HARDWARE DESCRIPTION LANGUAGE FOR SPECIFICATION OF DIGITAL SYSTEMS

Gy. Csopaki

Institute of Communication Electronics,
Technical University, H-1521 Budapest

Received January 13, 1987
Presented by Prof. Dr. S. Csibi

Abstract

The new hardware description language presented in this article was developed for the Computer Aid for Recursive Synthesis (CARS) system. System CARS has been developed to assist the designer in the synthesis phase of digital system design. CARS applies a top-down structured approach in the design process and can be used at any user defined level of refinement. The description language developed for CARS is well suited to describe behavioural concepts and structures as well.

Introduction

The hardware description language was developed for the CARS (Computer Aid for Recursive Synthesis) system. The system CARS uses a top-down structured approach in synthesis. The description language enables the designer to describe the behaviour and structure of a system at any level of design [1]—[3].

Requirements for the description language:

— the language provides the description of behaviour of a unit as a functional specification at any level and provides the definition of structure by components (lower level functional units) and their interconnections;

— the language elements and procedures provided should be totally level-independent;

— the description of a functional specification and its structure should be a document of this unit;

— the elements of the language provide time correct description of units.

1. Des'gn process in CARS

The design process in CARS consists of three steps to be repeated recursively [4]—[6]:

— the user describes the expected behaviour of units with time parameters. This description is called functional specification, and refers only to inputs and outputs. CARS checks the syntax;

— the user describes the propesed structure of this unit using functional specification of components and interconnections among them. CARS checks the syntax;

— the user executes the simulation process on the functional unit and the structure by the same input data. CARS compares the results of simulation on the functional specification and the structure;

— if the structure fulfils requirements, the procedure may be repeated for the components of the structure. When all functional specifications are descriptions of existing units, the design process is finished.

The features and elements of language

2.1. Description of behaviour

The behaviour of functional units can be described as a functional specification. In the language the functional specification is called TYPE. A type is identified by an identifier. The type identifier is succeded by the formal parameters; by the external input, output and bus signal identifiers.

Type- and signal identifiers have to be simple identifiers, i.e. strings beginning with letters and containing letters and numbers.

For example:

TYPE: ADDER (*A*, *B*, *CI*, *S*, *CO*)
ADDER is here the type identifier;
A, *B*, *CI*, *S*, *CO* are the formal parameters.

The user defines the external connections and the behaviour algorithm in the body of a type. The external connections are signals, grouped as inputs, outputs and busses. They can be described by an identifier. These identifiers have attributes, as width in bits, realization and representation of the signal values and limitations for set of values used. The realization of outputs and busses can be totem pole, open-collector and tristate. The direction for busses can be defined as bidirectional or output.

In general the declaration part:

INPUTS: ⟨signal declarations⟩.
OUTPUTS: ⟨signal declarations⟩.
BUSSES: ⟨signal declarations⟩.

The ⟨signal declarations⟩ part is a list of signal declarations separated by semicolons. All signals, having the same attributes can be declared together listing their names, separeted by commas and putting the common attributes after the names. Accordingly form of a general signal declaration:

⟨signal names⟩ ⟨width⟩ BITS, ⟨digit number⟩ ⟨representation⟩
DIGIT, ⟨output type⟩ ⟨bus direction⟩
EXCEPT (⟨exception list⟩)

The ⟨width⟩ and ⟨digit number⟩ parts are decimal numbers, the ⟨representation⟩ can be either BIN (binary), OCT (octal), DEC (decimal) or HEX (hexadecimal), while ⟨bus direction⟩ can be BIDIRECTIONAL or OUTPUT. The ⟨output type⟩ part can be OC (open collector) or TS (tristate), while the default option is totem pole. The ⟨output type⟩ and ⟨bus direction⟩ parts can be used only for output or bus signals. The ⟨exception list⟩ is a list of numbers or intervals, that specifies the prohibited values for the declared signals.

For example:

INPUTS: *A, B 4 BITS, 1 HEX DIGIT*; *CI.*
OUTPUTS: *S 4 BITS, 1 HEX DIGIT*; *CO.*

The signals are defined for a 4-bit binary adder, the length of the *A* and *B* input signals is four bits and *CI* (carry input) input signal is one bit long. When the length of signals is not specified, the default option is one bit. The representation is hexadecimal for the *A* and *B* input signals and for the *S* output signal, the default option is binary.

