

# High-Level Design of Digital Logic Hardware\*

Sunggu Lee<sup>†</sup>

<sup>†</sup> Dept. of Electrical Engineering  
Pohang University of Science and Technology (POSTECH)  
San 31 Hyoja Dong, Pohang 790-784, SOUTH KOREA  
slee@postech.ac.kr

## Abstract

With the aid of electronic design automation (EDA) computer-aided design (CAD) tools, it is now possible for engineers to design highly complex digital logic hardware circuits using only high-level circuit descriptions. Such EDA tools include software tools for graphical design input, hardware description language support, simulation, digital logic synthesis, and the creation of configuration (or netlist) files for customizable hardware platforms. A high-level design approach can be used with the aid of such tools as long as the desired behavior of the target circuit can be described in an unambiguous manner.

Given a general design problem, it may be solvable using software or hardware methods. A customized hardware solution, however, has an advantage over a software solution in terms of performance and cost, and in some cases may be the only viable approach. Thus, this paper describes a general approach for taking a software algorithm, converting it into a high-level design description, and then using that design description to automatically produce customized hardware circuits with the aid of EDA CAD tools.

**Keywords**—Algorithm, Algorithmic State Machine (ASM), Electronic Design Automation (EDA), Hardware Description Language (HDL), High-Level Design.

---

\*This work was supported in part by the IC Design Education Center (IDEC).

# 1 Introduction

Given an ideal design automation environment, circuit design should be simple. The designer should be able to describe his desired circuit using a simple English description, and then the design automation tool should be able to produce the desired circuit based on that description. The current state-of-the-art electronic design automation (EDA) tools have not yet reached this level of design automation. However, current EDA tools do support various types of high-level design descriptions, as long as those descriptions are able to describe the target circuit in an *unambiguous* manner using keywords and constructs supported by the specific EDA tool used.

This paper presents a general approach for the high-level design of digital logic hardware circuits. Given a digital logic hardware design problem, a detailed approach to solving the problem must first be devised. This solution can be represented using pseudocode or a flowchart, which are widely used methods for specifying software algorithms. This type of solution is then converted into an algorithmic state machine (ASM) chart based on heuristics presented in this paper. Next, an EDA tool such as Exsedia's Nimbus can be used to simulate and verify this ASM chart. The Nimbus tool also converts an ASM chart into hardware description language (either VHDL or Verilog) code. Then, other EDA tools can be used to synthesize and implement the desired logic circuit.

## 2 Algorithmic State Machines (ASMs)

Several types of high-level design entry methods are supported by current EDA tools. These methods can be categorized into textual entry and graphical entry methods. Textual entry methods are typically based on the use of hardware description languages such as VHDL and Verilog. Since VHDL and Verilog are standard languages supported by most commercial EDA tool vendors, designs entered using these languages have the advantage of being portable. Graphical entry methods typically include schematic design entry, state machine diagrams, and algorithmic state machine (ASM) diagrams. Schematic design entry is used for very specific low-level design entry and hierarchical design using previously defined subcircuits and library components. State machines and ASMs can be used for a higher-level of design entry based on a synchronous sequential circuit model.

Of the above design entry methods, ASM design entry has the advantage of being a high-level design entry method that can describe a hardware design at an “algorithmic” level. However, an ASM is used to describe digital logic hardware, and thus has characteristics that differentiate it from an algorithm meant to be implemented in software. Components in a digital logic hardware circuit execute concurrently and continuously, and carefully designed interactions between those components produce the *effect* of a coordinated sequence of operations based on an algorithm. Thus, an ASM description should be able to accurately describe the behavior of this type of circuit.

