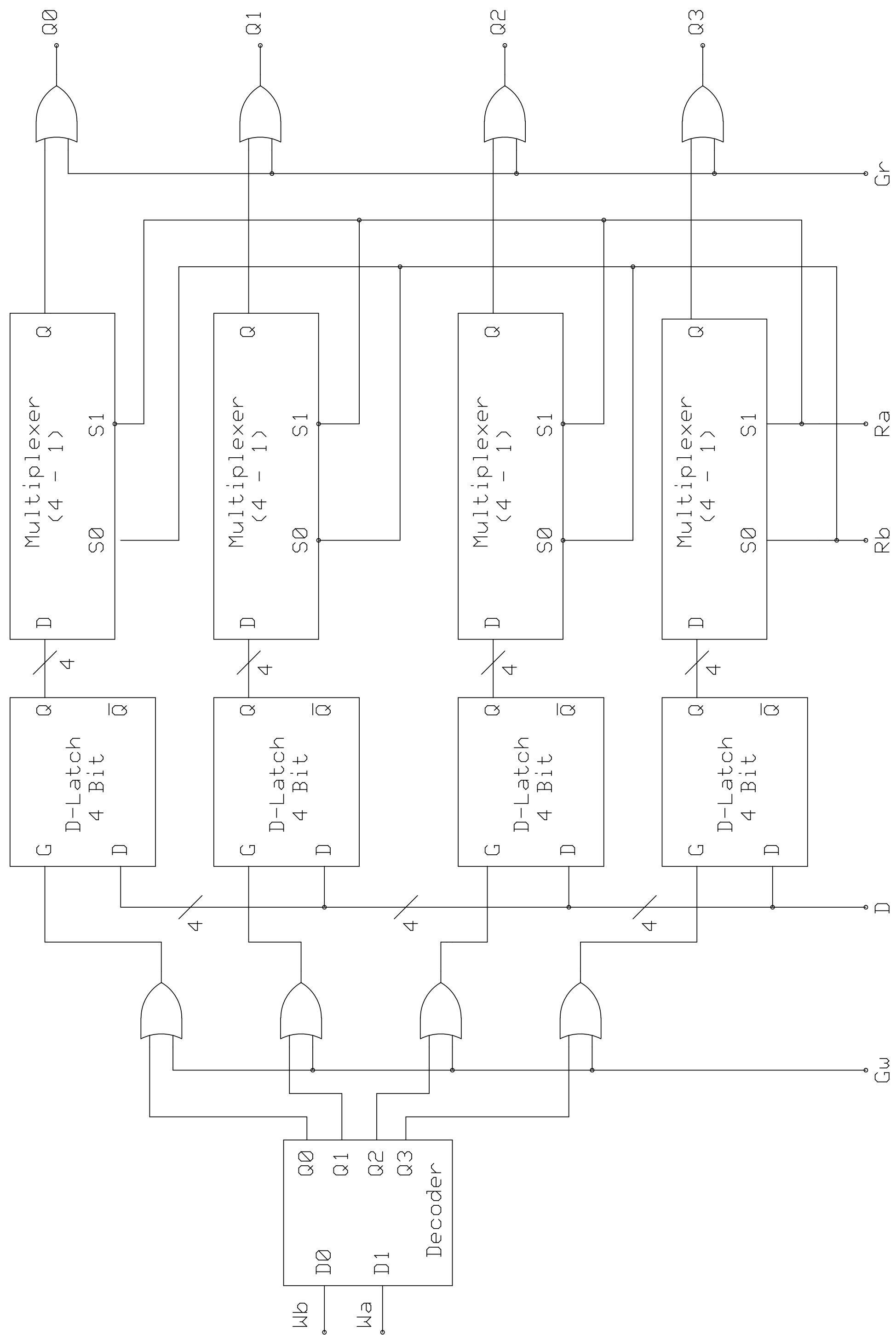


Figure 20: 4*4-Bit Memory with Read and Write

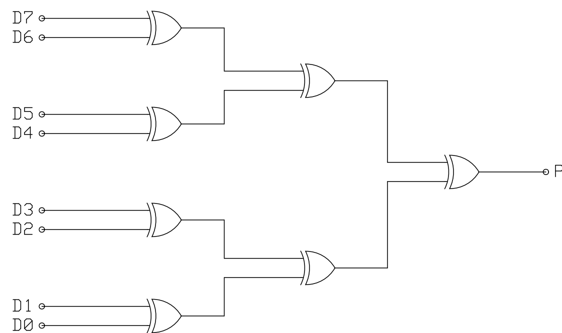


Figure 21: 8-Bit Parity Checker/Generator

C.18 2-1 Multiplexer

See figure 22.

```
process Multiplexer2to1[A,B,S,C] :noexit :=
   hide Ain,BIn,Sin in
     (Inverter[S,Sin]
   |[S,Sin]|
     ( And2[A,Sin,Ain] ||| And2[B,S,BIn]))
   |[Ain,BIn]|
       Or2[Ain,BIn,C]
endproc (* Multiplexer2to1 *)
```

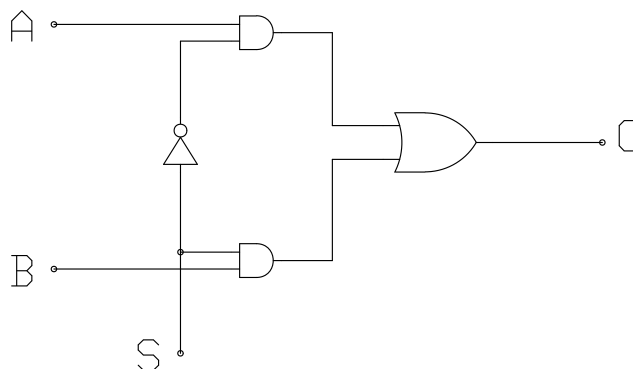


Figure 22: 2-to-1 Multiplexer

C.19 8-bit 2-1 Multiplexer

```
process Multiplexer2to1_8[MWire(8,'A,B'),S,MWire(8,C)] :noexit :=
   MComp(8,S=,'Multiplexer2to1[A,B,S=,C]')
endproc (* Multiplexer2to1_8 *)
```