At the inputs of type the signal changes and at the outputs of type the responses for the input changes have to be defined. Both the input changes and the responses take the form of input and output events, respectively. Events are changes in the value of a signal. An input event consists of an identifier and the description of changes forming the event. The change description consists of a signal identifier and the value taken by the given signal.

Form of output event:

⟨event name⟩ : ⟨signal name⟩ CHANGES-TO ⟨value⟩.

The language allows the reference of any input signals by name or subsets of these signals, by using the ANY or ALL keyword together with the INPUTS or BUSSES signal type description. The change description can refer to one or more signals and to one value or to any value.

For example:

INPUT—EVENTS:
ARISE: A CHANGES-TO 1;
INPCHG: ANY-INPUTS CHANGE;
ALLCHG: ALL-INPUTS CHANGE-TO 1;
BCHG: B CHANGES;
CLKRISE: CLK CHANGES-TO 1;

Output events are responses for the input events changing the value of one or more output signals. An output event takes place if an input event triggers it, and certain time dependent conditions are met.

The time of the output event is expressed by a delay time from the triggering event.

The structure of an output event:

⟨event name⟩: AT ⟨time⟩, IF ⟨condition⟩, ⟨effect⟩

The ⟨time⟩ consists of an input event name reference and conditionally a delay time. The condition consists of logic expression that can include time restrictions. The time restrictions can refer to setup and hold time and to constant value of signals.

For example:

```
FLOPSET: AT CLKRISE+40 NSEC,
  IF D=1 FROM CLKRISE−10 NSEC TO CLKRISE+5 NSEC, Q=1
```

This example describes the setting of an edge triggered *D* flip-flop. The *Q* output signal takes the value *1* after 40 nsec of the CLKRISE input event. The value assignment takes place only after the *D* input signal satisfies the 10 nsec setup and the 5 nsec hold time condition. The condition part may be more complex and the effect part may consist of many assignments. At the left side of the assignments may be signal reference indexed or not indexed, and at the right side of the assignment may be a logic assignment referring to any declared signals or any constant. The effect part may consist of memory read or memory write operations.

If there are many output events depending on the same condition, this may be declared separately as a common condition. Common conditions may be nested without any restrictions.

A type may consist of any input, output and bus signal, any input event and any output event. There are no restrictions for the complexity of condition or effect part. An example of a complete type gives description of behaviour of a *D* flip-flop.

```
TYPE: DFLOP (D, CP, CL, PR, Q, NQ).
INPUTS: D, CP, CL, PR.
OUTPUTS: Q, NQ.
INPUT-EVENTS: CPRISE: CP CHANGES-TO 1 BIN;
              CLFALL: CL CHANGES-TO Ø BIN;
              PRFALL: PR CHANGES-TO Ø BIN;
OUTPUT EVENTS:
  SET: AT CPRISE+33 NSEC, IF D STEADY
        FROM CPRISE−20 NSEC TO CPRISE+5 NSEC
        AND CL=1 BIN AND PR=1 BIN,
        Q=D, NQ=NOT D;
  CLEAR: AT CLFALL+33 NSEC, Q=Ø BIN, NQ=1 BIN;
  PRESET: AT PRFALL+33 NSEC, Q=1 BIN, NQ=Ø BIN.
DFLOP END.
```

In this example also can be seen that the type description gives nothing about the internal structure of the described object.

2.2. Language elements for simplification

If the designer does not know the exact time when the output signals get their new values, he or she can use an interval time in which the value of actual output signals are not available. The name of this element is operation.

The structure of an operation:

```
⟨operation identifier⟩: ⟨time and condition⟩;
                        ⟨effect part⟩
                        ⟨operation identifier⟩ END.
```

The prefix ⟨operation identifier⟩: and the suffix ⟨operation identifier⟩ END flank the description body.

In the ⟨time and condition⟩ part can be specified the start and stop time for the operation.

Form of start specification:

STARTS-AT ⟨time expression⟩, ⟨condition⟩
Form of stop specification:
TERMINATES-AT ⟨time expression⟩ OR ⟨condition⟩

If the condition part is specified, the operation will be executed only if the condition is met. If stop is specified with a condition, the execution of the operation spec ified by the effect part will be ended if this condition is met.

