A FINITE STATE MACHINE DECISION ALGORITHM

John Wiese
Alpha Design Corp.
San Jose, CA

John Wiese is president of Alpha Design Corp., which specializes in contract software and hardware solutions for control, imaging, communications, monitoring, and other real-time applications. He is working on next-generation, computer-aided real-time design tools and algorithms that maintain performance and programmability for performance-oriented systems. He received an electrical/computer science degree from Ohio State University.
A Finite State Machine Decision Algorithm

John Wiese
Consultant
Alpha Design Corporation
1061 Oaktree Drive
San Jose, California 95129
(408) 253-7662

INTRODUCTION

Finite state machines have long been a tool used by hardware designers while software designers have often overlooked this valuable method. One of the main reasons a software designer may not use finite state machines is that there has not been an easy way to implement them that is both fast and flexible. This paper will introduce a new, decision-testing algorithm for software-based state machines which was co-developed by Kader Laroussi and John Wiese. Its main features include: fast state transitions where speed is not a linear function of the number of inputs but a function of the processor’s word size; an easy-to-change state table; a small table size by using don’t care terms; and centralized decision-making, which makes it easier to debug and maintain.

This paper will focus on the decision algorithm and its use as opposed to finite state machine design methods. Any design which can produce a state transition table can use this algorithm. A subsequent paper will present an algorithm with fast decisions for boolean equation based state machines.

DECISION ALGORITHM

The decision algorithm has two inputs: the state machine inputs and the transfer condition which cause the state machine to transfer to the next state. For the sake of demonstration, we will first use a simple one-input system and expand on this. We will represent the input as (I) and the transfer condition as a combination of the transfer condition mask (B) and the don’t care mask (C). When the following logic equation is zero, the condition to transfer is TRUE.

\[(I \text{ xor } B) \text{ and } C = 0 \quad \text{transfer}\]
\[(I \text{ xor } B) \text{ and } C = 1 \quad \text{no transfer}\]
where:

- **I** is the input
- **B** is the transfer condition mask with a 0 for don't care conditions
- **C** is the don't care mask to indicate don't care conditions with a 1 for valid condition and 0 for don't care condition

The extension of this algorithm to a N bit-wide system is a matter of simply packing each test as one of the bit positions within a N bit-wide word. We will use subscripts to note the bit positions. Therefore, the equations become:

- \((I_i \text{ xor } B_i) \text{ and } C_i = 0\) transfer
- \((I_i \text{ xor } B_i) \text{ and } C_i => 1\) no transfer

Example of a three bit-wide system:

Let the transfer condition be \(10X\) where \(X\) is a don't care condition

Then \(B\) will be 100 and \(C\) will be 110

Table I will represent all possible inputs for \(I\).

<table>
<thead>
<tr>
<th>(I)</th>
<th>(I \text{ xor } B) and (C)</th>
<th>TRANSFER</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>100</td>
<td>FALSE</td>
</tr>
<tr>
<td>001</td>
<td>100</td>
<td>FALSE</td>
</tr>
<tr>
<td>010</td>
<td>110</td>
<td>FALSE</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>FALSE</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>TRUE</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>TRUE</td>
</tr>
<tr>
<td>110</td>
<td>010</td>
<td>FALSE</td>
</tr>
<tr>
<td>111</td>
<td>010</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Table I

As expected \(I = 100\) and \(I = 101\) will cause a transfer. Also note that when the result is not equal to zero, the bit position(s) that are equal to one is the test that failed. This nice effect is great for debugging your state machine designs and status logs.

Besides using this decision algorithm for finite state machines there are many other applications. It can be used anywhere you have a bit pattern to test and when the don't care capabilities are needed.

**STATE TRANSITION TABLE**

The state transition table consists of four columns: Present State, Next State, Event(s), and Action(s). The table defines what actions the state machine will do given any input or state. Table II illustrates a three-state example.
For a state machine with \( n \) states, there will be \( 2^n \) possible transitions from the current state to the next state that includes a transition to itself (e.g. B to B). In practice it is very rare that a design will use all of these possible transitions. In our implementation we will use two bits of our \( N \) bit-wide decision word for control. The control bits are not shown in the state transition table. One bit will be used to tell us if the current state to next state transition is a valid transition called the no transfer bit.