C.20 8-bit 2-1 Multiplexer with Register Output

Outputs change at the negative transition of the clock signal.

```
process Multiplexer2to1Reg_8[MWire(8,'A,B'),S,Ck,MWire(8,Q)] :noexit :=
   hide MWire(8,'C,Qbar') in
```

```
    Multiplexer2to1_8[MWire(8,‘A,B’),S,MWire(8,C)]
  |[MWire(8,C)]|
    Multiplexer_8[MWire(8,C),Ck,MWire(8,‘Q,Qbar’)]
endproc (* Multiplexer2to1Reg_8_Decl *)
```

C.21 8-Bit Register

D7–D0: data inputs

Q7–Q0: data outputs

Qbar7–Qbar0: data outputs negated

C: common clock input

Outputs change at the negative transition of the clock signal.

```
process Register_8 [MWire(8, D), C,MWire(8,‘Q,Qbar’)] : noexit :=
  MComp(8, C=, ‘DFlipFlop[D, C=, Q, Qbar]’)
endproc (* Register_8 *)
```

C.22 8-Bit Shift Register

D8: data input

D0: data output

C: common clock input

Data output changes at the negative transition of the clock signal.

```
process ShiftRegister8 [D0, C,D8] : noexit :=
  hide MWire(7, D+),
    MWire(8, Dbar+) in
  MComp(8, ‘D, C=’, ‘DFlipFlop[D, C=, D+, Dbar+]’)
|[MWire(8, Dbar+)]|
  MComp(8, ‘Sink [Dbar+]’)
endproc (* ShiftRegister8 *)
```

C.23 4-Bit D-Type Register with Tri-State Output (74LS173)

See figure 23.

D3–D0: data inputs

G1, G2: data enable, used to select register (0 active)

Clr: asynchronous clear

Clk: clock

M, N: output control

Q3–Q0: data outputs

Data outputs change at the positive transition of the clock signal.

```
process Register_4_Tri[M,N,MWire(4,D),G1,G2,Clr,Clk,MWire(4,Q)] :noexit :=
 hide Mn,G,GIn,Clkin,MWire(4,'QIn,QBar,DIn,DDIn,Ind'),Pre in
  (Nor2[M,N,Mn] ||| (Nor2[G1,G2,g] |[g]| Inverter[G,GIn])
  ||| Inverter[Clk,Clkin] ||| One[Pre])
 |[Mn,G,GIn,Clkin,Pre]|
  ( ((((And2[G,D3,DIn3] ||| And2[QIn3,GIn,DDIn3])
   |[DIn3,DDIn3]|
    Or2[DIn3,DDIn3,Ind3])
   |[Ind3,QIn3]|
    DFlipFlop_PreClr[Ind3,Pre,Clr,Clkin,QIn3,QBar3])
   |[QBar3]|
    Inverter_PosTri[QBar3,Mn,Q3]
    )
  |[Mn,G,GIn,Pre,Clr,Clkin]|
    ((((And2[G,D2,DIn2] ||| And2[QIn2,GIn,DDIn2])
   |[DIn2,DDIn2]|
    Or2[DIn2,DDIn2,Ind2])
   |[Ind2,QIn2]|
    DFlipFlop_PreClr[Ind2,Pre,Clr,Clkin,QIn2,QBar2])
   |[QBar2]|
    Inverter_PosTri[QBar2,Mn,Q2]
    )
  |[Mn,G,GIn,Pre,Clr,Clkin]|
    ((((And2[G,D1,DIn1] ||| And2[QIn1,GIn,DDIn1])
   |[DIn1,DDIn1]|
    Or2[DIn1,DDIn1,Ind1])
   |[Ind1,QIn1]|
    DFlipFlop_PreClr[Ind1,Pre,Clr,Clkin,QIn1,QBar1])
   |[QBar1]|
    Inverter_PosTri[QBar1,Mn,Q1]
    )
  |[Mn,G,GIn,Pre,Clr,Clkin]|
    ((((And2[G,D0,DIn0] ||| And2[QIn0,GIn,DDIn0])
   |[DIn0,DDIn0]|
    Or2[DIn0,DDIn0,Ind0])
   |[Ind0,QIn0]|
    DFlipFlop_PreClr[Ind0,Pre,Clr,Clkin,QIn0,QBar0])
   |[QBar0]|
    Inverter_PosTri[QBar0,Mn,Q0]
    )
  )
endproc (* Register_4_Tri *)
```

