ODE : Output Direct State Machine Encoding

Dr. J. Forrest
Dept. of Computation,
UMIST,
Manchester UK.

Abstract

A somewhat novel approach is presented for determining FSM state codes. Instead of producing an assignment designed to minimise the overall logic of the machine, all Moore outputs are converted to state bits. Pure state bits are only introduced as a final resort. This results in very simple output equations at the expense of more complex next state equations. The total number of output and state bits is usually reduced – a feature that has major advantages on most PLDs. Perhaps the greatest advantage, though, is that outputs are glitch-free. The propagation delays for PLD implementations are also minimised.

1. Introduction

This paper is concerned with the implementation of synchronous Finite State Machines (FSMs). FSMs have been long used as part of the digital design process. Increasingly automated, their implementation path consists of two related steps: the derivation of state encodings, and the synthesis of the subsequent combinatorial logic. The steps are related because the encoding method used can have a marked effect on the ability of the synthesis step to minimise the logic, and that in turn is dependent on the underlying technology used for implementation.

The paper first defines the basic FSM terminology used. It goes on to describe the principal points of PLDs and the ODE algorithm itself. Finally, some examples are given and conclusions made.

2. FSM Terminology

FSMs, as first described independently by Mealy[1] and Moore[2], are used frequently in the implementation of digital systems. Both the Moore class and Mealy class of FSMs link inputs to outputs via a simple concept of memory known as their *state*. IIn digital system FSMs, the inputs consist of a number of separate bits or signals – termed the *input variables* (I) – and the “current input” is the set of the values on these variables at the given moment. Similarly for outputs and *output variables* (O). In Moore-class FSMs, the current output is purely a function of the current state, while in Mealy-class FSMs the output is a function of both the current state and the input variables.

The term *Moore Machine* is used as a synonym for Moore-class FSMs. However, the term *Mealy Machine* is used to refer to those Meally-class FSMs that are not Moore machines. Meally machines can react faster than functionally equivalent Moore machines and often use less states. However, Moore machines are often prefered because their design is simpler. Another useful distinction is between *Moore* and *Mealy Outputs*. These terms refer to single output bits, where a Moore output is itself only a function of the current state whereas a Mealy output is a function of the current state and the current inputs. A Moore machine only has Moore outputs, while a Mealy machine typically has a mixture of Moore and Mealy outputs. This paper is primarily aimed a Moore machines but is also of use with the Moore outputs of Mealy machines.

3. Common State Assignment Techniques

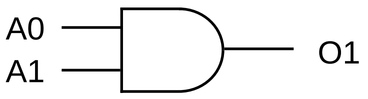
A common aim of state assignment is to produce minimal logic. In traditional practice, the minimum number of state bits is used. For example, if there are five states, three state bits will be used – leaving three potential state codes that will be unused, but are normally taken as *don’t care* during logic minimisation. The original argument behind using as few flip-flops as possible was based on flip-flops being comparatively expensive compared with combinatorial logic. This has long ceased to be the case. A more compelling reason is that manual logic minimisation methods are severely limited in terms of the number of state bits they can handle. With the use of logic minimisation software, the reasoning behind this approach is reduced – although there may well be limits on the number of flip-flop bits available in given environments.

In practice many engineers use more ad hoc techniques for state assignment – such as using grey-code or even normal counting. It is unlikely that these give the best results, but they are quick to use in manual situations. Such practices are common in PLD based designs, which suggests that logic minimisation under such technologies is not critical – this point is returned to below.

One alternative to the above strategy is *one-hot encoding*, where each state is allocated its own state bit which is one when the machine is in that state and zero otherwise. This results in simplified next state logic at the expense of more state bits – a given state bit's next state function will depend merely on the state bits of proceeding states and the associated input variables. Output equations are also simple: equivalent to OR functions of the state bits of the states where that output is asserted, or NOR functions of the states where that output is not asserted – whichever is simplest. However, although the logic is simplified the number of inputs to that logic is often higher than traditional methods. Consider a 16 state FSM where one particular Moore output is high in 8 of the states and low in the others. Using one-hot encoding, the output function will be a predictable 8-input OR function. Using the minimum number of state bits, the worst case is an arbitrary function of 4 inputs, while the best case is that the output directly corresponds to a state bit. Although the internal logic derived using minimum state bits will possibly be more complex, the number of bits used as input to that logic will always be fewer. Under many programmable logic techniques the number of logic inputs is at least as important as the function complexity. For example, with Xilinx 4000 devices[3] any single output function of up to four inputs has an identical cost – because of the lookup table approach used. This contrasts with full custom approaches, where the function itself directly affects the implementation cost. The underlying technology is thus important when deciding factors such as state assignment and minimisation.

