ADL : AN HIERARCHICAL LOGIC DESIGN LANGUAGE

Hilary J. Kahn, A. K. Burston and D. J. Kinniment
Department of Computer Science
University of Manchester
U.K.

1. INTRODUCTION

The use of Computer Aided Design techniques in the design of computer systems themselves is already well established, as can be observed from the widespread use of programs for the layout of PCBs and IC masks, automated production of design documentation and the use of automated manufacturing techniques [1]. Simulation, particularly digital component level simulation, has also proved an extremely useful tool in the development of computer systems [2]. The most successful of the CAD tools available have, inevitably, been at the lower (i.e. manufacturing) end of the design process where the problems are well understood and amenable to algorithmic solution. In the important area of test pattern generation and at the higher levels of the design process, the tools available have proved depressingly inadequate.

The Department of Computer Science, University of Manchester, has, for a number of years, specialised in the design of large, fast, computer systems, e.g. Atlas [3] and MU5 [4], and has considerable experience of the practical difficulties in large system design, manufacture and maintenance. Following the successful use of CAD techniques in the design of MU5, a more comprehensive design system is currently under development. The main emphasis in the design of the CAD tools has been the provision of practical, usable (and hence used) tools rather than a more generalised system. It is felt that greater generality would require more time, manpower and computer resources than are readily available, and might still prove unsatisfactory in solving the real problems faced by the hardware engineer.

This CAD system is centred on a formal Data Base Management System, MUD [5,6] with non-programmer access via a flexible Command Processor [7]. The aim is to provide

- a high level hardware design language (ADL) translatable by machine into logic;
- a system level simulator associated with ADL;
- a lower level design language allowing 'hand-designed' logic to be incorporated;
- a gate level logic simulator which uses component models developed in a specialised logic description language;
- layout systems;
- documentation aids including a logic diagram package.

Part of the system is already in operation; much of the rest is currently under development.

1.1. Constraints on ADL

The main requirement of ADL (A Design Language!) is that it should be capable of describing large, fast, asynchronous systems in which a high degree of parallelism is inevitably present. At the same time, a hardware engineer using ADL would expect the system to automatically generate logic which used the highest speed technology readily available. The logic generated should, of course, be efficient both in time and volume. In order to help inter-designer communication, it is also intended that ADL should provide good design documentation.

ADL aims to provide a useful aid while still permitting the designer some freedom to develop new approaches when faced with new problems. Furthermore, the constructs of the language have deliberately been kept closely related to practical hardware implementations to enable the designer to have a 'feel' for the actual logic which will eventually be generated. This approach it is hoped will permit the skill of experienced designers to be used to the full.

1.2. Formal Design Methods

A number of high-level design systems [8] already exist. Some, such as ISP and PMS are more properly logic description systems and are too high level to be of practical use for automatic logic generation. Others, e.g. DDL, are aimed at synchronous or serial systems.

Although the two graphical approaches, Petri Nets [9] and LOGOS [10] are suited to parallel, asynchronous system design, they appear impractical for large systems. The pictorial approach makes it very difficult to examine more than a small part of the design at any one time. In addition, LOGOS, which uses two graphs - a data graph and a control graph - seems to need a significant amount of extra text to cross reference between the graphs.

2. THE STRUCTURE OF AN ADL DESIGN

A design expressed in ADL is hierarchical in that it is a block definition. Within that block definition reference may be made to constituent blocks (called subblocks) which may or may not already be in existence. Naturally, these subblocks can themselves be defined as ADL blocks with their own constituent subblocks. The lowest level of the hierarchy consists of basic subblocks such as registers, decoders, etc. Design of these low-level constituents is best done at the gate level rather than in ADL as they are conceptually simple and should be represented by the most efficient logic possible. An extendable library of these basic subblocks is held in the data base and includes many of the common primitives required.