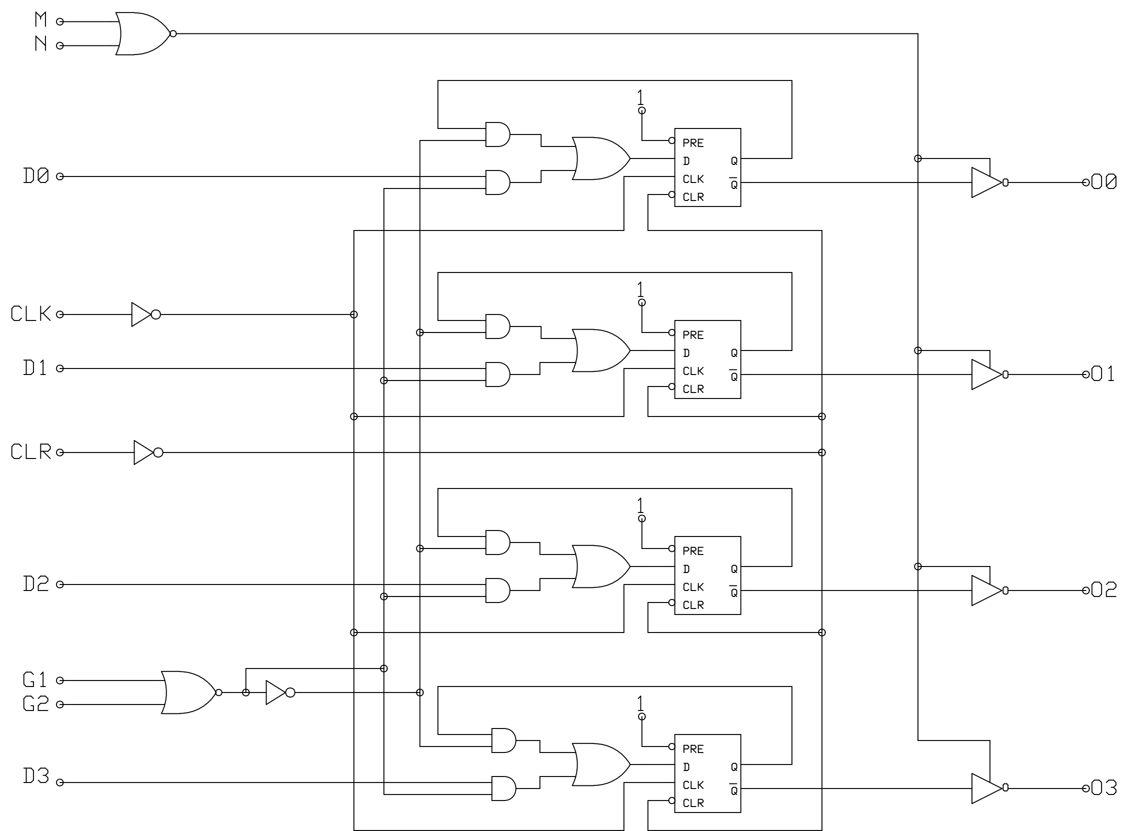


Figure 23: 4-Bit D-Type Register with Tri-State Output

D Signals in the Sub-CPU

D.1 Signals in the Behaviour Specification

LOTOS gates in the behaviour specification of the sub-CPU may represent either multi-bit wires or one-bit wires.

IR: Instruction input, 9-bit wire

DtIn: Data Input from the memory, 4-bit wire

Clk: system Clock, one-bit wire

R3,R2,R1,R0: outputs of four data Registers, each output being a 4-bit wire; shortened as *MWire(4,R)*

Flag1,Flag0: outputs of flag registers, each a 1-bit wire; Flag1 is the carry flag and Flag0 the comparison flag, with 1 meaning two operands are equal and 0 that they are unequal

BusAorMArr: Memory Address, 4-bit wire

BusBorDtOut: Data Output to memory, 4-bit wire

MW: Memory Write signal. one-bit wire. 1 stands for memory write enable, 0 for memory write disable

D.2 Signals in the Structural Specification

LOTOS gates in the structural specification represent one-bit wires. Multi-bit wires can be shortened with MWire macro.

MWire(9,IR): Instruction inputs

MWire(4,DtIn): Data Inputs from the memory

Clk: same as that in the behaviour specification

MWire(4,'R3, R2, R1, R0'): the outputs of the four data Registers; for example, R33 is the most significant bit of the register R3

Flag1, Flag0: same as that in the behaviour specification

MWire(4,'BusAorMArr'): the internal Bus for source operand A within the sub-CPU, or the external bus for Memory Address

MWire(4,'BusBorDtOut'): the internal Bus for source operand B within the sub-CPU, or the external bus for Data Outputs to memory

MW: same as that in the behaviour specification

The following are the important control signals from the control unit of the sub-CPU:

RW: Register Write signal; 1 stands for write enable, 0 for disable

AorC: Addition or Comparison; 1 stands for addition, 0 for comparison

MorF: data comes from Memory or from function unit (ALU); 1 stands for from memory, 0 for from function unit

LI: data loaded from immediate operand or from a memory unit; 1 stands for loading from immediate operand, 0 for from a memory unit

ADD: 1 stands for an addition instruction

CMP: 1 stands for a comparison instruction

MWire(4,G): determine the destination register, 1 active; for example 0001 means that the loaded data will be sent to register R0.

MWire(4,SA): determine the value on the BusA bus, 1 active; for example 0001 means that the content of register R0 will be appear on BusA, the signals controlling the enable pins of the tri-state gates

MWire(4,SB): similar to *MWire(4,SA)* but referring to the BusB bus

E Behaviour Specification of the Sub-CPU

This is a simplified behaviour specification of the sub-CPU. In the specification, we assume that if an instruction needs to get a value from registers or data buses, but the value is an arbitrary one at that time, this instruction will be neglected.