How we define the actions or outputs in our state transitions table determines the type of state machine we implement. If each valid current to next state transition has its own action defined, then the machine is classified as a Mealy machine. When the actions are defined uniquely by the state we are transitioning to the machine is a Moore machine. We will use a Mealy machine where for each valid transition there is an associated action defined. It is easy to create a software-based state machine for a Moore machine or a combination of Mealy and Moore.

The Events are a list of transfer conditions which will cause a state transfer to take place. The Events column is expressed as a sum of products of the transfer conditions. In Table II the transition from A to B has two minterm transfer conditions expressed as sum of products: 101, X11. As mentioned earlier, our implementation will be using two bits of a \( N \) bit-wide word for machine control. One bit will be used to tell us if the transition from current state to next state is a valid transition for that particular state machine. The other bit tells when we have reached the last minterm in the sum of products for that current to next state transition.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
<th>Events</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>101</td>
<td>10011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X11</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>NOT VALID</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>X10</td>
<td>00011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1X0</td>
<td>01000</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>100</td>
<td>11100</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>11X</td>
<td>01110</td>
</tr>
</tbody>
</table>

Table II - Mealy State Transition Table

STATE MACHINE

The state machine needs to be executed on a regular basis. This time interval is determined by evaluating the response needs of the system it is controlling and how fast the inputs and outputs can respond. Figure I shows the structure of the state machine. The input processing job is to collect, filter, and preprocess the inputs to the state machine. The input is then placed in its assigned bit position within the \( N \) bit-wide input word. The state transition decision-making block tests all possible state transitions from the current state for a transfer condition. If a transition is found, it will call a function which executes any logic
to perform the actions for that new state. Depending on design needs, the designer can choose not to implement this step and do everything in the output processing block. The output processing block is responsible for taking the outputs defined by the State Transition Table and outputing them.

![Figure I - State Machine Flow](image)

**Input Processing**

This module takes data from the system inputs and places it into the assigned bit position corresponding to same bits in the transfer condition word. It does any preprocessing of the input data necessary including filtering, gathering, level testing and conversions. All data sent to the state machine is digital. Analog values are converted to binary clamped values. The input to the state machine could be one of three values: normal, high, or low. This would require two bits of input into the state machine.

**State Transition Decision Making**

This step takes the state transition minterms which is translated into the decision algorithms transfer condition mask (B) and don't care mask (C) and tests the input against each minterm for a current to next state transfer. This is done for all possible transfers from the current state. The following C code is for illustration of the concepts only. For the best performance, this should be done in assembly code. We will be assuming a 32-bit wide word, which means that the target processor must have 32-bit logic instructions. After reserving the two control bits, we will be able to process 30 bits of inputs in parallel.

```c
#define INT32U unsigned long
#define STATE_A 0
#define STATE_B 1
#define STATE_C 2
/* the following defines assume control bits at bit 30 and 31 */
#define LAST_MINTERM_CONDX 0x80000000
#define LAST_MINTERM_XMASK 0x80000000
#define NO_XFER_CONDX 0x40000000
#define NO_XFER_XMASK 0x40000000

struct b_ { /* packed bits */
    INT32U lastMinterm:1 ,noTransfer:1,
    b29:1,b28:1,b27:1,b26:1,b25:1,b24:1,
    b23:1,b22:1,b21:1,b20:1,b19:1,b18:1,b17:1,b16:1,
    b15:1,b14:1,b13:1,b12:1,b11:1,b10:1,b09:1,b08:1,
    b07:1,b06:1,b05:1,b04:1,b03:1,b02:1,b01:1,b00:1;
};

union ba_ { /* bit access union */
    INT32U lwd; /* 32 bit word access */
}
```
struct b_ bits; /* bit access */
);  

struct ste_ { /* State table entry */
       INT32U condx, /* transfer condition */
       x_mask; /* don't care mask */
    };  

xferTest(struct ste_ *table, union ba_ *input)
{  /* Checks the minterms for a transfer of current state to 
    the next state.
    Returns: TRUE for a valid state transition otherwise FALSE
    Assumes: At least one table entry for lastMinterm or noTransfer */

union ba_ test, lastTableEntry;

do /* check all minterms for a given current to next state */
{
    test.lwd = (input->lwd ^ table->condx) & table->xmask;
table++;
} while(test.lwd != 0);
/* see if we have a transfer or an exit condition */
lastTableEntry = (union ba_)(--table)->condx;
if (lastTableEntry.bits.noTransfer ||
    lastTableEntry.bits.lastMinterm)
    return(FALSE);
else
    return(TRUE);
} 

The xferTest() function takes a pointer to the first minterm of the state to state transfer being tested for transition. It also takes a pointer to the input union which is set up by the processInputs() function. The do while loop runs the decision algorithm on each minterm until the transfer condition test.lwd equals zero. This means that the last entry in the table must always transfer which is why we must add an extra entry into the table. When we exit the while loop, we then test to see if we have a true transfer, a no transfer condition, or the end of the table. The function will return TRUE when we have a valid transfer to next state; otherwise, it will be FALSE.