Within an ADL block there may exist a number of control paths operating in serial or parallel as required. Use of appropriate branch and synchronisation constructs permits paths to diverge and converge. A given path is divided into alternate task and control sequence sections; a task specifies the set of concurrent events which are to occur when the task is active and a control sequence determines which task(s) require activation when the current task is deactivated. A task remains active until the control signal combination for which the task waits has occurred. For example

```
T1 : 'FLOW' a ← b ,
            c ← d ;
     'SET'  sig3, sig4;            Task T1
     'WAIT  FOR' sig1 & sig2;
      → T5;                        Control sequence
```

Here, when T1 becomes active (as a result of a control transfer to it from some other task(s)), the data transfers from b to a and d to c are enabled and sig3 and sig4 set to '1'. This state persists until the control signals sig1 and sig2, are both set to '1' in response to the setting of sig3 and sig4. At this point, the task is deactivated and control is transferred unconditionally to T5.

3. ADL LANGUAGE CONSTRUCTS

The language features are summarised below; a more detailed description may be found in [11,12]. An example of the use of ADL is given in the appendix.

3.1. Static Declaration Section

An ADL block definition starts with a specification of the block interface and the interfaces of constituent subblocks. Note that a block interface is defined in terms of data ports and control signals. In addition, facilities exist to permit control signals local to the block to be defined as well as any invariant data paths. For example

```
'BLOCK' B ['INPUT' BIN [0:15]/BINREQ/BINACK/
           'OUTPUT' BOUT [0:7]
           'CONTROL IN' BA  'CONTROL OUT' BZ];
'SUBBLOCK' SB-
        SBOCC1 ['INPUT' SB1 [0:3], SB2 [-2:1] 'OUTPUT' SBO [0:7]
                'CONTROL IN' SBCA 'CONTROL OUT' SBCX];
'LOCAL CONTROL' LCA, LCB, LCC;
'CONNECTION'    BOUT ← SBO;
```

This defines a block B which has a 16-bit input port, BIN, 8-bit output port, BOUT, and four control signals, BINREQ, BINACK, BA, BZ. Note that BINREQ/BINACK are specifically used to control data flow to port BIN and form a request/acknowledge pair which can be used to provide a 'handshake' signalling system between blocks. A subblock of type SB with two 4-bit input ports, an 8-bit output port and two control signals may or may not have already have been defined; an occurrence SBOCC1 is used with interface names SB1, SB2, SBO, SBCA and SBCX.

The 'local control' construct is used to define control signals additonal to those defined as part of the interfaces of block B and subblock SBOCC1. LCA, LCB and LCC are available only within B and are useful in providing communication between various control paths and in controlling the internal timing of the block.

The 'connection' statement indicates that data ports SB0 and BOUT are to be permanently interconnected and hence no further transfer-enabling logic will be required. In its most general form, this construct can be used to define quite complex data port connections.

Any digital system must be set to a predefined state when initially 'powered up' in order to ensure correct operation. In ADL, it is assumed that a 'general reset' signal will exist and that all tasks and control signals will be made inactive unless specifically excluded by use of an INITIALIZE or INITIALIZE CONTROL statement.

3.2. Control Section

This section contains task definitions and control sequences.

3.2.1. Tasks

Tasks are delimited by a task label and a timing statement such as WAIT FOR

e.g.
```
T1 : statements
     'WAIT FOR' LCA + LCB;
```

The effect of the WAIT FOR is to suspend control within T1 until the appropriate signal state combination has occurred. Note that all control signals in ADL have an associated flag and that when the task is deactivated, once either LCA=1 or LCB=1, the flag for the relevant control signal is reset.

The most common statements which occur within a task are FLOW, SET and RESET which allow data paths and control signals to be modified. For example

(i) 'FLOW' SB1 ← BIN [12:15];
causes a temporary interconnection between a part of BIN and SB1. This interconnection is only enabled when the task in which the statement appears is active. Note that a single port may have data 'flowed' to it from many different sources at different times and logical operators may be used to combine data ports.

(ii) 'SET' LCA; or 'RESET' SBC1, LCB;
These permit explicit setting/resetting of control signals in order to enable communication between separate tasks.

3.2.2. Control Transfer

A range of control transfer instructions is available to permit control paths to branch conditionally or unconditionally. The most basic of these is →. For example

→T4 transfers control to task T4
→(T3, T9) transfers control to T3 and T9 simultaneously.