The assumption makes the behaviour specification more concise than a complete one and, more importantly, makes the simulation very easy due to the omission of setting these arbitrary values explicitly. On the other hand, a complete specification is more precise than this one. It would specify all the output states of the sub-CPU, including those corresponding to the initial undetermined inputs.

```
divert(−1)
include(dill.m4)
divert

define(cpu_gates,‘IR,DtIn,Clk,MWire(4,R), Flag1,Flag0,BusAorMArr,BusBorDtOut,MW’)

define(cpu_pars,
  ‘dtIR,dtDtIn,dtClk,MWire(4,dtR), dtFlag1,dtFlag0,dtBusAorMArr,dtBusBorDtOut,dtMW’)


circuit(‘cpu_beha[cpu_gates]’,‘
  cpu_beha[cpu_gates]

where

process cpu_beha[cpu_gates] : noexit :=
 cpu_behaAux[cpu_gates]
    (bit(X)#X#X#X#X#X#X#X#X,
     bit(X)#X#X#X,
     X of bit,
     bit(X)#X#X#X,
     bit(X)#X#X#X,
     bit(X)#X#X#X,
     bit(X)#X#X#X,
     X of bit, X of bit,
     bit(X)#X#X#X,
     bit(X)#X#X#X,
     X of bit)

 where
 process cpu_behaAux[cpu_gates]
                 (dtIR,dtDtIn : BitArray,dtClk: Bit,
                  MWire(4,dtR) :BitArray,dtFlag1,dtFlag0 : Bit,
                  dtBusAorMArr,dtBusBorDtOut :BitArray,dtMW: Bit) : noexit :=

  IR ? dtIR: BitArray;
  cpu_behaAux[cpu_gates](cpu_pars)
 []
  DtIn ? dtDtIn: BitArray;
  cpu_behaAux[cpu_gates](cpu_pars)
 []
  Clk ? newdtClk: Bit;
  (
   [(dtClk ne 0 of bit) or (newdtClk ne 1 of bit) or (dtIR eq bit(x))] ->
     cpu_behaAux[cpu_gates](Pars(dtClk,‘cpu_pars’))
```

```
[]
 [(dtClk eq 0 of bit) and (newdtClk eq 1 of bit) and (dtIR ne bit(x))] ->
  (
      [(dtIR.natnum(7) eq 0 of bit) and (dtIR.natnum(6) eq 1 of bit)] ->(* store *)
       (
        cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
       ) (* end of store *)
     []
      [(dtIR.natnum(7) eq 1 of bit) and (dtIR.natnum(6) eq 0 of bit)] ->(* load *)
       (
        [(dtDtIn eq bit(x)) and (dtIR.natnum(8) eq 0 of bit) ] ->
         cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
        []
         [dtDtIn ne bit(x) or (dtIR.natnum(8) eq 1 of bit) ] ->
          ( let newreg: BitArray =
               bittoba(dtIR.natnum(3), dtIR.natnum(2), dtIR.succ(0), dtIR.0) in
           (
           [(dtIR.natnum(5) eq 0 of bit) and (dtIR.natnum(4) eq 0 of bit)] ->
            (
            [(dtR0 ne dtDtIn) and (dtIR.natnum(8) eq 0 of bit)] ->
            R0 ! dtDtIn;
            ( let dtR0: BitArray = dtDtIn in
            cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
            )
           []
            [(dtR0 eq dtDtIn) and (dtIR.natnum(8) eq 0 of bit)] ->
            cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
           []
            [(dtIR.natnum(8) eq 1 of bit) and (dtR0 ne newreg)] ->
               R0 ! newreg;
               ( let dtR0 : BitArray = newreg in
                 cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
               )
           []
            [(dtIR.natnum(8) eq 1 of bit) and (dtR0 eq newreg)] ->
               cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
           )
          []
          [(dtIR.natnum(5) eq 0 of bit) and (dtIR.natnum(4) eq 1 of bit)] ->
          (
           [(dtR1 ne dtDtIn) and (dtIR.natnum(8) eq 0 of bit)] ->
            R1 ! dtDtIn;
            ( let dtR1 : BitArray = dtDtIn in
              cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
            )
           []
            [(dtR1 eq dtDtIn) and (dtIR.natnum(8) eq 0 of bit)] ->
            cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
           []
              [(dtIR.natnum(8) eq 1 of bit) and (dtR1 ne newreg)] ->
               R1 ! newreg;
               ( let dtR1 : BitArray = newreg in
                 cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
               )
```

```
      []
       [(dtIR.natnum(8) eq 1 of bit) and (dtR1 eq newreg)] ->
         cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
      )
   []
    [(dtIR.natnum(5) eq 1 of bit) and (dtIR.natnum(4) eq 0 of bit)] ->
     (
      [(dtIR.natnum(8) eq 0 of bit) and (dtR2 ne dtDtIn)] ->
       R2 ! dtDtIn;
       ( let dtR2 : BitArray = dtDtIn in
       cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
       )
      []
       [(dtIR.natnum(8) eq 0 of bit) and (dtR2 eq dtDtIn)] ->
       cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
      []
        [(dtIR.natnum(8) eq 1 of bit) and (dtR2 ne newreg)] ->
         R2 ! newreg;
         ( let dtR2 : BitArray = newreg in
         cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
         )
      []
       [(dtIR.natnum(8) eq 1 of bit) and (dtR2 eq newreg)] ->
         cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
      )
   []
    [(dtIR.natnum(5) eq 1 of bit) and (dtIR.natnum(4) eq 1 of bit)] ->
     (
      [(dtIR.natnum(8) eq 0 of bit) and (dtR3 ne dtDtIn)] ->
       R3 ! dtDtIn;
       ( let dtR3 : BitArray = dtDtIn in
       cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
       )
      []
       [(dtIR.natnum(8) eq 0 of bit) and (dtR3 eq dtDtIn)] ->
        cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
      []
        [(dtIR.natnum(8) eq 1 of bit) and (dtR3 ne newreg)] ->
         R3 ! newreg;
         ( let dtR3 : BitArray = newreg in
         cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
         )
      []
       [(dtIR.natnum(8) eq 1 of bit) and (dtR3 eq newreg)] ->
         cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
      )
     ) (* end of let *)
   ) (* end of dtDtIn ne bit(x) *)
  ) (* end of load *)
[]
 [((dtIR.natnum(7) eq 0 of bit) and (dtIR.natnum(6) eq 0 of bit)) or
   ((dtIR.natnum(7) eq 1 of bit) and (dtIR.natnum(6) eq 1 of bit)) ] ->
   (
    (
```

```
    ([(dtIR.succ(0) eq 0 of bit) and (dtIR.0 eq 0 of bit)] ->
       exit(any BitArray, dtR0)
   []
     [(dtIR.succ(0) eq 0 of bit) and (dtIR.0 eq 1 of bit)] ->
       exit(any BitArray, dtR1)
   []
     [(dtIR.succ(0) eq 1 of bit) and (dtIR.0 eq 0 of bit)] ->
       exit(any BitArray, dtR2)
   []
     [(dtIR.succ(0) eq 1 of bit) and (dtIR.0 eq 1 of bit)] ->
       exit(any BitArray, dtR3)