An ASM description, referred to as an *ASM chart*, consists of several graphical blocks interconnected in a manner similar to a flowchart [Clare 1973]. An ASM chart consists of *state boxes*, *condition boxes*, and *conditional*

*output boxes.* A state box, drawn as a rectangle, contains operations to be performed within one “state” of the circuit. Note that the operations within a state box are completed at the *end* of the corresponding state (any updated variable values cannot be observed until the next state). A condition box, drawn as a diamond, contains a boolean condition to be checked. The “true” (1) and “false” (0) paths out of the condition box are used to transition to different states based on the result of the boolean condition check. A conditional output box, drawn as an oval, can also be connected to the output of a condition box, and is used to specify operations to be conditionally executed based on the result of a boolean condition check. Figure 1 shows an example of an ASM chart.

The ASM chart is most suited for the design of synchronous sequential digital logic circuits, which are digital circuits in which the outputs change at times specified by a clock signal based on the values of current and past inputs. A synchronous sequential circuit can be partitioned into a *datapath* component, which consists of registers and combinational logic used to manipulate the values stored in registers (such as adders and multiplexers), and a *control logic* component, which controls the overall operation of the synchronous sequential circuit and produces control signals to control the operation of the modules in the datapath component. The operation of such circuits can be described as a sequence of *register transfer* operations, which are operations involving the transfer of data from one or more registers to another (or the same) register. The data can be transformed (e.g., shifted, added, subtracted, etc.) while it is being transferred, and memory locations can be used instead of registers.

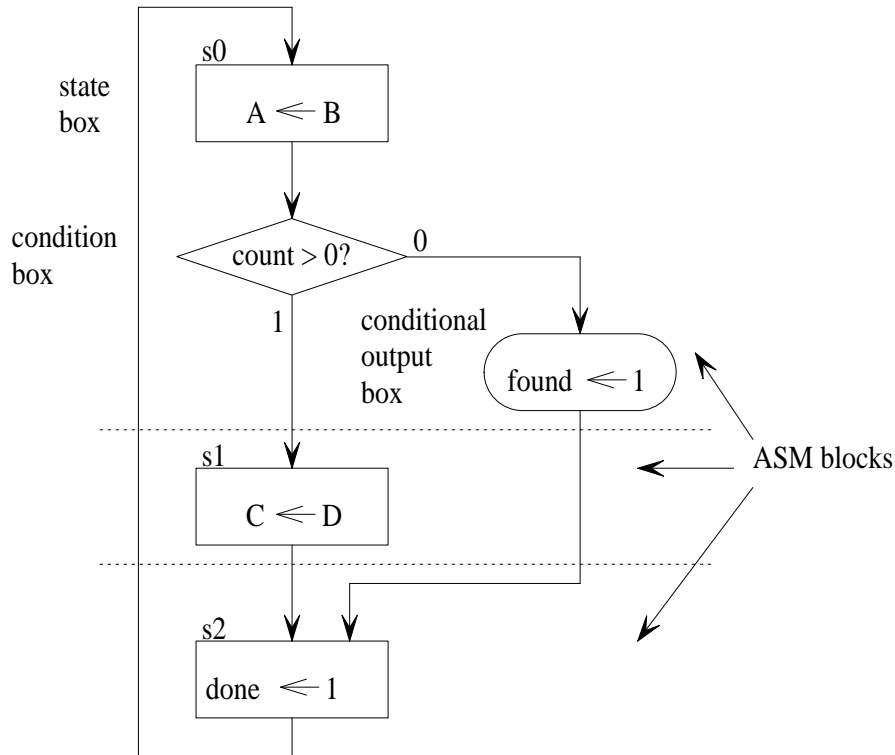


Figure 1: An ASM chart with the major elements labeled.

A register transfer operation can be described in the following manner:

$$R_i \leftarrow f(R_1, R_2, \dots, R_k) ,$$

where  $R_j$  values correspond to register values and  $f(\cdot)$  corresponds to a data transformation function such as addition, left-shift, right-shift, AND, OR, etc. As shown in the example of Figure 1, state boxes and conditional output boxes in ASM charts should contain only register transfer operations. Condition boxes should contain simple combinational logic queries. Then, given such a construction, each state box and the condition boxes and conditional output boxes that follow it (up to the next state box) can be considered as an

*ASM block.* Each ASM block is active during one clock cycle of the system clock. A state transition occurs from one ASM block to another following each transition of the system clock.