3.1 Glitch-less Outputs

A constraint sometimes applied to state machine implementation is that one or more of the outputs must be glitch-free – a glitch being a temporary $0 \to 1 \to 0$ or $1 \to 0 \to 1$ transition as the machine changes state. Such behaviour is usually perfectly acceptable in a purely synchronous system, but rarely is when outputs are fed to asynchronous logic. As a reminder of how easily glitches occur, take the following simple example:



where output *O1* is a simple AND function of state bits *A0* and *A1*. Consider that we are changing from a state where the pair (A0,A1) are (0,1) to another where they are (1,0). In both these states the output O1 should be 0. However, if the physical nature of the circuit is such that the $0 \to 1$ A0 transition arrives at the gate before the $1 \to 0$ A1 transition, there will be a short time when both inputs are 1, and thus O1 will be too.

One way around this problem is to derive a state assignment such that there is only one bit difference between the state codes on each possible state transition. That is, to ensure that only one state changes on each next state transition – it is multiple bit changes to combinatorial logic that causes the problem. Although this does provide a solution, assignments that fulfil this constraint do not exist for every state machine. It is also likely to run counter to the requirements of minimisation.

A common engineers' solution to this is quite simple – if an output is depending on only one state bit, then only one bit can change and thus no glitches can occur. The requirement is thus to derive a state assignment such that there is a state bit that directly reflects each output bit that must be glitch free – either by holding the required value or its complement. The proposal in this paper directly follows this practice, but instead of being used for only those outputs that must be glitch free, it is used for all Moore outputs. To see the benefits of this approach, it is useful to look at the characteristics of programmable logic devices – which have proved ideal targets in practice.

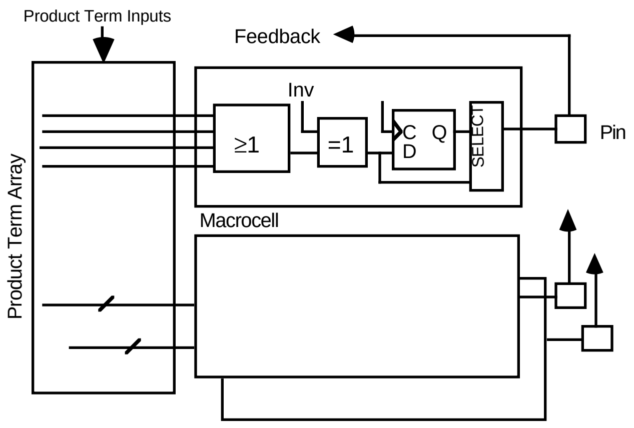


Fig 1: Basic PLD Model

4. Programmable Logic Devices

This section presents a simplified picture of Extended Programmable Logic Devices (PLDs), in order to see how these influence the state assignment and logic minimisation processes. EPLDs consist of a number of PAL type blocks, as depicted in Fig 1. Each block contains a product term array, with a fixed number of input signals that provide all the combinatorial logic for the block's macrocells – a macrocell being linked to an I/O pin and containing a D-type flip-flop. In each macrocell a fixed set of product terms are either ORd or NORd together, the result being passed to the pin, optionally through the flip-flip. The result is fed back to be used as an input to the product term array – alternatively the macrocell can be disabled to provide pure inputs.

The numbers of product term arrays and macrocells, along with the assignment of macrocells to arrays and the maximum number of inputs to each array, are features of each device and cannot be exceeded – unless a larger device can be used. Similarly for the number of product terms used per macrocell, although many devices allow for a macrocell to use some unused terms that notionally belong to its neighbours.

Real devices are more complex than this. A major variation being to allow macrocells to be used independently of I/O pins – denoted as *buried*, since they cannot be observed directly. Nevertheless, the above model is sufficient to consider the effects of state machine implementation techniques. Typical examples of these devices include Altera (née Intel) Flexlogic[4] and AMD Mach 3 and 4 devices[5]. The work described here was developed using Flexlogic devices. Since all programmable devices have fixed resources, it is important to know which of these constraints is broken most commonly – since minimisation approaches should emphasise that quantity before others. Prior to developing ODE, it was the experience of the author with Flexlogic devices, that the resources ran out first in descending order of frequency as follows:

C1. Macrocells – the total number of macrocells used in designs.
C2. Product term array inputs – the total number of inputs used in the associated macrocell equations.
C3. Product terms per macrocell.
C4. Total I/O pins required.