    )
  |||
   ([(dtIR.natnum(3) eq 0 of bit) and (dtIR.natnum(2) eq 0 of bit)] ->
       exit(dtR0, any BitArray)
  []
    [(dtIR.natnum(3) eq 0 of bit) and (dtIR.natnum(2) eq 1 of bit)] ->
    exit(dtR1, any BitArray)
  []
    [(dtIR.natnum(3) eq 1 of bit) and (dtIR.natnum(2) eq 0 of bit)] ->
     exit(dtR2, any BitArray)
  []
   [(dtIR.natnum(3) eq 1 of bit) and (dtIR.natnum(2) eq 1 of bit)] ->
     exit(dtR3, any BitArray)
  )
 ) (* end of interleaving *)
>> accept dtBusAorMArr, dtBusBorDtOut : BitArray in
 (
  [(dtBusAorMArr eq bit(x)) or ( dtBusBorDtOut eq bit(x))] ->
   cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
 []
  [(dtBusAorMArr ne bit(x)) and (dtBusBorDtOut ne bit(x))] ->
   (
    [(dtIR.natnum(7) eq 0 of bit) and (dtIR.natnum(6) eq 0 of bit)] -> (* cmp *)
    (
     [dtBusAorMArr eq dtBusBorDtOut] ->
      ([dtFlag0 ne 1 of bit] ->
        Flag0 ! 1 of bit;
        ( let dtFlag0 : Bit = 1 of bit in
            cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
        )
      []
        [dtFlag0 eq 1 of bit] ->
         cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
      )
     []
     [dtBusAorMArr ne dtBusBorDtOut] ->
      ([dtFlag0 ne 0 of bit] ->
         Flag0 ! 0 of bit;
         ( let dtFlag0 : Bit = 0 of bit in
           cpu_behaAux[cpu_gates](Pars(dtClk,'cpu_pars'))
         )
      []
        [dtFlag0 eq 0 of bit] ->
```

```
      cpu_behaAux[cpu_gates](Pars(dtClk,‘cpu_pars’))
     )
    ) (* end of cmp *)

[]
 [(dtIR.natnum(Dec(7)) eq 1 of bit) and (dtIR.natnum(6) eq 1 of bit)] ->(* add *)
  ( let newdtDr : BitArray = dtBusAorMArr + dtBusBorDtOut,
         newdtCarry : Bit = carry(dtBusAorMarr,dtBusBorDtOut) in
   (
    (
     [(dtIR.natnum(5) eq 0 of bit) and (dtIR.natnum(4) eq 0 of bit)] ->
      (
       [dtR0 ne newdtDr] ->
        R0 ! newdtDr;
        exit( dtR3,dtR2,dtR1,newdtDr, any bit)
      []
      [dtR0 eq newdtDr ] ->
       exit( dtR3,dtR2,dtR1,newdtDr, any bit)
      )
  []
   [(dtIR.natnum(5) eq 0 of bit) and (dtIR.natnum(4) eq 1 of bit)] ->
   (
    [dtR1 ne newdtDr] ->
     R1 ! newdtDr;
     exit( dtR3,dtR2,newdtDr,dtR0, any bit)
    []
     [dtR1 eq newdtDr] ->
     exit( dtR3,dtR2,newdtDr,dtR0, any bit)
     )
 []
  [(dtIR.natnum(5) eq 1 of bit) and (dtIR.natnum(4) eq 0 of bit)] ->
   (
     [dtR2 ne newdtDr] ->
      R2 ! newdtDr;
      exit( dtR3,newdtDr,dtR1,dtR0, any bit)
    []
     [dtR2 eq newdtDr] ->
      exit( dtR3,newdtDr,dtR1,dtR0, any bit)
    )
  []
   [(dtIR.natnum(5) eq 1 of bit) and (dtIR.natnum(4) eq 1 of bit)] ->
    (
     [dtR3 ne newdtDr] ->
      R3 ! newdtDr;
      exit( newdtDr,dtR2,dtR1,dtR0, any bit)
    []
     [dtR3 eq newdtDr] ->
      exit( newdtDr,dtR2,dtR1,dtR0, any bit)
     )
    ) (* end of register changing *)
   |||
    (
     [dtFlag1 ne newdtCarry] ->
      Flag1 ! newdtCarry;
```

```
                  exit(MWire(4,any BitArray=),newdtCarry)
                 []
                  [dtFlag1 eq newdtCarry] ->
                     exit(MWire(4,any BitArray=),newdtCarry)
               ) (* end of flag1 changing *)
              ) (* end of let *)
             >>
              accept MWire(4,dtR) : BitArray, dtFlag1 : Bit in
               cpu_behaAux[cpu_gates](Pars(dtClk,‘cpu_pars’))
            ) (* end of add *)
           ) (* end of sourceA and sourceB ne bit(x) *)
          ) (* end of accept of BusA and BusB *)
        ) (* end of add or cmp *)
      ) (* end of appropriate transition *)
    ) (* end of Clk *)
   []
    [(((dtIR.natnum(7) eq 0 of bit) and (dtIR.natnum(6) eq 0 of bit)) or
     ((dtIR.natnum(7) eq 1 of bit) and (dtIR.natnum(6) eq 0 of bit)) or
     ((dtIR.natnum(7) eq 1 of bit) and (dtIR.natnum(6) eq 1 of bit))) and
     (dtMW ne 0 of bit)] ->
      MW ! 0 of bit;
      (let dtMW : bit = 0 of bit in
      cpu_behaAux[cpu_gates](cpu_pars)
      )
   []
    [(dtIR.natnum(Dec(7)) eq 0 of bit) and (dtIR.natnum(6) eq 1 of bit)
      and (dtMW ne 1 of bit) ] -> (*store*)
        MW ! 1 of bit;
       ( let dtMW : Bit = 1 of bit in
        cpu_behaAux[cpu_gates](cpu_pars)
        )
   []
    [(dtIR.natnum(3) eq 0 of bit) and (dtIR.natnum(2) eq 0 of bit) and
     (dtR0 ne dtBusAorMArr) and (dtR0 ne bit(X)) ] ->
       BusAorMArr ! dtR0;
       ( let dtBusAorMArr : BitArray = dtR0 in
          cpu_behaAux[cpu_gates](cpu_pars)
       )
   []
    [(dtIR.natnum(3) eq 0 of bit) and (dtIR.natnum(2) eq 1 of bit) and
      (dtR1 ne dtBusAorMArr) and (dtR1 ne bit(X))] ->
        BusAorMArr ! dtR1;
        (let dtBusAorMArr : BitArray = dtR1 in
          cpu_behaAux[cpu_gates](cpu_pars)
        )
    []
     [(dtIR.natnum(3) eq 1 of bit) and (dtIR.natnum(2) eq 0 of bit) and
      (dtR2 ne dtBusAorMArr) and (dtR2 ne bit(X))] ->
        BusAorMArr ! dtR2;
        (let dtBusAorMArr : BitArray = dtR2 in
          cpu_behaAux[cpu_gates](cpu_pars)
        )
    []
     [(dtIR.natnum(3) eq 1 of bit) and (dtIR.natnum(2) eq 1 of bit) and
```

```
    (dtR3 ne dtBusAorMArr) and (dtR3 ne bit(X))] ->
     BusAorMArr ! dtR3;
     (let dtBusAorMArr : BitArray = dtR3 in
       cpu_behaAux[cpu_gates](cpu_pars)
      )
 []
  [(dtIR.succ(0) eq 0 of bit) and (dtIR.0 eq 0 of bit) and
      (dtR0 ne dtBusBorDtOut) and (dtR0 ne bit(X))] ->
       BusBorDtOut ! dtR0 ;
       (let dtBusBorDtOut : BitArray = dtR0 in
        cpu_behaAux[cpu_gates](cpu_pars)
       )
 []
  [(dtIR.succ(0) eq 0 of bit) and (dtIR.0 eq 1 of bit) and
    (dtR1 ne dtBusBorDtOut) and (dtR1 ne bit(X))] ->
     BusBorDtOut ! dtR1 ;
     (let dtBusBorDtOut : BitArray = dtR1 in
      cpu_behaAux[cpu_gates](cpu_pars)
     )

 []
  [(dtIR.succ(0) eq 1 of bit) and (dtIR.0 eq 0 of bit) and
  (dtR2 ne dtBusBorDtOut) and (dtR2 ne bit(X))] ->
    BusBorDtOut ! dtR2 ;
    (let dtBusBorDtOut : BitArray = dtR2 in
      cpu_behaAux[cpu_gates](cpu_pars)
     )
 []
  [(dtIR.succ(0) eq 1 of bit) and (dtIR.0 eq 1 of bit) and
    (dtR3 ne dtBusBorDtOut) and (dtR3 ne bit(X))] ->
     BusBorDtOut ! dtR3 ;
     (let dtBusBorDtOut : BitArray = dtR3 in
      cpu_behaAux[cpu_gates](cpu_pars)
     )
 endproc (* cpu_behaAux *)
endproc (* cpu_beha*)
')
```