The control transfer may be made conditional by prefacing the '→' with 'IF' condition 'THEN'. For example

```
                    'IF' SB0 = 0 'THEN' → T8;
```

A 'no destination' statement, * , is available to terminate a control path.

A more complex conditional, DECODE, provides a parallel control path switch based on the state of a data port. For example

```
          'DECODE' BIN [3:6] → [T3, T5, T6, (T7, T10) ];
```

transfers control to all destinations for which the corresponding bit of BIN is at a logical 1.

The basic '→' statement and the simple conditional version may be made to operate in either parallel (//) or serial (#) mode. For example

```
          T1 : statements
                'WAIT FOR' condition 1;
             // → T4;
             #  'IF' condition 2 'THEN' → T3;
                → T2
```

Here T4 will always be activated; T3 will be activated if condition 2 is true otherwise control is transferred to T2.

The constructs discussed so far provide most of the facilities needed by the hardware designer. However, there remain two problem areas of considerable importance to the designer of complex, parallel systems; priority resolution and the control of mutually exclusive access to a shared resource.

3.2.3. Priority Handling

It is typical of parallel systems that a control path may be activated from a number of different positions. Assuming only one activation at a time is permitted, a decision must be made about which activation to allow. In ADL this is done by inserting a special priority mechanism in the control path. This mechanism consists of a PRIORITY WAIT which appears inside a task (instead of a WAIT FOR) and makes use of a PRIORITY BLOCK to do the decision making. For example

```
                    'PRIORITY BLOCK' PB-
                                     PBOCC  [3];
```

can be used to decide between three conflicting requests and might be accessed by

```
              T5 : 'PRIORITY WAIT' PBOCC-
                   P1 : 'WHEN' LCA 'THEN' → T6,
                   P2 : 'WHEN' LCB 'THEN  → T7,
                   P3 : 'WHEN' LCC 'THEN' → T8;
```

In this example, suppose T5 is active when one or more of the control signals occurs. The states of all three control signals are staticised and input to PBOCC together with a special 'make a decision' signal. After a delay, a 'decision made' signal will be generated and a wire corresponding to an active control signal will be set so that control can be passed to <u>one</u> of T6, T7 or T8. If required, data ports can be used by the priority block to alter the decision making criteria.

3.2.4. Mutual Exclusion

This problem is one of preventing simultaneous access to a shared resource by two or more parallel control paths. The ADL solution to this problem uses the hardware equivalent of a semaphore. A special controlled priority block which is used by two or more priority waits must be defined. The priority waits are at the start of the sections of control path concerned with accessing the shared resource. The sections are called the 'critical sections' of the relevant paths. For example

```
'CONTROLLED PRIORITY BLOCK' CPB-
            CPBOCC [2];

T1 : 'PRIORITY WAIT' CPBOCC-
      P1 : 'WHEN' LCA 'THEN'→ T2;
T41: 'PRIORITY WAIT' CPBOCC-
      P1 : 'WHEN' LCB 'THEN'→ T42;
```

Assuming both T1 and T41 are active, when either LCA or LCB is set a priority decision is made and control continues with T2 or T42. The other path is suspended and CPBOCC is 'locked'. When the active path no longer requires the common resource, it issues a release statement such as 'RELEASE' CPBOCC. Any outstanding requests to the priority block will then be considered.

4. IMPLEMENTATION

4.1. The ADL Translator

A logic design expressed in ADL is input to the translator which applies syntactic checks and produces as output an intermediate data structure (IDS) which can be used by a number of different programs including the logic generator. The IDS, which is stored in the data base, is a set of tables which are closely correlated with the original text except that certain duplication is avoided. For example, if the same flow statement appears in more than one task the flow information is stored once only so that the logic generator need only create one version of the logic.

4.2. The ADL Logic Generator

The logic generator uses the IDS to create the logic for an ADL block. It operates in two passes and requires that the interface details of the block and of any constituent subblocks be fully defined. These details include loading and fan-out constraints which can be input via the Command Processor.

During the first pass, the IDS is examined and idealised 'meta-logic' is produced. The main logic synthesis is carried out at this stage but practical constraints of fan-in and fan-out are ignored. The second pass of the logic generator transforms the meta-logic into the particular technology required and adjusts the gating to take account of fan-in, fan-out, inversion and loading. This approach, which is commonly used to aid portability in programming languages, is flexible and localises the effects of having to generate logic using a number of rapidly developing technologies. An example of part of the logic generated for the design given in the Appendix is shown in Fig.A1.

