Low-Power State-Encoding for Partitioned FSMs with Mixed Synchronous/Asynchronous State Memory

Cao Cao and Bengt Oelmann

Abstract: Partitioned Finite-State Machine (FSM) architectures in general enable low-power implementations and it has been shown that for these architectures, state memory based on both synchronous and asynchronous storage elements gives lower power consumption compared to the fully synchronous ones. In this paper we present state-encoding techniques for a partitioned FSM architecture based on mixed synchronous/asynchronous state memory. The state memory here is composed of a synchronous local state memory and a global asynchronous state memory. The local state memory is shared by all sub-FSMs and uses synchronous storage elements. The global state memory is operating asynchronously and is responsible for handling the interaction between the different sub-FSMs. Even though the partitioned FSM contains asynchronous mechanisms, its input/output behavior is cycle by cycle equivalent to the original monolithic synchronous FSM. In this paper we study state encoding for partitioned FSMs that have been partitioned according to their state-transition probabilities. For the local state assignment we present a, what we call, state-bundling procedure to enable states residing in different sub-FSMs to share the same state codes. Two state-encoding techniques, one based on binary encoding and one optimized for low-power consumption, are compared.

1 Introduction

Dynamic Power Management (DPM) is a commonly used approach for low-power optimization on the Register-Transfer (RT) and architectural level [1]. The objective for a DPM scheme is to shut-down the parts of the design that are temporarily idle. By shutting down a part it is meant that the power dissipation is reduced for that part. For reduction of dynamic power dissipation, clock-gating and input-disabling is used. For reduction of leakage currents, such as subthrehold and diod leakage, various approaches have been proposed in the literature, e.g. [2]. Common for all these techniques is that mechanisms for detecting idle conditions of the different units are added to the design. Also means for shutting down the units are added. Implementation of these will result in additional circuits that will add to the circuit area and power dissipation. Before introducing shut-down circuits in the design, careful analysis must be made to achieve a solution with as low power consumption as possible. The objective for a power optimization procedure is to find the most beneficial idle conditions, taking the overhead into account. Complex designs, composed of several functional units, such as microprocessors that are composed of large functional units like floating-point unit and cache memory can be temporarily shut-down when not used. For a given architecture, this kind of coarse-grained DPM is possible to implement manually by the designer thanks to the small number of places shut-down circuits are introduced and to the fact that the different units are functionally well separated and therefore easy to identify. When applying DPM to a single functional unit, the unit has to be partitioned into two or more sub-units where each of them can be individually shut-down. The unit is decomposed in such a way that the lowest possible power consumption is achieved. This type of fine-grained DPM requires an automated optimization procedure since the optimum decomposition is not necessarily made according to the functionality it is therefore not obvious to know how to make the decomposition. The most commonly used approaches for power optimization procedures, takes a behavioral description of the design and seeks the optimum, or near-optimum solution, for a pre-defined architecture. For data path units (combinational logic) the precomputation-based logic has been proposed [3]. The idea is to pre-compute a part of the function one clock cycle ahead in order to gate the clock signal to the register holding the inputs to the combinational logic and thereby reducing the average switching in the logic. Different architectures have been proposed that block either all inputs or a subset of the inputs [4]. This approach can also be used for synchronous FSMs. For low-power FSM design Benini et al. presented an approach called computational kernels [5]. From the State Transition Graph (STG) of the FSM, a sub-FSM is extracted that implements the function of the FSM for a subset of its states whose steady-state occupation probability is high. When the FSM is in one of these states a smaller and less power-consuming circuit is used (the kernel) and otherwise the original function is used. Chow et al [6] propose an implementation architecture that resembles of the one used for computational kernels. They propose a decomposition model for multiple coupled sub-FSMs. A shared state memory stores two sets of states, the original states and additional states that are used for determining which one of the different sub-FSMs that is active. For state-encoding they present a method that consider the crossing transitions (transitions where the source and destination state do not reside in the same sub-FSM) is used. In contrast to the shared state memory architecture, an architecture with separate state memory, one for each sub-FSM, has been used by for example [7,8]. For state encoding and other optimizations, each sub-FSM can be separately optimized using standard methods. The disadvantage is that the circuit area for the state memory becomes larger compared to using shared state memory.