### 3 Converting Software into ASMs

Given a design problem to be solved, a first step toward designing a solution to the problem can involve the creation of a high-level software algorithm description written in pseudocode or flowchart notation. Such software algorithm solutions are relatively simple to write since a free-form syntax can be used and we do not have to be concerned with timing (specific timing behavior can, of course, be added if desired). However, such free-form solutions cannot be converted to hardware designs by the current generation of EDA tools.

A software algorithm is typically not concerned with the timing relationships between successive steps of the algorithm. Thus, a software algorithm, typically represented using *pseudocode* or a flowchart, consists of a sequence of steps, with each step assumed to execute *after* the previous step in the algorithm. It may be the case that two or more operations are independent of each other, and thus can be executed concurrently. However, software algorithms are typically not concerned with this situation, since it is assumed that a compiler will make all of the transformations necessary to convert a software program to machine code. The programmer can simply assume that each step of his/her software algorithm will execute one after the other, and the compiler will take care of the actual conversion to machine executable

code.

A hardware algorithm, on the other hand, *must* be concerned with the timing relationships between successive register transfer operations since all hardware devices have delays associated with them. Depending on the register transfer operation being performed, it may require a short delay (e.g., with a transfer of untransformed data) or an extremely long delay (e.g., with a division operation). An operation requiring a long delay can be pipelined or split into a series of simpler operations executed in separate states. Also, two or more register transfer operations that can be performed in parallel should be allowed to execute in parallel in order to use the available hardware resources efficiently. If higher performance is required, it may be possible to use extra hardware resources to execute more register transfer operations in parallel.

In order to be able to create digital logic hardware starting from a software algorithm description, it must first be converted into a hardware algorithm description. An ASM chart is a high-level hardware algorithm description that is syntactically and semantically similar to a software algorithm description, yet is amenable to direct hardware implementation. Thus, a general high-level hardware design method involves the creation of a software algorithm, followed by conversion of the algorithm into an ASM chart, followed by the use of EDA tools to convert the ASM chart into working hardware.

### **3.1 Software-to-ASM Conversion Heuristics**

A series of heuristics can be used to convert a software algorithm into an ASM chart. Let us assume that the software algorithm is written in pseudocode

notation (an analogous set of heuristics can be used with flowcharts as well). Also, for simplicity, let us assume that the pseudocode consists of variable assignments (perhaps involving arithmetic calculations), conditional actions, **for** loops, **while**, and **repeat-until** loops. All other constructs, such as subroutine calls, will be converted to the above basic constructs before adopting the heuristics listed in this section.

First, variable assignments can be converted to ASM states with register-transfer operations. Each variable used in the pseudocode will be treated as a register, an input signal, or an output signal. All variable assignments which do not have to occur in sequence (i.e., which are independent of each other) can be placed into the same state box.

Second, conditional actions in pseudocode can be converted to condition boxes (or condition boxes and conditional output boxes) in ASM charts. If the condition check involves checks of several variables, then it may be converted into a series of condition boxes. Also, if the conditional action involves a conditional variable assignment, then the corresponding condition box may be followed by a conditional output box. However, since a conditional output box will execute during the state corresponding to the preceding state box (i.e., within the same ASM block), the variable assignments during this preceding state box must be independent of the variable assignments in the conditional output box (otherwise, an empty state box can be used).