It should be noted that the ADL logic generator does not need to fill in the detailed logic for subblocks within the block being examined. A separate integrating program assembles a complete network from the logic information which is stored in the data base for each of the blocks and subblocks.

4.3. Meta-logic

ADL language constructs are represented by combinations of simple function modules which are, in principle, implementable in any technology. The full set of these meta-logic modules and some typical implementations are shown in Fig.1. The purpose of each module is summarised as follows :

Task	Initiates the functions (e.g. FLOWs, SETs) within a task and waits for task completion.
Signal	Provides a static flag to indicate the occurrence of a control signal.
If	Propagates one of two control paths depending on a data condition.
Decode	A group of decode modules selects a subset of control paths to propagate.
Edge buffer	Converts an edge to a level for testing.
Flip-flop and Delay	Used to staticise signals to be tested inside priority wait statements.

In addition, three basic gate types, AND, OR and EQUIVALENCE, which are assumed to have infinite fan-in, are available. Further details of the operation of the modules may be found in [7].

5. CONCLUSIONS

A number of experimental logic designs have been developed to test the system. Although the quality and efficiency of the generated logic is yet to be evaluated, experience indicates that ADL is convenient to use and provides a viable formalism for expressing the concepts of logic design.

Future work planned includes implementation of the logic integrating program, production of a system level simulator and an automatic diagram drawing package to provide graphical output to accompany an ADL design.

REFERENCES

[1] de Man H. :
"Computer Aided Design : Trying to Bridge the Gap"
European Solid State Circuits Conference, Amsterdam, 1978.

[2] Kahn H.J. and May J.W.R. :
"The Use of Logic Simulation in the Design of a Large Computer System"
Radio Electron. Eng., Vol.1, 497-503, 1973.

[3] Lavington S.H. :
"The Manchester Mark I and ATLAS - A historical perspective"
CACM, Vol.21, No.1, 1978.

[4] Ibbett R.N. and Capon P.C. :
"The Development of the MU5 Computer System"
CACM, Vol.21, No.1, 1978.

[5] Wilson T.B. :
"A Data Description Language"
M.Sc. Thesis, University of Manchester, 1974.

[6] Wilson T.B. :
"A Data Base Management System for the MU5 Computer"
Ph.D. Thesis, University of Manchester, 1976.

[7] Burston A.K. :
"An Integrated Logic Design System"
Ph.D. Thesis, University of Manchester (to be submitted).

[8] Special Issue on Hardware Description Languages
COMPUTER, December, 1974.

[9] Petri C.A.:
"Kommunikation mit Automaten"
Schriften des Rheinsch - West-Falischen Inst. fur Instrumentelle Math., Univ. Bonn, 1962.

[10] Rose C.W., Bradshaw F.T., Katze S.W. :
"The LOGOS Representation System"
IEEE Proc. COMPCON, September 1972.

[11] Burston A.K. :
"The Development of a Computer Logic Design Language"
M.Sc. Thesis, University of Manchester, 1975.

[12] Burston A.K., Kinniment D.J., Kahn H.J. :
"A Design Language for Asynchronous Logic"
Computer Journal, November 1978.