The approaches to low-power FSM design described above, all assume fully synchronous implementations. For both architectures, based on shared and separate state memory, fully synchronous implementations have disadvantages. For the cycle when a crossing transition occurs, the two sub-FSMs involved, both have to be clocked which is very power consuming. For partitioned FSMs with separate state memory an asynchronous hand-over mechanism has been proposed that removes the requirement of clocking two sub-FSMs at a crossing transition and thereby the power-overhead introduced for managing the interaction between the sub-FSMs can be reduced. In [9] it has been shown that asynchronous control for sub-FSM interaction is 5.8 times more power efficient when idle compared to synchronous control.

In [8] it was demonstrated that automated synthesis for low-power FSMs based on a mixed synchronous/asynchronous architecture with separate state memory achieved power reductions of 45% in average for a set of FSM benchmark circuits. For a recently presented decomposition model [10] for FSMs with shared state memory power, reductions of 56% in average. Here, state encoding optimizations for low power was not considered.

In this paper, a novel low-power state encoding algorithm for coupled FSMs is proposed and applied to partitioned FSMs based on mixed synchronous/asynchronous state memory. The main contributions of this paper are the following:

- A state assignment procedure: State bundling enables crossing transitions in one single clock cycle (or in other words, only one sub-FSM has to be clocked).
- Power-optimized state encoding: A computational efficient state-encoding algorithm for coupled FSMs.
- Demonstration of efficiency: The algorithms presented have been implemented in a tool for low-power synthesis of partitioned FSMs and it is demonstrated that the state-encoding algorithm leads to power reductions of 6% in average for low-power partitioned FSMs originating from the MCNC benchmark circuits. The total average power reduction that is the result from both partitioning and state-encoding is 59%.

The outline for the rest of this paper is as follows: The next chapter introduces the partitioned FSM implementation architecture with a focus on the organization and operation of the mixed synchronous/asynchronous state memory. In chapter 3 the basic binary state encoding procedure and our procedure that we propose for power optimized state encoding are presented. In chapter 4 some experimental results from automatic synthesis of a set of FSM benchmark circuits show the possibility of reducing the power consumption in a partitioned FSM by using power-optimized state encoding after partitioning. In chapter 5 we conclude the paper by a discussion regarding the limitations of the two step approach with a partitioning step followed by a state encoding step.

2 Partitioned FSM with Mixed Synchronous/Asynchronous State Memory

2.1 Implementation Architecture

The straight-forward way to implement a partitioned FSM is to have separate state memory for each of the sub-FSMs, see Figure 1a. From state encoding point of view, state-encoding is made separately for each of them and well-established optimization algorithms can therefore be used. Since only one of the sub-FSM is active at a time, the state memory can be shared by all the sub-FSMs. The main advantage with shared state memory is the reduced area for the state memory, see Figure 1b. There is, however, a need for a global state memory determining which one of the sub-FSMs is for the moment active. For power-optimized state encoding, state-transition probabilities of the crossing transistions must be considered which is not the case for the separate state memory implementation. For a synchronous solution the global state memory needs to be clocked by the system clock signal that cannot be gated and will therefore increase the power consumption substantially, especially for a partitioned FSM composed of large number sub-FSMs. The architecture considered in this paper is a mixed synchronous/asynchronous architecture developed in [10] that has a shared local state memory (LSM) with a global asynchronous state memory (GSM). The basic idea is to have synchronous local state memory in the part always clocked and asynchronous memory for the global state memory. The partitioned FSM is made on the basis of the state transition probabilities which results in clustering of states with high probabilities that will be implemented in the same sub-FSM. The state-transition probabilities between the sub-FSMs will be of low probability and hence the state-change probability is low for the global states which make an asynchronous implementation power efficient [12].

2.2 STG decomposition

In this section the decomposition of the STG of the monolithic FSM for the architecture described in the previous section is presented. To describe to basic ideas of the design model for STG decomposition, the example in Figure 2 will serve as an illustration.