Third, a **for** loop can be converted into a sequence of state boxes and condition boxes. Suppose that an index variable *count* is used, as in “for (count = 0; count  $\leq$  MAX; count = count + 1) ...”. Then, the sequence of state boxes and condition boxes shown in Figure 2(a) can be used. Note that

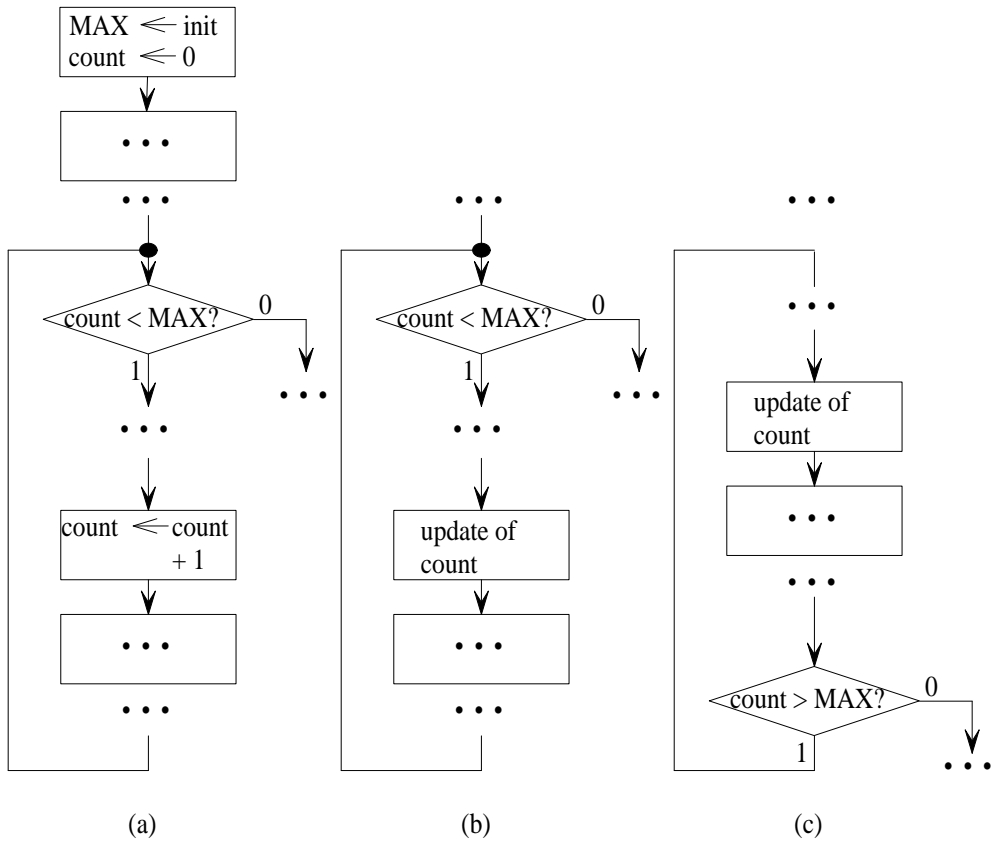


Figure 2: ASM charts corresponding to (a) **for** (b) **while**, and (c) **repeat-until** loops.

the initialization of *count* to 0, as well as the increment of *count*, must take place at least two states before the check  $count < MAX$ . This is because the update of *count* will only take place at the *end* of the state in which it is changed. Thus, the updated value of *count* cannot be checked immediately after the state in which it is changed.

Fourth, **while** loops and **repeat-until** loops can be converted into sequences of state boxes and condition boxes as shown in Figures 2(b) and 2(c). For a **while** loop (e.g., “while ( $count \leq MAX$ ) do ...”), the condition box is placed at the beginning of the sequence of states forming the loop. For a **repeat-until** loop (e.g., “repeat ... until ( $count \leq MAX$ )”), the condition box is placed at the end of the sequence of states in that loop. For both types of loops, note again that any register-transfer operations that update the variable being checked (e.g., *count*) must take place at least two states before the corresponding condition box.

## 3.2 Examples

Let us illustrate the above software-to-ASM conversion method with the aid of two practical example designs. The first circuit will be an asynchronous communications interface and the second circuit will be a two-elevator control circuit.