F Structural Specification of the Sub-CPU

```
divert(−1)
include(dill.m4)
divert

define(cpu_gates, ‘MWire(9,IR),MWire(4,DtIn),Clk,MWire(4,‘R3,R2,R1,R0’),Flag1,Flag0,
                  MWire(4,‘BusAorMArr,BusBorDtOut ’),MW’)

circuit(‘cpu_abst[cpu_gates]’,‘
hide RW,AorL,MorF,LI,Ls0,ADD,CMP, Gr0,Ls4,Gr4, Carry, Zero,C0,Clr,
      MWire(4,Sum),MWire(4,G),MWire(4, ‘Dr,SA,SB’),
      MWire(4,‘AddA,AddB,ComA,ComB’),MWire(4,Data),MWire(4,RegIn) in


 (
   CtrlWord_WB[IR8,IR7,IR6,RW, AorL,MW, MorF, LI, ADD, CMP]
  |||
   Decoder2_WB_One[IR5,IR4, MWire(4,Dr)]
  |||
   Decoder2_WB_One[IR3,IR2, MWire(4,SA)]
  |||
   Decoder2_WB_One[IR1,IR0, MWire(4,SB)]
 )

 |[RW,AorL,MorF,LI,ADD,CMP,MWire(4,‘Dr,SA,SB’), IR3,IR2,IR1,IR0]|
 ((One[Clr]
  |[Clr]|
   ( (RegLoadClr4_BB_Pos[MWire(4,RegIn),G3,Clr,Clk,MWire(4,R3)] |[G3]| And2[ RW,Dr3,G3])
    |[MWire(4,RegIn),Clk,RW,Clr]|
     (RegLoadClr4_BB_Pos[MWire(4,RegIn),G2,Clr,Clk,MWire(4,R2)] |[G2]| And2[ RW,Dr2,G2])
    |[MWire(4,RegIn),Clk,RW,Clr]|
     (RegLoadClr4_BB_Pos[MWire(4,RegIn),G1,Clr,Clk,MWire(4,R1)] |[G1]| And2[ RW,Dr1,G1])
    |[MWire(4,RegIn),Clk,RW,Clr]|
     (RegLoadClr4_BB_Pos[MWire(4,RegIn),G0,Clr,Clk,MWire(4,R0)] |[G0]| And2[ RW,Dr0,G0])
   )
  )
 |[MWire(4,R3),MWire(4,R2),MWire(4,R1),MWire(4,R0),MWire(4,RegIn),Clk,Clr]|
  (
   ((
     Repeater_One_4[MWire(4,R3),SA3,MWire(4,BusAorMArr)]
    |[MWire(4,BusAorMArr)]|
     Repeater_One_4[MWire(4,R2),SA2,MWire(4,BusAorMArr)]
    |[MWire(4,BusAorMArr)]|
     Repeater_One_4[MWire(4,R1),SA1,MWire(4,BusAorMArr)]
    |[MWire(4,BusAorMArr)]|
     Repeater_One_4[MWire(4,R0),SA0,MWire(4,BusAorMArr)]
   )
   |[MWire(4,R3),MWire(4,R2),MWire(4,R1),MWire(4,R0)]|
    (
     Repeater_One_4[MWire(4,R3),SB3,MWire(4,BusBorDtOut)]
    |[MWire(4,BusBorDtOut)]|
     Repeater_One_4[MWire(4,R2),SB2,MWire(4,BusBorDtOut)]
    |[MWire(4,BusBorDtOut)]|
```

```
      Repeater_One_4[MWire(4,R1),SB1,MWire(4,BusBorDtOut)]
    |[MWire(4,BusBorDtOut)]|
      Repeater_One_4[MWire(4,R0),SB0,MWire(4,BusBorDtOut)]
   ))
   |[MWire(4,BusAorMArr),MWire(4,BusBorDtOut)]|
   (
    (
     (MComp(4, AorL=, 'Demultiplexer1to2_WB[BusAorMArr,AorL=,AddA,ComA]')
    |[AorL]|
      MComp(4, AorL=, 'Demultiplexer1to2_WB[BusBorDtOut,AorL=,AddB,ComB]')
     )
    |[MWire(4,'AddA,ComA,AddB,ComB')]|

      (
       ((Zero[C0] |[C0]| Adder4_BB[MWire(4,AddA),MWire(4,AddB),C0,MWire(4,Sum),Carry])
      |||
       ((Zero[Ls4] ||| Zero[Gr4])
      |[Ls4,Gr4]|
       Comparator4_BB[Ls4,Gr4,MWire(4,'ComA,ComB'),Ls0,Gr0]
       |[Ls0,Gr0]|
       Or2[Ls0,Gr0,Zero]
       ))
      |[Carry,Zero,MWire(4,Sum)]|
      (
       (
         RegLoadClr_BB_Pos [Carry,ADD,Clr,Clk,Flag1 ]
       |[Clk,Clr]|
          RegLoadClr_BB_Pos[Zero,CMP,Clr,Clk,Flag0 ]
       )
      |||
      (
        MComp(4,MorF=,'Mul2to1WB[Sum, DtIn, MorF=,Data]')
       |[MWire(4,Data)]|
        MComp(4,LI=,'Multiplexer2to1_WB[Data,IR,LI=,RegIn]')
      )
     )
    )
   )
  )
 )
)
```