Form of effect part:

⟨normal effect part⟩, ⟨early stop part⟩

In the normal effect part there are assignments specifying the value of the signals at stop time. The values of these signals are unavailable during the operating time. When the operation is completed because of the condition in the stop specification, the value of the signals referred in the effect part takes those specified in the abort part of the assignments.

In the effect part the next statements may be used: simple assignment, IF assignment, ON assignment, memory operations. On the right side of the simple assignment expressions including arithmetical and logical operators may be specified. The IF assignment consists of one or two simple assignments and a condition. If the condition is met the first simple assignment (after the THEN keyword) will be executed otherwise the second one (after the ELSE keyword).

In the ON assignment more assignments may be given. Depending on the value of the referred variable the appropriate assignment will be executed.

In the top-down design process not only TYPES are decomposed into STRUCTURES but OPERATIONS are decomposed into lower level OPERATIONS and/or EVENTS as well. This feature of the language helps the designer to make time refinements.

Nevertheless the bottom-up approach is also supported by the operation concept since different event and/or operations can be composed into a higher level operation, thus eliminating unimportant details of timing considerations at a particular level.

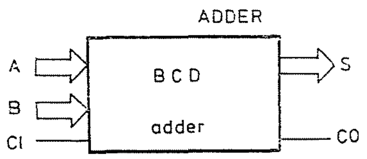

Fig. 1. BCD adder

Example for operations (Fig. 1.):

```
TYPE: ADDER (A, B, CI, S, CO).
INPUTS: A, B 4 BITS, 1 HEX DIGIT; CI.
OUTPUTS: S 4 BITS, 1 HEX DIGIT; CO.
INPUT-EVENTS:
CHG: ANY-INPUTS CHANGE.
OPERATIONS:
SUM: STARTS-AT CHG
TERMINATES-AT CHG+24 NSEC;
RESULT: S=A+B+CI.
SUM END.
CRY: STARTS-AT CHG
TERMINATES-AT CHG+16 NSEC;
RESULT: IF (A+B+CI)〉=10000BIN THEN CO=1 BIN;
IF (A+B+CI)〈10000 BIN THEN CO=0 BIN.
CRY END.
OPERATIONS-END.
ADDER END.
```

In this example the input signals are two four bit operands and a carry-in digit for a full adder. The BIN keyword is compulsory in the effect part and in the conditional part, while the default option is decimal.

2.3. *Description of structure*

The structure description has name, input and output connections and elements.

The external connections must be declared as INPUTS, OUTPUTS or BUSSES. After the signal declarations, structure body has two lists: the list of elements constituting the structure and the list of connections among the elements.

The elements of structures are realized by types, and as more elements may be realized by the same type, the actual realization is regarded as copies of that particular type. Copies are identified by proper names indicating the actual place of use and parameters.

The form of element list:

ELEMENTS: ⟨type identifier⟩: ⟨copy list⟩.

Types listed in this element list are component types of the given structure and their level is considered lower than level of actual structure.

The parameters for a given copy in the element list correspond to those in type definition. Signal names as formal parameters in the element list may coincide with names of input or output signals or with parameters of other elements in the list. In both cases the system automatically connects all terminals concerned. This is an implicit definition of connections.

The connection list defines all remaining nets in the structure. A net definition is a list of connected signals separated by concatenation mark (—). The designer may refer to any subset of a multi-bit wide signal by using subscripted identifiers.

The form of connection list:

CONNECTIONS: ⟨list of nets⟩.

where ⟨list of nets⟩ is the definition of all nets separated by semicolons.

In this explicit case of definition inputs and outputs of the components have to be identified by the qualified identifier:

⟨element name⟩ . ⟨name of signal in type declaration⟩

Qualified names may be subscripted, too.