### 3.2.1 Asynchronous Communications Interface

In order for two hardware devices to communicate with each other, they must agree on a protocol for sending and receiving data. At the most basic level, this protocol must include methods for determining exactly when data

sent by one device is sampled by the other device and for the sending device (transmitter) to determine if its data has been received properly or not. In *synchronous* communication, this is accomplished by sending a periodic *clock* signal with the data signal. Then the receiving device (receiver) can sample the data signal during the *transitions* of the clock signal, and the transmitter can simply assume that its data will be received properly within one clock period. In *asynchronous* communication, there is no clock signal. Instead, a *handshaking* protocol is used to determine when the data signal is sampled and when data has been received properly. Asynchronous communication is more flexible than synchronous communication since the former does not require the transmitter to operate at the same speed as the receiver while the latter method requires the transmitter and receiver to agree on a common transmission speed (the clock rate).

A commonly used *handshaking* protocol is the *four-phase handshake*. In this method, in addition to the data signal, there is a *request* (*REQ*) signal sent from the transmitter to the receiver, and an *acknowledge* (*ACK*) signal sent from the receiver to the transmitter. When the data signal is ready to be sent, the data is sent first followed by a *REQ* signal ( $REQ = 1$ ). Upon observing the change in the *REQ* signal, the receiver can sample the data signal and then send an *ACK* signal ( $ACK = 1$ ) back to the transmitter. When the transmitter observes  $ACK = 1$ , it can deassert its *REQ* signal ( $REQ = 0$ ) in preparation for the transmission of the next data signal. This four-phase process is illustrated in Figure 3.

Let us consider the design of a circuit for asynchronous communication based on the four-phase handshake. The desired operation of this circuit can

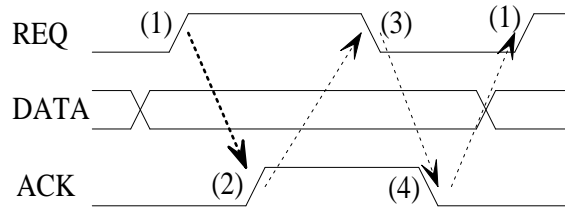


Figure 3: The four-phase handshaking protocol for asynchronous communication.

be inferred from the description in the above paragraph. More formally, we can derive a pseudocode description of the operation of this circuit, as shown below. This pseudocode description could be the basis for a software program to emulate the desired behavior. Alternatively, as will be shown here, the pseudocode can form the basis for the target hardware circuit itself.

*Pseudocode Solution for the Four-Phase Handshake Circuit:*

0.  $REQ \leftarrow 0$ ;
1. Wait until (next data item is ready);
2. Send data signal;
- 3  $REQ \leftarrow 1$ ;
4. Wait until ( $ACK = 1$ );
- 5  $REQ \leftarrow 0$ ;
6. Wait until ( $ACK = 0$ );
7. Go to Step 1;

In order to convert the above pseudocode solution into an ASM chart, let us follow the conversion heuristics presented in Section 3.1. Step 0 is a variable assignment, and thus can be converted into a state box with the

corresponding register transfer operation (the *REQ* signal will be stored in a 1-bit register named REQ). Steps 1, 4, and 6 are equivalent to **repeat-until** loops of the form “repeat (do nothing) until (condition).” The “do nothing” action can be implemented as an empty state box, which can sometimes, but not always, be combined with the previous state box (Step 1 requires an empty state box since Step 7 returns to Step 1, while Steps 4 and 6 do not require extra empty state boxes). Steps 2, 3, and 5 are again variable assignments that can be converted into register transfer operations. Finally, Step 7 simply loops back to Step 1, thereby producing an infinite loop (hardware circuits are often modeled using infinite loops since hardware executes infinitely, or until the power is turned off). In Step 1, the condition check “until (next data item is ready)” implies the use of a special combinational logic for this check or a special signal used to indicate this condition, which in this case will be a *READY* signal. The resulting ASM chart for this four-phase handshake circuit is shown in Figure 4.