Next State Execution

The stateMachine() function illustrates how states transfer conditions are tested and how the state functions are called. For the present state we must check all possible transfer conditions. If one of the calls to xferTest() returns TRUE indicating a transfer condition, the state being tested is called. If none of the xferTest() returns TRUE, we will stay in the present state. The state function can process the outputs, or it can leave it to the processOutputs() function. The choice will depend on the type of state machine implemented.

In Table III we used the decision algorithm to create the transfer condition mask (B) and the don’t care mask (C). These will be used to create the tables to be tested by the
xferTest() function. For the unused bit positions, set all values to zero; this will cause them to be don't care bits. This is a tedious task and begs to be automated by another program.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
<th>Events</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>101</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X11</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>NOT VALID</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>X10</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011</td>
<td>011</td>
<td>111</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1X0</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>100</td>
<td>100</td>
<td>111</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>11X</td>
<td>110</td>
<td>110</td>
</tr>
</tbody>
</table>

Table III - Mealy State Transition Mask Table

/* the following table are created from the three state example using the last three bits of the 32 bit word */
struct ste_a_b[] = { {0x00000005, 0x00000007},
                      {0x00000003, 0x00000003},
                      {LAST_MINTERM_CONDX, LAST_MINTERM_XMASK}};
struct ste_a_c[] = { {NO_XFER_CONDX, NO_XFER_XMASK}};
struct ste_b_a[] = { {0x00000002, 0x00000003},
                      {0x00000003, 0x00000007},
                      {LAST_MINTERM_CONDX, LAST_MINTERM_XMASK}};
struct ste_b_c[] = { {0x00000004, 0x00000005},
                      {0x000000004, 0x000000005},
                      {LAST_MINTERM_CONDX, LAST_MINTERM_XMASK}};
struct ste_c_a[] = { {0x00000004, 0x00000007},
                      {0x000000004, 0x000000007},
                      {LAST_MINTERM_CONDX, LAST_MINTERM_XMASK}};
struct ste_c_b[] = { {0x00000006, 0x000000006},
                      {0x000000006, 0x000000006},
                      {LAST_MINTERM_CONDX, LAST_MINTERM_XMASK}};

stateMachine()
{
  processInputs(&inputs);
  switch(presentState)
  {
    case STATE_A:
      if (xferTest(a_b, &input))
        bState();
      else if (xferTest(a_c, &input))
        cState();
      break;
    case STATE_B:
      if (xferTest(b_a, &input))
        aState();
      else if (xferTest(b_c, &input))
        cState();
      break;
    case STATE_C:
      if (xferTest(c_a, &input))
        aState();
      else if (xferTest(c_b, &input))

Output Processing

The outputs can be processed in the state functions of the processOutputs() function. If you design a Mealy machine then it makes sense to create an array of output values for each possible state transition and then process the outputs based on where it transfers state from and to. For a Moore machine it is easiest to process all outputs in the state function.

Control Bits

The state machine input word should always have both control bits set to 1. When the state transition conditions are translated into the two decision algorithm components, the transfer condition (B) and don’t care mask (C), the control bits must be set correctly. For invalid state transitions the transfer condition (B) no transfer bit is set to 1, otherwise 0. For the last minterm the transfer condition (B) last minterm bit is set to 1, otherwise 0. The don't care mask word (C) should always have both the no transfer and last minterm bit set to 1.

Each set of present to next state transition conditions must have an additional B and C with a lastMinterm bit set to 1, the defines for LAST_MINTERM_CONDX and LAST_MINTERM_XMASK show how to do this. When the state transition is invalid, there also needs to be one B and C. The defines NO_XFER_CONDX and NO_XFER_XMASK shows how to set these bits.

CONCLUSION

Although this algorithm seems complicated at first, the payoff is huge performance increases for state machines with many inputs. It is also flexible since it is table driven, we can easily change and update our state machine. New state tables could also be downloaded to make the system programmable. General purpose process control applications particularly will benefit.

The author would be interested in hearing of any new applications of this algorithm or improvements.