The initial monolithic machine is decomposed into two separate sub-FSMs F^1 and F^2 as indicated in Figure 2a. We can see that there are two crossing transitions, one from s_2 in F^1 and one from s_5 in F^2. For each crossing transition, an additional g-state is introduced and the source state of the original crossing transition will have that as destination state. In Figure 2b the destination states of the crossing transitions from s_2 are changed from s_3 and s_4 to g_3 and g_4 respectively. A crossing transition is completed by the following sequence of events. When the machine enters a g-state this is detected and the global state, denoted *R*, of the partitioned FSM will change. The global state is pointing out which one of the sub-FSMs that is active. A change in the global state will deactivate the sub-FSM containing the source state and activate the sub-FSM containing the destination state. Consider the crossing transition from s_2 to s_3 in the example. The transition from s_2 will enter g_3. This will cause the global state *R* making a transition from r_1 to r_2. The global state will after completion of the crossing transition point out F^2 as the active sub-FSM and not F^1 as before. In a synchronous FSM the crossing transition, as all transitions, must be completed within one clock cycle. From the example above it can be seen that a crossing transition requires two state transitions which will take two clock cycles to complete in a synchronous machine. To solve this, the transition from the g-state to the entry state of the destination sub-FSM (e.g. g_1 to s_1) is made asynchronously. By that it is meant that the transition is triggered by a signal transition rather than by the active edge of the clock signal. A control signal, decoded from the g-state, makes the global state to change. The states originating from the initial FSM and the additional g-states are stored in a state memory clocked by the common clock signal. We call this local state memory. A global, asynchronous, state transition does not permit a local state change which puts a restriction on the state encoding. The local states must be coded in such a way that the code for a g-state and its associated entry state must be identical. From the example in Figure 2b, the following pairs of states, that we call coupled-states, must have identical codes: (s_1,g_1), (s_3,g_3), and (s_4,g_4). The states s_2 and s_5 may share the same state code since they are located in different sub-FSMs and distinguished be the global state.

A, what we call, a coupled-state table describes the behaviour of the decomposed FSM including the sub-FSM interaction. To illustrate the construction of the coupled-state table the example from Figure 2 is used. Its coupled-state table is shown in Figure 3. Each row in the table holds all states for one sub-FSM and each column represents a bundle of states that will have the same local state code. A sequence of state transition $s_1 \to s_2 \to s_3 \to s_5$ will result in the following sequence in the partitioned FSM $s_1 \to s_2 \to g_3 \to s_3 \to s_5$ where the transition $g_3 \to s_3$ is asynchronous. In the table, a local state transition is represented by a horizontal change and a global state transition is represented by a vertical change.

The coupled-states will of course impose restrictions on the state encoding of the local states because it contains information about how the different sub-FSMs are related.

3 State Encoding for Local States

After the FSM partitioning, the state encoding is performed in two steps. First the coupled-state table is build by locating the coupled-states together in bundles. After that the total number of bits in the local state memory is minimised by the "coupled-state merging" algorithm which also takes the state-transition probabilities into account in order to reduce the power. In the second step state-codes are assigned to each bundle. State encoding is made for one sub-FSM at a time, starting with the most active sub-FSM.

3.1 Basic Definitions

The monolithic Mealy-type FSM is defined as a sextuple: $F = (S, X, Y, \delta, \lambda, s_0)$ where *S* is the set of states, *X* is the set of binary inputs, *Y* is the set of binary outputs, δ is the transition function, λ is the output function and s_0 is the initial state.

Let there be a partition on the set S: $\Pi = \{S^1, S^2, \ldots, S^n\}$ where Π is defined as a collection of subsets such that $\bigcup_{m=1}^{n} S^m = S$ and $S^i \cap S^j = \varnothing$ for $i \neq j$ where $1 \leq i, j \leq n$.

The monolithic FSM is decomposed into a set of sub-FSMs where every subset $S^i \in \Pi$ defines a sub-FSM as: $F^m = (S^m, X^m, Y^m, \delta^m, \lambda^m, s_0^m)$. We call states S^m internal states of the sub-FSM. X^m is the set of input variables at all transitions from the states in S^m, and Y^m is the set of outputs variables on the sets S^m and X^m.

We define a set of states, $T(S^m)$, not included in F^m to which there are transitions from the states of F^m : $T(S^m) = \{s_j \mid \delta(s_k, X_h) = s_j, s_j \notin S^m, s_k \in S^m\}$.

$Q(S^m)$ is defined as the set of states in F^m where there are transitions from other sub-FSMs as: $Q(S^m) = \{s_j \mid \delta(s_k, X_h) = s_j, s_j \in S^m, s_k \notin S^m\}$.

For the above defined sets we will use the shorter notations T^m, Q^m.

The set of g-states G^m, that reflects the set of destinations states of the crossing transitions in F^m is defined as: $G^m = \{g_i \mid s_i \in T^m\}$.

Let the set of local states in the transformed network of F^m to be U^m: $U^m = S^m \cup G^m$.