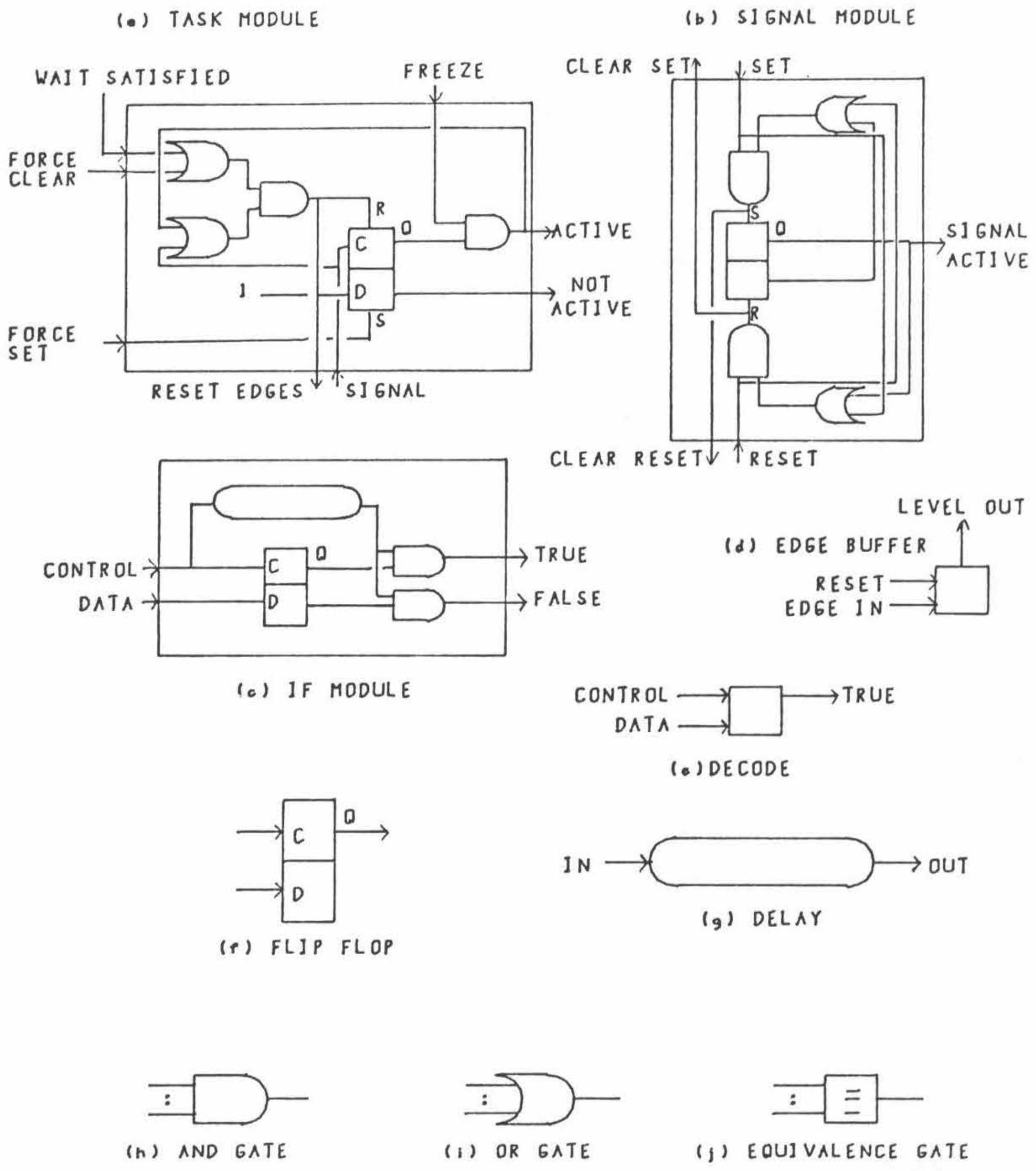

Figure 1 : Meta-logic Modules

Appendix - An Example of ADL

The example is the design of a subblock to perform the "compression" function. The operation is defined as follows: given two equal length bit vectors MASK and DATA a vector of less than or equal length, RESULT, is produced from DATA by suppressing all the bits of DATA for which a "0" appears in the corresponding position in MASK. For example:

```
DATA          10110101
MASK          01100110
RESULT         01  10      = 0110
```

The subblock works on 8 bit elements, unused bits of RESULT are zero filled. The output of the unit is buffered, that is, a second calculation can be performed whilst the result of the first is held on the output, ready to be accepted by the outer block. Handshake control is used throughout.