### 3.2.2 Two-Elevator Control System

As a second design example, let us consider the problem of controlling a two-elevator system using one set of up and down switches. Some assumptions are necessary in order to be able to specify this problem and its solution in sufficient detail for hardware implementation. This control circuit is used to control the circuitry on one floor of a multi-story building with two elevators. The signals *REQ\_UP* and *REQ\_DOWN* are asserted when the user presses the “up” and “down” elevator buttons, respectively. Then, the elevator that is closest to the current floor is called. If that elevator is already moving

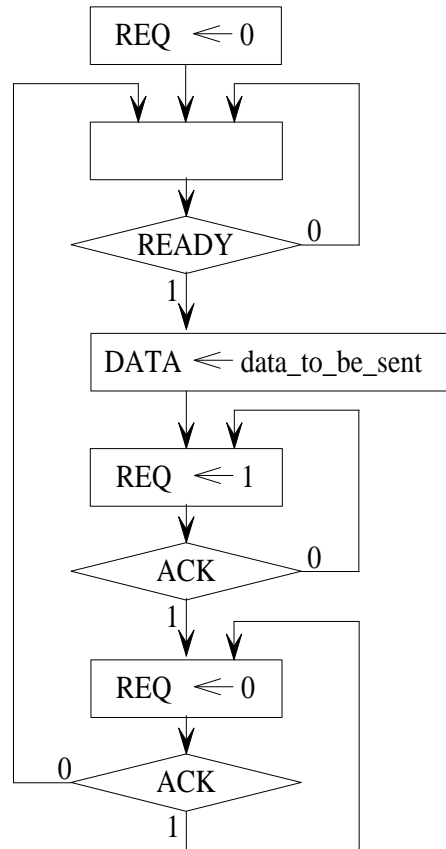


Figure 4: The ASM chart for the four-phase handshake circuit.

toward the current floor in response to a previous call (from a different floor) and its direction of movement is the same as the direction of the user request, then that elevator will stop on the current floor when it reaches it and assert *STOP1* or *STOP2* depending on whether it is elevator 1 or 2, respectively. If the elevator being called moves away from the current floor, then (1) it will eventually move farther away from the current floor than the other elevator, which can then be called instead, or (2) it will stop for a while on some other floor and then move toward the current floor. To cover all of these various cases, the positions of the two elevators will be continuously sampled, and either *CALL1* or *CALL2* asserted depending on which elevator is closer to the current floor at that time. The resulting pseudocode and ASM chart are shown below.

*Pseudocode Solution for the Two-Elevator Control Circuit:*

0.  $CALL1 \leftarrow 0;$   
 $CALL2 \leftarrow 0;$
1. Wait until ( $REQ\_UP$  or  $REQ\_DOWN$ );
2. Repeat
  - 2a.  $DIFF1 \leftarrow POS1 - CURRENT;$   
 $DIFF2 \leftarrow POS2 - CURRENT;$
  - 2b. if ( $|DIFF1| < |DIFF2|$ ) then
    - $CALL1 \leftarrow 1;$
    - $CALL2 \leftarrow 0;$
 else
    - $CALL1 \leftarrow 0;$
    - $CALL2 \leftarrow 1;$

3. until ( $STOP1$  or  $STOP2$ );
4. Go to Step 0;