where

Decoder2_WB_One_Decl
RegLoadClr4_BB_Pos_Decl
Repeater_One_4_Decl
Demultiplexer1to2_WB_Decl
Zero_Decl
One_Decl
Or2_Decl
Adder4_BB_Decl
Comparator4_BB_Decl
And2_Decl

```
DFlipFlop_BB_Pos_Decl
Multiplexer2to1_WB_Decl


define(ctrlword,‘IR8,IR7,IR6,RW,AorL,MW,MorF,LI,ADD,CMP’)
define(ctrlpars,‘dtIR8,dtIR7,dtIR6,dtRW,dtAorL,dtMW,dtMorF,dtLI,dtADD,dtCMP’)

process CtrlWord_WB[ctrlword] : noexit :=
  CtrlWord_WB_Aux[ctrlword](MWire(10, X of bit=))

 where

 process CtrlWord_WB_Aux[ctrlword]
  (dtIR8,dtIR7,dtIR6,dtRW,dtAorL,dtMW,dtMorF,dtLI,dtADD,dtCMP: Bit) : noexit :=
  IR8 ? dtIR8 : Bit;
  CtrlWord_WB_Aux[IR8,IR7,IR6,RW,AorL,MW,MorF,LI,ADD,CMP](ctrlpars)
   []
  IR7 ? dtIR7 : Bit;
  CtrlWord_WB_Aux[ctrlword](ctrlpars)
 []
  IR6 ? dtIR6 : Bit;
  CtrlWord_WB_Aux[ctrlword](ctrlpars)
 []
  (let newdtRW : Bit = dtIR7,
       newdtAorL: Bit = dtIR6,
       newdtLI : Bit = dtIR8 and (dtIR7 and not(dtIR6)),
       newdtMW : Bit = not(dtIR7) and dtIR6,
       newdtMorF: Bit = (dtIR7 xor dtIR6) xor (not(dtIR7) and dtIR6),
       newdtADD : Bit = dtIR7 and dtIR6,
       newdtCMP : Bit = not(dtIR7 )and not(dtIR6) in
   (
    [(newdtRW ne X of bit) and (newdtRW ne dtRW)] ->
     RW ! newdtRW;
     CtrlWord_WB_Aux[ctrlword](Pars(dtRW,‘ctrlpars’))
   []
    [(newdtAorL ne X of bit) and (newdtAorL ne dtAorL)] ->
     AorL ! newdtAorL;
     CtrlWord_WB_Aux[ctrlword](Pars(dtAorL,‘ctrlpars’))
   []
    [(newdtMW ne X of bit) and (newdtMW ne dtMW)] ->
     MW ! newdtMW;
     CtrlWord_WB_Aux[ctrlword](Pars(dtMW,‘ctrlpars’))
   []
    [(newdtMorF ne X of bit) and (newdtMorF ne dtMorF)] ->
     MorF ! newdtMorF;
     CtrlWord_WB_Aux[ctrlword](Pars(dtMorF,‘ctrlpars’))
   []
    [(newdtLI ne X of bit) and (newdtLI ne dtLI)] ->
      LI ! newdtLI;
     CtrlWord_WB_Aux[ctrlword](Pars(dtLI,‘ctrlpars’))
   []
    [(newdtADD ne X of bit) and (newdtADD ne dtADD)] ->
      ADD ! newdtADD;
     CtrlWord_WB_Aux[ctrlword](Pars(dtADD,‘ctrlpars’))
```

```
    []
     [(newdtCMP ne X of bit) and (newdtCMP ne dtCMP)] ->
       CMP ! newdtCMP ;
       CtrlWord_WB_Aux[ctrlword](Pars(dtCMP,‘ctrlpars’))
    )
   )
 endproc
 endproc

’)
```

G Execution Path of An instruction Sequence of the Sub-CPU

In this section we list the execution path of the instruction sequence:

```
Load  R0, 0000
Load  R1,0011
Load  R2,1110
Load  R3,0011
Add   R0, R1,R2
Cmp   R0,R2
Cmp   R1,R3
Load  R3,0000
Store R3,R1
Load  R2,R3
```

for both the behaviour specification and the structural specification. The lines beginning % are comments.