Category C3 is rarely exceeded in practice, and if so it is often related to arithmetic or complex XOR terms, for which these devices are not well suited. These observations were derived from implementing control style functions – including state machines and related functions such as MUXs – and have held true over several designs. Where there are problems with C3, it has been possible to factorise the equations, although this does in itself use extra macrocells. The implications of factorised logic will be ignored here, to simplify comparisons.

The total macrocells used for implementing an FSM is the sum of the state and output bits. It will be noted that one-hot encoding essentially maximises this figure. It is often argued the number of state bits is not so important, as they can use buried macrocells which are less critical. However, the above evidence suggests that it is macrocells per se, rather than the pins, that are the constraint. As EPLDs grow larger, it is important to use them to implement not just state machines and decoders, but related functions that were previously allocated to separate devices – these include registers, counters and other functions that directly relate to state machines. By placing these functions on the same IC as the state machines, the integration of the system is improved. There are many other uses for buried macrocells than state bits, and it is important to minimise their use. One idea is to note where a Moore output corresponds exactly to a state bit – since the same macrocell can be used for both. The approach used in this paper, described in the next section, takes this to extreme – all Moore outputs being used as state bits.

5. The ODE Algorithm

The Output Direct Encoding algorithm is inherently simple:

1. For each of the FSM's Moore outputs O_i, $i \in \{0..n-1\}$ – n being the number of Moore output bits – allocate a state bit X_i. An x-code is allocated to each state by giving X_i, $i \in \{0..n-1\}$, the value of O_i in that state.
2. Calculate array *U*, which is the usage counts of each distinct x-code, and *u* the largest element in U. Then calculate *v*, where v is the smallest number where $2^v \geq u$.
3. If $v>0$, allocate extra state bits Y_j, $j \in \{0..v-1\}$. Allocate a y-code to each state, such that the first state with a particular x-code has a numeric y-code of 0, the second 1, the third 2, etc.
4. If $v=0$, the y-code of each state is a null string.
5. The state code of each state is the concatanation of its x- and y-codes.

In practice, one makes each output a state bit, and allocates extra state bits to yield unique state codes. It is vital that the state bit reflects the required value, and not its complement. That way, the state bit can use the same PLD macrocell as the output. This has two advantages: the number of macrocells is reduced, and the outputs are taken directly from flip-flops without passing through a product term array again. On an iFX780-10 the direct outputs produced using ODE are valid 6.5ns after the clock input, while outputs that require an additional combinatorial function take 16.5ns[4]. Another benefit of direct outputs is that all the Moore outputs are automatically glitch-free.

Mealy outputs can be treated as they would using other state machines – by generating the output equations using the state code. However, if a Mealy output equation F can be rewritten:

$$F = A_0Z_0 + A_1Z_1 \ldots$$

where A_i are input conditions and each Z_i a state bit or its complement, then F will be glitch-free if only one of these state bits changes on each cycle and the associated condition code is stable whenever the Z bit is true. In many situations it is worth adding extra state bits to ensure this property. These Z bits are not true outputs, and can be implemented using buried macrocells. There should probably be a step in the synthesis procedure to see if these extra bits are equivalent to one of the true outputs – in which case they can be replaced by that output. This is currently performed manually.

6. Sample Results

The purpose of this section is not so much to demonstrate that the algorithm works, but to compare the outcome with alternative approaches. Two examples are presented – the commonly used Mead and Conway traffic lights controller [6] and one from a real design

The traffic light system FSM is described in table 1c. This table shows the original version of the machines, where ST is a Mealy output and the other outputs Moore. No attempt has been made to ensure that ST is glitch free. Table 1b shows a number of sample state encodings that have been derived for this machine. The straight encoding counts up from 0 in the order of the states. Jedi[7] is a state assignment program. With assignment 3, jedi was told to use 4 state bits instead of the default two. Assignment 4 is the ODE encoding, that directly reflects the Moore outputs. Assignment 5 is a simple one-hot version – assigned manually. Espresso[8] was used to minimise the state machine's combinatorial logic, using each of the above assignments. Throughout this section, the -Dso_both flag was used with Espresso, which forces it to look at each of the outputs individually, including working out whether to take the complement of the function, and is the most appropriate for the PLD model described above – by default Espresso is aimed at PLA implementations.