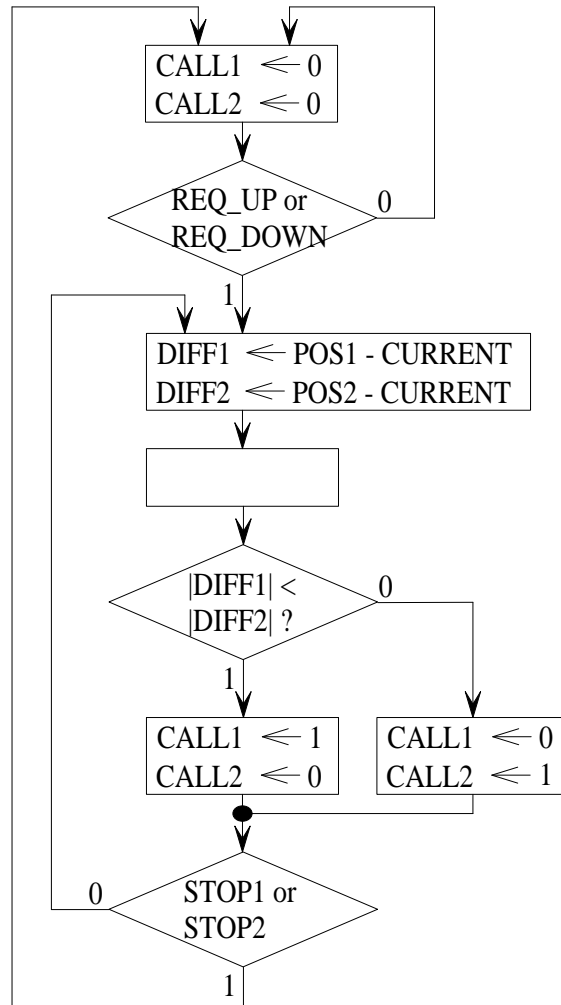


Figure 5: The ASM chart for the two-elevator control circuit.

As before, the pseudocode solution consists of variable assignments and **repeat-until** loops (the “wait until” statement can be treated as a special

type of **repeat-until** loop). These statements can be converted to ASM chart structures using the conversion methods presented above. Note that in this simple solution, shown in ASM chart form in Figure 5, the user may have to wait a bit longer than necessary if the elevator closer to him/her is moving away from him/her.

## 4 ASM Design Entry and Simulation

As an example of the use of ASM design entry and simulation, let us use Exsedia's Nimbus EDA tool [Exsedia 2004]. This tool uses a slightly modified form of an ASM chart referred to as a flowdiagram [Davis and Sheldon 1997]. However, since a flowdiagram is a superset of an ASM chart, this tool also supports design and simulation using ASM charts. The example ASM chart used will be the four-phase handshake circuit presented in the previous section. Figure 6 shows the corresponding flowdiagram entered using Nimbus. Then, the simulation for this flowdiagram results in the simulation waveforms shown in Figure 7.

## 5 Discussion

This paper has presented a general approach for the high-level design of digital logic hardware circuits based on converting a pseudocode solution to an ASM chart, and then using EDA tools to simulate the ASM chart and convert the ASM chart into HDL code that can be synthesized into the final hardware circuit. An ASM chart has a well-defined structure that facilitates

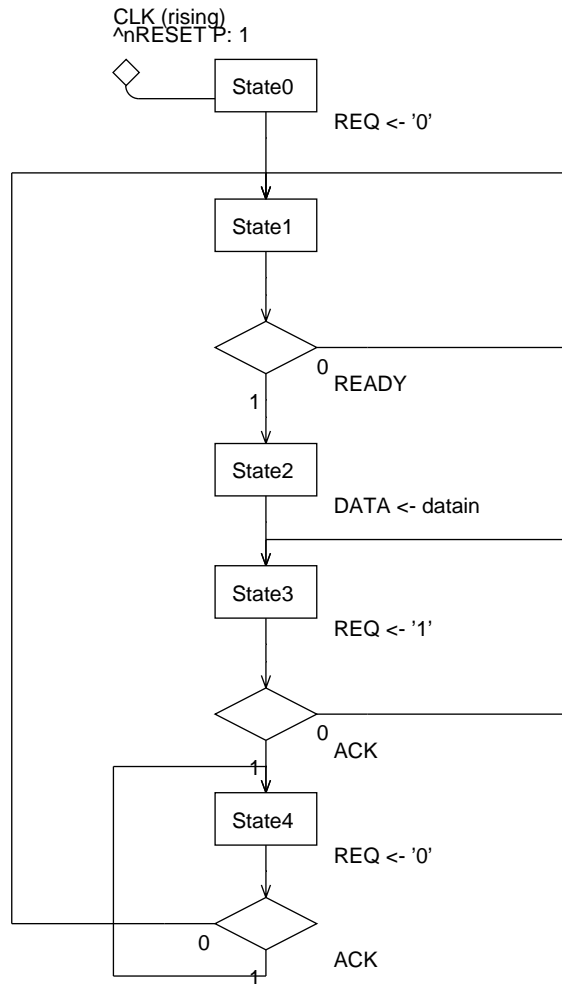


Figure 6: The flowchart design for the four-phase handshake circuit, entered using Exsedia's Nimbus tool.