3.2 State Bundling

There are two reasons for state-bundling: 1) it enables state in different sub-FSMs to share state codes and 2) it enables an efficient asynchronous global state transition. In the state encoding step the state bundles are considered as states. In this section the criteria and procedures for state bundling will be introduced. First a basic procedure is introduced which will give good results for most partitioned FSMs. Then a procedure for merging the coupled-states into the same bundles is presented. This will for even exceptional cases give improved results.

3.2.1 Basic algorithm

We start with the following example of a partitioned FSM. Let there be a partition $\Pi = \{S^1, S^2, S^3, S^4\}$ which results in the following local sets of states: $U^1 = \{s_1, s_2, s_3, g_4\}$, $U^2 = \{s_4, s_5, s_6, g_7\}$, $U^3 = \{s_7, g_1\}$, and $U^4 = \{s_8, s_9, s_{10}, s_{11}, s_{12}, g_1, g_5\}$. The duty time of each partition U^m, or the probability of the corresponding sub-FSM to be active, is given by the sum of the static state probability of states inside the partition, that is $T(U^m) = \sum prob(s_i), s_i \in S^m$. In the rest of the paper, it is denoted as T^m.

State bundling starts from the coupled-states, the states that are the source and destination states of an asynchronous transition. From previous discussion we know that these have identical state codes. We construct a table of *n* rows for an n-way partitioned FSM where each column represents a bundle of states that after state encoding will have the same state code. The set of bundles *B* needed is defined as $B = \{b_1, b_{2,} \ldots, b_p\}$, where $p = \left| \bigcup_{m=1}^{n} Q^m \right|$. In other words, the number of bundles needed for the coupled-states are the sum of the entry-states of all sub-FSMs. Two probabilities are defined that reflects the property of state bundles. *State bundle probability* is defined as: $prob(b_m) = \sum prob(s_i), s_i \in b_m$. *Bundle transition probability* is defined as: $prob(b_m b_k) = \sum prob(s_i s_j), s_i \in b_m, s_j \in b_k$ describes the probability for a state transition between states in the bundles b_m and b_k. The states are aligned in columns in such a way that all states coupled to each other reside in the same column. In Figure 4, showing the state table for our example, the entries for the coupled-states are shaded grey. We can see that for example s_4 in F^2 is in the same column as g_4 in F^1, i.e. s_4 and g_4 are coupled-states. The state bundling procedure first adds the bundles containing coupled-states and thereafter states not coupled may be freely positioned in any bundle as long as all states residing in the same sub-FSM have unique state codes. The pseudo-code for the bundling algorithm is shown in Figure 5.

The efficiency of this procedure is dependent on the ratio of the number of coupled-states to the number of *free states* given by: $c = \frac{\left|\bigcup_{m=1}^{n} Q^m\right|}{\left|\bigcup_{m=1}^{n} S^m \setminus Q^m\right|}$.

For most partitioned FSMs, partitioned according the state transition probabilities, have small numbers of crossing transitions and will therefore have small *c*. For that reason this basic state bundling procedure works well in most cases.

3.2.2 Merged coupled-state algorithm

Using the basic bundling algorithm for FSM partitions with large *c* will result in large local state memory. However, the number of clocked state memory bits for each sub-FSM will not necessarily be all state bits. The objective of merging coupled-states is to reduce the total number of state bits. To illustrate the merged coupled-state algorithm we use the example in Figure 6 that have a $c = 5/2$.

The initial coupled-state table, before merging the coupled-states, is shown in Figure 7a, where the five g-states reside in five different bundles. Fixed state codes for the state bundles are assumed were the bundle index indicate the binary value of the code (b_0 has the code “000”, b_1 “001” and so on). The merging procedure is performed in the following steps.

1) The objective of the *Sort()* function is to introduce prioritization among the sub-FSMs. It sorts the sub-FSMs according to the descending order of their duty time T^m. Since the sub-FSMs with high duty time generally contribute more to the final power dissipation, they are given higher priority in the coupled-state merging

step. The sorted coupled-state table is shown in Figure 7b. After that, the coupled-state with the highest state bundle probability is moved to the b_0 bundle that will always be assigned the state code “zero” after state encoding. The objective is to minimize the switching activity in the next-state bit-lines for crossing transitions. The reason for this is that a deactivated sub-FSM’s next-state is always encoded to “zero” in order to enable efficient implementation of merging the next-state variables of the different sub-FSMs [10] by using OR gates.