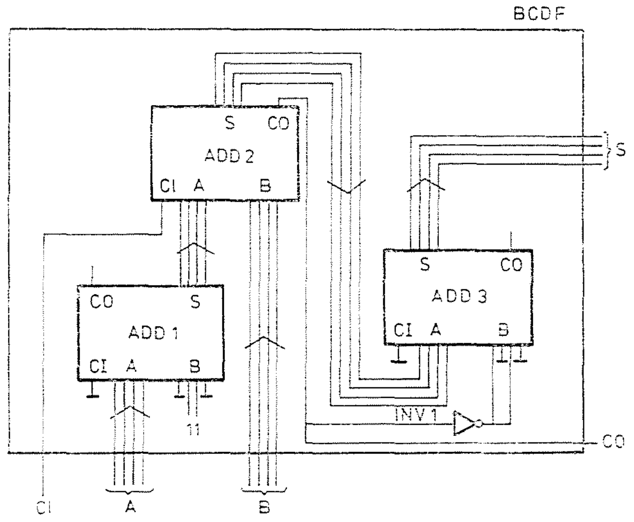

Fig. 2. The structure of a BCD adder

An example for a structure, describing structure of a *BCD* adder (Fig. 2): -

```
STRUCTURE BCDF.
INPUTS: A, B 4 BITS 1 DEC DIGIT; CI.
OUTPUTS: S 4 BITS 1 DEC DIGIT; CO.
ELEMENTS: ADDER:          ADD1 (A, 6 DEC, Ø BIN, *, *),
                          ADD2 (*, B, CI, *, *),
                          ADD3 (*, *, Ø BIN, *, S);
          INV:            INV1 (*, *).
          CONNECTIONS:    ADD2. S-ADD3.A,
                          ADD2. CO—INV1. A—CO,
                          INV1. Y—ADD3. B(1)—ADD3.B(3),
                          ADD3. B(Ø)—ADD3.B(2)—Ø BIN.
          BCDF END.
```

In this example *ADD1*, *ADD2* and *ADD3* are names of copies of *ADDER*, which is four bit binary full adder.

3. Test procedure

Let us consider two descriptions of the same digital system. One is the behavioural description, a type, the other is a structural one, a structure. Let us assume that all the elements in the structure are defined, thus both type and structure can be tested by simulation using the same input data.

The results of the two tests must be identical if the model is correct. This comparison is the last step of the process. A test set is specified by type description since it must contain all functions expected from the given type. On the other hand the structure may contain additional, unspecified functions, too. This way a positive result of the comparison does not mean identity of type and structure indicating only that the structure fulfils the specified requirements.

If the same structure is tested for a different type description, as necessary in bottom-up approaches, it may or may not fulfil additional or changed requirements.

Comparison means an evaluation, whether the set of events describing the behaviour of the type is a complete subset of possible events of the respective structure.

This evaluation may be performed at different levels. Obviously the evaluation on the lowest level is easy, because only numeric result-values of simulation must be compared, but simulation must be done for all possible values and combinations of inputs within the region of specified input value set.

Summary

The hardware design language described in this article was developed for the CARS system. System CARS has been developed to assist the designer in the synthesis phase of the digital system design. CARS applies a top-down structured approach in the design process and can be used at any user defined level of refinement. CARS does not pose any restriction on the creative work of the designer, but it only gives means to describe his concepts and tools to check his results. The design language developed for CARS is well suited to describe behavioural concepts and structures as well. One of the most important features is that the designer can use the same language elements for the behavioural and for the structural description.

References

1. Csopaki, Gy.: Hardware description language for design of digital equipment. Proc. of the Simulation in Research and Development Elsevier Science Publishers B. V. (North Holland) 1985.
2. Csopaki Gy.: Hardwarebeschreibungssprache für Darstellung digitaler, logischer Systeme. Tagungsberichte Heft. 10. 1986. 4. Symposium Grundlagen und Anwendung der Informatik. Wissenschaftliche Tagung. Febr. 1986.
3. Csopaki, Gy.: Digitális integrált áramkörök specifikációja. A Mikroelektronikai berendezés-orientált áramkörök tervezése könyv 6. része. Szerkesztő: Dr. Tarnay Kálmán EDUSYS. 1984.
4. Bohus, M.—Csopaki, Gy.—Filp A.—Hinsenkamp, A.—Máté, L.: Computer aid for recoursive synthesis. Working paper. Computer and Automation Institute, Hungarian Academy of Sciences. 1982.
5. Complander, H. D.—Janku, J. A.: Top down approach to LSI system design. Computer Design. Vol. 13. No. 8. 1974.
6. Hill, D.—van Cleemput, W.: SABLE a tool for generating structured multi-level simulation. Proc. of. 16th Design Automation Conference, San Diego, 1979.

Dr. Gyula Csopaki H-1521 Budapest Pf. 91.