Inputs					Outputs				
C	TL	TS	Present State	Next State	ST	HL0	HL1	FL0	FL1
0	-	-	HG	HG	0	0	0	1	0
-	0	-	HG	HG	0	0	0	1	0
1	1	-	HG	HY	1	0	0	1	0
-	-	0	HY	HY	0	0	1	1	0
-	-	1	HY	FG	1	0	1	1	0
1	0	-	FG	FG	0	1	0	0	0
0	-	-	FG	FY	1	1	0	0	0
-	1	-	FG	FY	1	1	0	0	0
-	-	0	FY	FY	0	1	0	0	1
-	-	1	FY	HG	1	1	0	0	1

Table 1a - Traffic Light State Transisition Table

	Description	State Encodings HG/HY/FG/FY
1.	Straight	00/01/10/11
2.	Default Jedi/Armstrong	00/01/11/10
3.	Jedi with 4 state bits	1010/1011/0011/0010
4.	ODE	0010/0110/1000/1001
5.	One-Hot	1000/0100/0010/0001

Table 1b Traffic Light Assignments

	Description	Total PT	Harm Mean PT	Max PT	Literals	No. Inputs	No. Macro-cells
1.	Straight	13	1.40	3	29	5	7
2.	Jedi/Armstrong	14	1.39	5	34	5	7
3.	4 bit Jedi	14	1.39	5	34	7	9
4.	ODE	15	2.85	4	37	7	5
5.	One-Hot	17	1.45	3	37	7	9

Table 1c Traffic Light Results

Inputs								Outputs				
Ref	Ram	DS	AS	Had Ref	Sam Pag	Present State	Next State	U/L	RCas	Ras	Cas 01	Dsack
1	-	-	-	-	-	Idle	Rf_Cas	1	1	1	1	1
0	0	-	-	-	-	Idle	Ras	1	1	1	1	1
0	1	-	-	-	-	Idle	Idle	1	1	1	1	1
-	-	-	-	-	-	Rf_Cas	Rf_CasRas	1	0	1	1	1
-	-	-	-	-	-	Rf_CasRas	Rf_CasRas2	1	0	0	1	1
-	-	-	-	-	-	Rf_CasRas2	Idle	1	1	0	1	1
-	-	1	-	-	-	Ras	Ras	0	1	0	1	1
-	-	0	-	-	-	Ras	RasCas	0	1	0	1	1
-	-	-	0	-	-	RasCas	RasCas	0	1	0	0	0
1	-	-	1	-	-	RasCas	Idle	0	1	0	0	0
0	-	-	1	1	-	RasCas	PageIdle	0	1	0	0	0
0	-	-	1	0	-	RasCas	Idle	0	1	0	0	0
1	-	-	-	-	-	PageIdle	Idle	0	1	0	1	1
0	0	-	-	-	-	PageIdle	PageIdle	0	1	0	1	1
0	1	-	-	-	0	PageIdle	Idle	0	1	0	1	1
0	1	1	-	-	1	PageIdle	PageIdle	0	1	0	1	1
0	1	0	-	-	1	PageIdle	RasCas	0	1	0	1	1

Table 2a DRAM Controller State Transition Table

		State Encodings
	Description	Idle/Rf_Cas/Ras/Rf_CasRas/Rf_CasRas2/RasCas/PageIdle
1.	Straight	000/001/010/011/100/101/110
2.	Default Jedi	001/000/101/100/110/111/011
4.	ODE	111110/101110/010110/100110/010000/010111

Table 2b DRAM State Assignments

	Description	Total PT	Harm Mean PT	Max PT	Literals	No. Inputs	No. Macro-cells
1.	Straight	23	1.58	6	74	9	8
2.	Default Jedi	19	1.49	6	55	9	8
4.	ODE	23	3.27	6	88	12	6

Table 2c DRAM Results

```
S_IDLE        = cUL * RCAS * RAS * CAS01 * DSACK1 * /DS0
```

Fig 2. PldShell Example

The results are shown in Table 1c. This shows total product terms, the harmonic mean and max (greatest for a single output) figures, the literal count, the total inputs to the combinatorial logic and the total macrocells used. The harmonic mean gives an indication of the typical product term usage per macrocell – for example, row 2 has a better harmonic mean value than row 1, indicating that in most circumstances the product term usage is better, even though the max figure is worse. It will be readily seen that in terms of both product term count and literals, which are common measurements of implementation complexity, ODE does not score particularly well. However, it uses the fewest macrocells, which is the main argument. Furthermore, the other figures are perfectly acceptable – so using such a different approach is not a problem. Although the mean product term usage is the highest, the maximum figure is not – this should just be seen as a bonus. On a different note, it should be pointing out that two of the four bits from assignment 3 turned out to be unused in the minimised logic – so assignments 2 and 3 are equivalent.