2) In order to reduce the number of bits in the local state memory, the algorithm merges two or more coupled-states into the same bundles when possible. The algorithm first tries to merge the coupled-states in bundles to the right of the leftmost bundle (b_0) in the sorted coupled-state table. In the cases where only one of two or more coupled-states can be merged, the one in the bundle with highest state bundle probability is chosen. After a merging has been completed, the table is sorted again as described in step 1. When no more coupled-states can be merged into b_0 it is locked. Now the same procedure is done for b_1 and continues until the last column has been reached. In the example given in Figure 7, it is shown that both b_2 and b_3 can be merged into b_0. Because the state bundle probability of b_3 is 0.3 ($\Pr ob(b_3) = prob(s_4) = prob(T^4)$), higher than that of b_2 ($\Pr ob(b_2) = prob(s_3) \leq prob(s_3) + prob(s_2) = prob(T^3) = 0.2$), b_3 is chosen to be merged into b_0. The updated coupled-state table after merging is shown in Figure 7c), where the total number of state bundles is reduced from 5 to 4.

3.3 Basic State Encoding Algorithm

The basic state encoding algorithm is a straight-forward technique that does not consider power optimizations at all. It takes the initial coupled-state table without merging and put free states (S^m) into the bundles starting from bundle b_0. The whole state bundling algorithm is given in Figure X. Each bundle is assigned the binary code that corresponds to its index. Binary-encoding makes sure the number of clocked local state bits in each sub-FSM is minimal.

3.4 Power Optimized State Encoding

Since we consider the state code assigned to the bundles to be fixed, the task of state encoding optimization is to move states to suitable bundles in order to reduce the switching activity in the state bit lines.

We first consider coupled-states in the table (Figure 7d). Since every bundle is given a unique state code and can be viewed upon as a state, the algorithm tries to reduce the switching activity in the transitions between these bundles. At the same time, the algorithm tries to keep the sub-FSMs with higher duty time to minimum-length encoding. The merging algorithm, described in the previous section, has sorted rows in descending order of the duty period. Therefore, encoding starts from the top row. For each row, the position of state bundles will be optimized first and then locked, which will not be changed afterwards. A greedy algorithm is used to minimize the hamming distance for the bundle transition probability. The algorithm is shown in Figure X. We illustrate the procedure through an example. In Figure X, the initial coupled-state bundles have been built including b_0, b_1, b_2, b_3. As mentioned before, b_0 is the state bundle with highest state bundle probability and its position is locked initially. We start the state bundle optimization from b_1 because it is the only bundle besides b_0 that has a valid state in the top row representing sub-FSM F^1. To make coupled-state bundles in F^1 use the minimum length codes, b_1 can only be assigned the code “01” and thereby the coupled-state bundle in F^1 only use one state bit. Since the position of b_0 and b_1 is locked after the assignment of F^1, only b_2 and b_3 coupled-state bundle are left. In F^4 the number of minimum local state bits needed for the state bundles is 2 (obtained from minimumCodeLength() function in Figure 10). Since the codes “00” and “01” already has been assigned for b_2 and b_3, the only possible codes are “10” or “11”. We compare the bundle transition probability of b_2 and b_3 with already assigned state bundles b_0 and b_1. If the transition probability between b_3 and b_0 is assumed to be the highest, we assign b_3 to the position “10” which has the hamming distance of 1 to b_0; b_2 is subsequently assigned the code “11”. Since all state bundles have been assigned, state encoding for the coupled-state bundles is completet. The result of the coupled-state encoding optimization is shown in Figure 11a) where the position of b_3 and b_2 has been swapped.

The next step is to encode the *free states*, i.e. states not coupled to any other sub-FSM. Since these are internal states in a sub-FSM, each sub-FSM can be separately optimized in an arbitrarily order. In each sub-FSM, one single free state is considered at a time. That is the one having the highest state transition probability to a certain state in this sub-FSM or to a state in another sub-FSM.

Constrained by minimum-length encoding, the algorithm minimises the hamming distance for local state transitions with high state transition probabilities. For example, in sub-FSM F^3, s_2 is a free state. We first determine the minimum state code bits for F^3 that is 2. (obtained from minimumLengthCode() function in Figure 12). Since s_2 only has the state transition to s_3, we put s_2 in the state bundle of b_1, which has the smallest hamming distance to b_2, which is 1, (where state s_3 is in). It can be noticed that s_2 also can be put in b_3, which has the same hamming distance from b_2. In sub-FSM F^5, s_6 is a free state. It has the state transitions to s_5 and s_0, where the former transition occurs in sub-FSM F^5 and the latter transition is between sub-FSM F^5 and F^1. Assume that the state transition probability between s_6 and s_5 is higher than that between s_6 and s_0, we put s_6 in the bundle b_2 with code “11”, which has only one bit hamming distance from b_3 with code “10”. Bundle b_1 is not chosen for s_6 is because there are two bits hamming distance between b_1 and b_3.