The overall operation of the unit consists of three parallel control paths. The first is the main loop (T1, T2, T4, T5, T6, T7, T8) which is concerned with shifting the data and setting RESULT. The second (T3) is concerned with counting the number of iterations performed. The third (T9) is concerned with buffering the output.

```
'BLOCK' COMPRESS['INPUT' MASKIN[0:7],DATAIN[0:7] 'OUTPUT' DATAOUT[0:7]
                 'CONTROL IN' GO, ACCEPTED 'CONTROL OUT' TAKEN, DONE];
!MASKIN - 8 bit input port for the MASK.
DATAIN - 8 bit input port for the DATA.
DATAOUT - 8 bit output port for the RESULT.
GO - start calculation, input data ready.
TAKEN - calculation performed, ready for new input data.
DONE - output data ready for taking.
ACCEPTED - output data taken, may now change;

'BASIC SUBBLOCK' EXREG -

A['INPUT' AIN[0:7] 'OUTPUT' AOUT[0:7]
  'CONTROL IN' LOADA,SHIFTA 'CONTROL OUT' LOADADN,SHIFTADN],

B['INPUT' BIN[0:7] 'OUTPUT' BOUT[0:7]
  'CONTROL IN' LOADB,SHIFTB 'CONTROL OUT' LOADBDN,SHIFTBDN],

C['INPUT' CIN[0:7] 'OUTPUT' COUT[0:7]
  'CONTROL IN' LOADC,SHIFTC 'CONTROL OUT' LOADCDN,SHIFTCDN],

D['INPUT' DIN[0:7] 'OUTPUT' DOUT[0:7]
  'CONTROL IN' LOADD,SHIFTD 'CONTROL OUT' LOADDDN,SHIFTDDN];
!EXREG - general purpose shift register with parallel load.
IN - 8 bit parallel input port. OUT - 8 bit parallel output port.
LOAD - start load cycle. LOADDN - load cycle complete.
SHIFT - start shift, output is shifted one place left, top bit is lost,
        bottom bit is replaced by bit on bottom end of IN.
SHIFTDN - shift cycle completed.
registers: A - MASK, B - DATA, C - RESULT, D - output buffer;
```

```
'BASIC SUBBLOCK' COUNTER -

COUNT7['OUTPUT' ZERO 'CONTROL IN' SET7,DEC 'CONTROL OUT' SET7DN,DECDN];

!COUNTER - down counter.
ZERO - equals "1" if counter contents are zero.
SET7 - set counter contents to 7. SET7DN - contents reduced by one.
DEC - reduce contents by one. DECDN - contents reduced by one;
'LOCAL CONTROL' SUBTRACT, AVAIL;
!SUBTRACT - set when a decrement of the counter is complete.
AVAIL - set when the output buffer is empty;

'CONNECTION' AIN <- MASKIN, BIN <- DATAIN, DIN <- COUT, DATAOUT <- DOUT;

'INITIALIZE' T1; 'INITIALIZE CONTROL' AVAIL;

'BEGIN';

'DECISIONS'; !decide if the current bit is to be saved;
D1: 'IF' AOUT[7]=0 'THEN'-> T5;
    -> T4;
'END DECISIONS';

T1: !wait for input, indicate ready;
    'SET' TAKEN;
    'WAIT FOR' GO;

T2: !initialize counter, RESULT, get input;
    'FLOW' CIN <- @00; 'SET' LOADA,LOADB,LOADC,SET7;
    'WAIT FOR' LOADADN&LOADBDN&LOADCDN&SET7DN;
    -> (T3,D1);

T3: !decrement counter;
    'SET' DEC;
    'WAIT FOR' DECDN;
    'SET' SUBTRACT;
    *; !terminate this control path;

T4: !transfer one bit from DATA to RESULT;
    'FLOW' CIN[0]<-BOUT[7]; 'SET' SHIFTC;
    'WAIT FOR' SHIFTCDN;

T5: !wait for end of cycle;
    'WAIT FOR' SUBTRACT;
    'IF' ZERO 'THEN' -> T7;
 // -> T3;

T6: !shift DATA and MASK up one place;
    'SET' SHIFTA, SHIFTB;
    'WAIT FOR' SHIFTADN&SHIFTBDN;
    ->D1;
```

```
T7: !wait for output buffer to become free;
    'WAIT FOR' AVAIL;

T8: !transfer RESULT to output buffer;
    'SET' LOADD;
    'WAIT FOR' LOADDDN;
    -> (T1,T9);

T9: !indicate output available and wait for reply;
    'SET' DONE;
    'WAIT FOR' ACCEPTED;
    'SET' AVAIL;
    *;

'END';
```

Figure A.1 shows the generated logic for the section of control surrounding task T5.