The second example is the DRAM controller for a 68332 microcontroller based system[9]. The state transition table is given in Table 2a. It should be noted that some of the “inputs” are expressions in the real design, for example IsDRAM is a function of AS and the appropriate address lines, but are treated as single inputs here for sake of simplicity. Also some of the signals are active low. As a final note, the real CAS signals are themselves functions of RCAS and CAS01:

CAS0' = RCAS' + IsUpper.CAS01'

This is to avoid glitches, as described above. It would have been possible to rewrite the DRAM controller itself to avoid this step, but the state machine would have been more complex – this was a design decision. This equation gives glitch-free output if RCAS and CAS01 are glitch-free – since at most one of these signals changes on each state transition and IsUpper is known to be stable throughout memory cycles.

Table 2b describes the state encodings that were tried on this machine. Unlike the previous example, the ODE assignment includes a y-code of length 1 – in order to differentiate between states RasCas and PageIdle. The list of assignments tried was shorter – since neither jedi with longer codes nor one-hot encoding seemed fruitful. The constraint of making RCAS, CAS01 and RAS glitch-free was not applied to assignments 1 and 2.

Table 2c shows the results achieved. Again the ODE method comes out worse in terms of the literal count, and its literal count is higher than optimum too. However, the macrocell count is down – which is the general aim.

Other examples have been tried, with similar results.

6. Conclusions

A new technique for state assignment has been introduced. While being essentially an extension of engineering techniques, what is new is the degree to which this has been applied. The goal is to reduce the macrocell usage on PLDs, and at this it has been shown to succeed. It might have been assumed that this would lead to an unacceptable rise in other critical values, such as product terms and inputs to the product term arrays. However, this has been shown to be not the case – at least on Flexlogic ICs. As evidence for this, it should be noted that both of the 68332 based examples were originally implemented using the Intel PldShell tool[10]. Although this uses Espresso to minimise the logic, it does not generate don't cares from unused state codes – the product term counts are much more complex than those presented here, yet there have been few resulting problems. However, ODE is advantageous not only because of the reduced macrocell count but also because Moore outputs appear more quickly than other methods and are always glitch-free.

The approach is not currently fully automated in a conventional sense. The PldShell tool does not itself perform state assignment, so there is no way of describing states machines with non-encoded symbolic names. Instead the symbolic names are given together with the assignment as an input to the tool. The manner adopted with ODE is to describe the outputs as flip-flops and then to describe the required output values in the state code – thus the ODE description is given directly. For example, the IDLE state assignment for the second example is given in Fig 2. There are no distinct output equations, and the next state equations are given using standard PldShell techniques. This ease of description was one of the original reasons for developing the approach – before it was realised that there were other advantages.

It should not be assumed that ODE algorithm is universally applicable. Machines with many states that have identical outputs are likely to suffer from the naive approach to generating the y-codes. The author uses the ODE technique almost universally now on PLD designs, but not on counters and similar state machines which have few outputs. More work may lead to a better generation of the y-code.

Other areas for further work involve attention to the number of inputs to the product term arrays – the next most critical constraint. It is known that there are scenarios where some of the fedback outputs could be removed during logic minimisation – that is, the logic could be rewritten without them. This may involve using extra or more complex product terms, but that is not a major constraint. Such minimisation would need to follow an initial mapping phase, to allow simultaneous optimisation of physical groups of macrocells, and would require modifications of tools such as PldShell. However, there could be major benefits. Using ODE, the product term array input number has proved a problem on more occasions – this is to be expected, although it has not been as bad as was initially assumed. Improved logic minimisation should help counter this.

It is highly likely that tools exist that produce a similar state assignment to the ODE technique – under the guise of relating state assignment to the system outputs – although the author has no experience of any. What this paper has tried to demonstrate is that far from being an obscure approach, with occasional applications, it such be used as the standard method on EPLD based systems.

References

1. *A Method for Synthesising Sequential Circuits*, G. H. Mealy, Bell System Tech. J., no. 34 (Sept. 1955).
2. *The Synthesis of Sequential Switching Circuits*, D.A. Huffman, J. Franklin Inst, no. 257 (March-April, 1954).
3. *The Programmable Logic Data Book,* Xilinx, 1994.
4. *Programmable Logic*, Data book, Intel, 1994.
5. *Mach® 3 and 4 Family Data Book*, Advanced Micro Devices, 1993.
6. *Introduction to VLSI Systems*, C. Mead and L. Conway, Addison-Wesley, 1980.
7. *Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages*, B. Lin and A. R. Newton, VLSI 89, Munich, 1989.
8. *Logic Minimization Algorithms for VLSI Synthesis*, R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, Kluwer Academic Publishers, 1984.
9. *MC68332 User's Manual*, Motorola, 1990.
10. *PLDshell Plus™ User's Guide v4.0,* Intel, 1994.