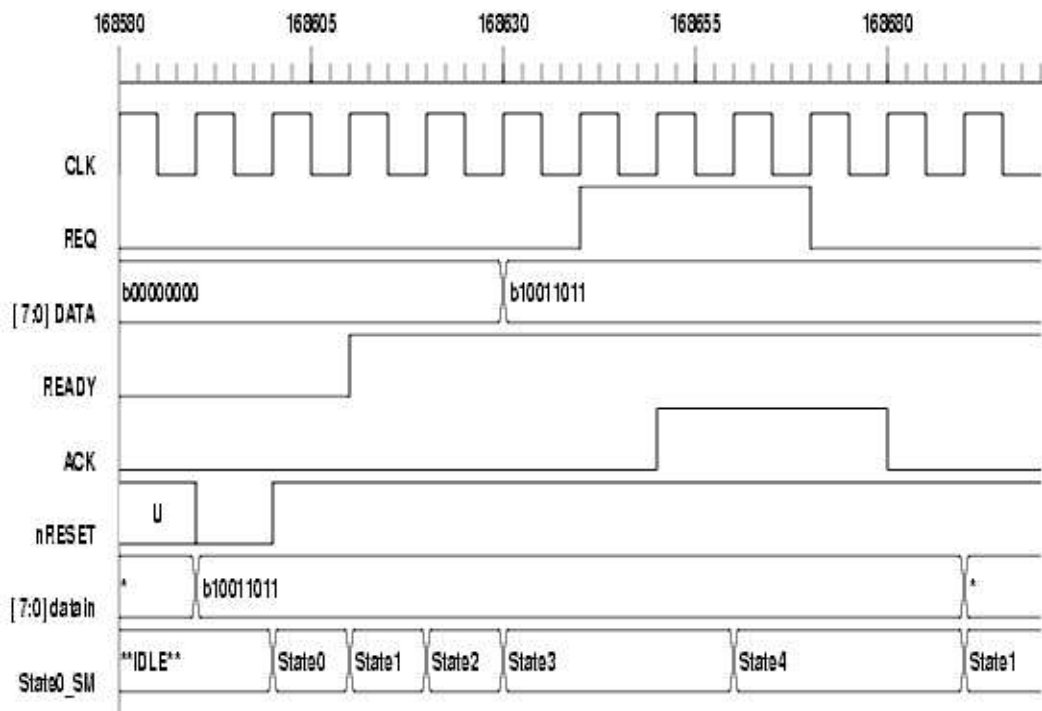


Figure 7: The Exsedia Nimbus simulation waveforms for the four-phase handshake circuit.

implementation in digital logic hardware, while a pseudocode description (or flowchart) uses a free-style format to describe general *algorithms*, which are typically meant to be executed in software. However, as shown in this paper, such a pseudocode description can be converted into an ASM chart using a set of heuristics that formalize structures used in the pseudocode and adhere to timing constraints imposed by synchronous sequential digital logic hardware. Examples have been used to demonstrate the use of this technique and its implementation using EDA tools such as Exsedia's Nimbus tool.

## **LIST OF REFERENCES**

CLARE, C. R., *Designing Logic Systems Using State Machines*, McGraw-Hill, New York, 1973.

<http://www.exsedia.com>, home page for Exsedia Inc.

DAVIS, J. AND C. SHELDON, "A graphical approach to high-level, HDL-based VLSI systems design," Knowledge Based Silicon Corp., 1997.