COMPUTER-AIDED DESIGN SESSION

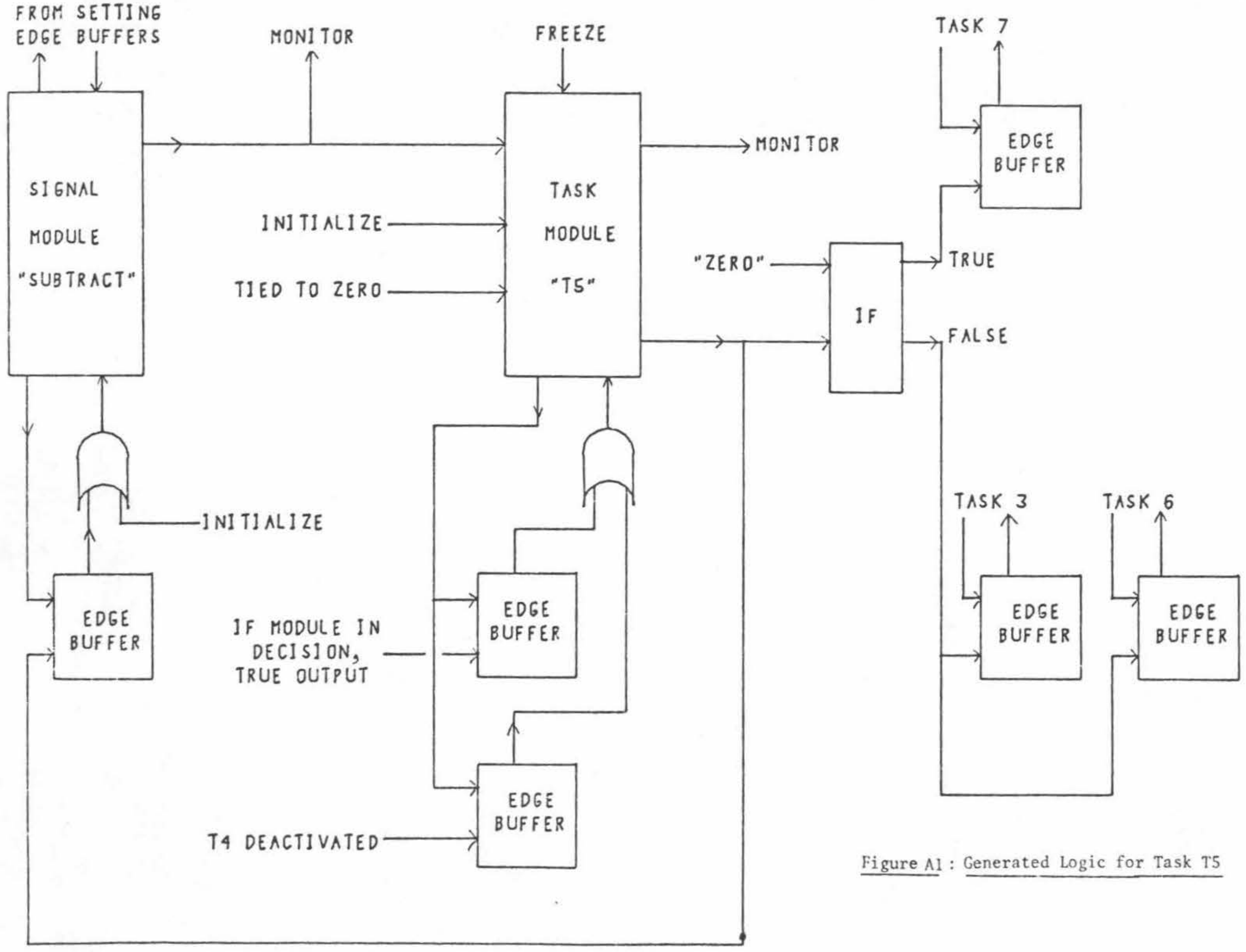

Figure A1 : Generated Logic for Task T5