G.1 Execution Path for the Behaviour Specification

```
  0     START

  %     Load  R0, 0000

  1     IR  !((((((((Bit(1) # 1) # 0) # 0) # 0) # 0) # 0) # 0) # 0)
  2     MW  !0
  3     Clk  !0
  4     Clk  !1
  5     R0  !(((Bit(0) # 0) # 0) # 0)
  6     BusAorMArr  !(((Bit(0) # 0) # 0) # 0)
  7     BusBorDtOut  !(((Bit(0) # 0) # 0) # 0)

  %     Load  R1,0011

  8     IR  !((((((((Bit(1) # 1) # 0) # 0) # 1) # 0) # 0) # 1) # 1)
  9     Clk  !0
 10     Clk  !1
 11     R1  !(((Bit(0) # 0) # 1) # 1)

 %      Load  R2,1110

 12     IR  !((((((((Bit(1) # 1) # 0) # 1) # 0) # 1) # 1) # 1) # 0)
 13     Clk  !0
 14     Clk  !1
 15     R2  !(((Bit(1) # 1) # 1) # 0)
 16     BusBorDtOut  !(((Bit(1) # 1) # 1) # 0)

 %      Load  R3,0011

 17     IR  !((((((((Bit(1) # 1) # 0) # 1) # 1) # 0) # 0) # 1) # 1)
 18     Clk  !0
 19     Clk  !1
 20     R3  !(((Bit(0) # 0) # 1) # 1)
 21     BusBorDtOut  !(((Bit(0) # 0) # 1) # 1)

 %      Add   R0, R1,R2

 22     IR  !((((((((Bit(0) # 1) # 1) # 0) # 0) # 0) # 1) # 1) # 0)
 23     BusAorMArr  !(((Bit(0) # 0) # 1) # 1)
 24     BusBorDtOut  !(((Bit(1) # 1) # 1) # 0)
```

```
25    Clk  !0
26    Clk  !1
28    Flag1  !1
29    R0  !(((Bit(0) # 0) # 0) # 1)

%     Cmp   R0,R2

31    IR  !((((((((Bit(0) # 0) # 0) # 0) # 0) # 0) # 0) # 1) # 0)
32    BusAorMArr  !(((Bit(0) # 0) # 0) # 1)
33    Clk  !0
34    Clk  !1
36    Flag0  !0

%     Cmp   R1,R3

37    IR  !((((((((Bit(0) # 0) # 0) # 0) # 0) # 0) # 1) # 1) # 1)
38    BusAorMArr  !(((Bit(0) # 0) # 1) # 1)
39    BusBorDtOut  !(((Bit(0) # 0) # 1) # 1)
40    Clk  !0
41    Clk  !1
43    Flag0  !1

%     Load  R3,0000

44    IR  !((((((((Bit(1) # 1) # 0) # 1) # 1) # 0) # 0) # 0) # 0)
45    BusAorMArr  !(((Bit(0) # 0) # 0) # 1)
46    BusBorDtOut  !(((Bit(0) # 0) # 0) # 1)
47    Clk  !0
48    Clk  !1
49    R3  !(((Bit(0) # 0) # 0) # 0

%   Store R3,R1

50    IR  !((((((((Bit(0) # 0) # 1) # 0) # 0) # 1) # 1) # 0) # 1)
51    MW  !1
52    BusAorMArr  !(((Bit(0) # 0) # 0) # 0)
53    BusBorDtOut  !(((Bit(0) # 0) # 1) # 1)

%     Load  R2,R3

54    IR  !((((((((Bit(0) # 1) # 0) # 1) # 0) # 1) # 1) # 0) # 0)
55    MW  !0
56    BusBorDtOut  !(((Bit(0) # 0) # 0) # 1)
57    DtIn  !(((Bit(0) # 0) # 1) # 1)
58    Clk !0
59    Clk  !1
60    R2  !(((Bit(0) # 0) # 1) # 1)
```

G.2 Execution Path for the Structural Specification

```
 0    START

 %    Load R0,0000

 5    IR8  !1
 6    IR7  !1
 7    IR6  !0
```

```
  8    IR5  !0
  9    IR4  !0
 10    IR3  !0
 11    IR2  !0
 12    IR1  !0
 13    IR0  !0
 14    MW  !0
 54    Clk  !0
 55    Clk  !1
 56    R02  !0
 57    R03  !0
 58    R00  !0
 59    R01  !0
 60    BusAorMArr3  !0
 61    BusBorDtOut3  !0
 62    BusAorMArr0  !0
 63    BusBorDtOut0  !0
 64    BusAorMArr1  !0
 65    BusBorDtOut1  !0
 66    BusAorMArr2  !0
 67    BusBorDtOut2  !0

 %     Load R1, 0011

 79    IR4  !1
 80    IR1  !1
 81    IR0  !1
 90    Clk  !0
 91    Clk  !1
 92    R13  !0
 93    R12  !0
 94    R11  !1
 95    R10  !1

 %     Load R2,1110

 96    IR5  !1
 97    IR4  !0
 98    IR3  !1
 99    IR2  !1
100    IR0  !0
112    Clk  !0
113    Clk  !1
114    R23  !1
115    R22  !1
116    R21  !1
117    R20  !0
118    BusBorDtOut3  !1
119    BusBorDtOut1  !1
120    BusBorDtOut2  !1

%      Load R3, 0011

126    IR4  !1
127    IR3  !0
128    IR2  !0
129    IR0  !1
141    Clk  !0
```

```
142     Clk  !1
143     R33  !0
144     R32  !0
145     R31  !1
146     R30  !1
147     BusBorDtOut3  !0
148     BusBorDtOut2  !0
149     BusBorDtOut0  !1

%       ADD R0,R1,R2

153     IR8  !0
154     IR6  !1
155     IR5  !0
156     IR4  !0
157     IR2  !1
158     IR0  !0
171     BusAorMArr0  !1
172     BusBorDtOut3  !1
173     BusAorMArr1  !1
174     BusBorDtOut0  !0
175     BusBorDtOut2  !1
195     Clk  !0
196     Clk  !1
197     Flag1  !1
198     R00  !1

%       CMP R0,R2

199     IR7  !0
200     IR6  !0
201     IR2  !0
205     IR0  !0
209     BusAorMArr1  !0
234     Clk  !0
235     Clk  !1
236     Flag0  !1

%       CMP R1,R3

237     IR2  !1
238     IR0  !1
243     BusBorDtOut3  !0
244     BusAorMArr1  !1
245     BusBorDtOut2  !0
246     BusBorDtOut0  !1
253     Clk  !0
254     Clk  !1
255     Flag0  !0

%       Load R3,0000

256     IR8  !1
257     IR7  !1
258     IR5  !1
259     IR4  !1
260     IR2  !0
261     IR1  !0
```

```
262     IR0  !0
274     BusAorMArr1  !0
275     BusBorDtOut1  !0
278     Clk  !0
279     Clk  !1
280     R30  !0
281     R31  !0

%       Store R3,R1

282     IR8  !0
283     IR7  !0
284     IR6  !1
285     IR5  !0
286     IR4  !0
287     IR3  !1
288     IR2  !1
289     IR0  !1
291     MW  !1
302     BusAorMArr0  !0
303     BusBorDtOut1  !1
316     Clk  !0
317     Clk  !1

%       Load R2,R3

318     IR7  !1
319     IR6  !0
320     IR5  !1
321     IR0  !0
323     MW  !0
331     BusBorDtOut1  !0
339     DtIn3  !0
340     DtIn2  !0
341     DtIn1  !1
342     DtIn0  !1
345     Clk  !0
346     Clk  !1
347     R22  !0
348     R23  !0
349     R20  !1
```