The final state table including merging coupled-state and state encoding procedure is shown in Figure 11b) whereas the initial state bundle table without optimization shown in Figure 11c).

4 Experimental Results

In this section we present results showing how the state bundling and state encoding algorithms, given in section 3, influences the power consumption of partitioned FSMs. We have implemented the algorithms in an automatic synthesis tool that is based on our previous work [12]. Seven of the MCNC [13] standard benchmarks were used in the experiments. The number of states in these benchmarks range from 19 to 118. To determine the state transition probabilities of the FSMs, average input probability and switching probability are inputs to the tool. In our experiments, both are set to 0.5. The power and area figures presented in graphs and tables come from gate-level estimations in Power Compiler and logic synthesis is done by Design Compiler, both these tools from Synopsys [14]. We use a 0.18μm CMOS standard cell library [15] and we assume power supply voltage V_{dd} of 1.8V and a clock frequency of 20 MHz.

The total average power of a monolithic FSM is $P_{tot,mono} = P_{clk} + P_{reg} + P_{ns} + P_{out}$. Where P_{clk} is the clock net power, P_{reg} is the power in the state registers, P_{ns} is the power in the next-state function, and P_{out} is the power in the output function. The total power of the partitioned FSM is $P_{tot,part} = P_{clk} + P_{reg} + P_{ns} + P_{out} + P_{oh}$. Where the P_{oh} is the power-overhead which is the sum of the power dissipated in the global state memory, circuits for idle condition detection, and shut-down circuits.

FSM partitioning is alone an efficient method for achieving power reductions. As shown in Figure 11, significant reductions have been obtained for the mixed synchronous/asynchronous architecture without optimized state encoding.

In the partitioned FSM a significant part of the power is dissipated in the global state memory and the circuits for idle condition detection and shut-down circuits (P_{oh}). This part is not affected by state encoding procedures presented in this paper. To look in detail on how the proposed procedures affect the power consumption, we first consider only power dissipation in the sub-FSMs ($P_{tot,sub-FSM} = P_{tot,part} - P_{oh}$). In Figure 12, it is shown how coupled-state merging and optimized state encoding affects power dissipation in comparison to the basic procedures. Merging of coupled-states has very little to say for the power consumption and for three of the benchmarks (s832, s820, and scf) coupled-state merging has not affect at all. This is what could be expected since the objective here is to minimize the number of state bits and not the power. The state-encoding gives in average a reduction of 13%.

As shown in Figure 11, the sub-FSM power ($P_{tot,sub-FSM}$) is only a portion the total ($P_{tot,part}$) which is in average 40%. For the total power, the reductions are shown in Figure 12 with an average reduction of 6%.

It Table 1 can be seen that the partitioning algorithm result in small sub-FSMs with high duty probability T^i and the large sub-FSMs have low duty period. From In Figure 13 it can be seen that sub-FSMs with large number of bits in the local state memory, the power optimization procedure is efficient but for the ones with few bits only small reductions can be obtained.

5 Conclusions

In this paper we have presented a state encoding algorithm for partitioned FSM composed of coupled sub-FSM with shared state memory. The algorithm takes the properties of partitioned FSMs and the constraints imposed by the implementation architecture in to account. The relation between the coupled sub-FSMs is given by the state bundling. State encoding is carried out sequentially, one sub-FSM at a time where high priority is given to sub-FSMs with high duty time. The power reductions we achieved for the sub-FSMs are promising. The reductions for the partitioned FSMs as a whole are obviously lower since state encoding cannot reduce the power in the asynchronous state memory, idle condition detection logic, and the shut-down logic that are already established before state encoding. This limitation comes from the fact that we first have the partitioning procedure followed by the state-encoding. An algorithm for simultaneous partitioning and state encoding, as the one presented in [**Fel! Hittar inte referenskälla.**], removes this limitation. But the complexity of the problem increases dramatically and so do the run-times for the algorithms. The average power reduction achieved in [**Fel! Hittar inte referenskälla.**] is very close to ours. It is however difficult to compare our results to theirs since there is no information on the statistics given of the input signals to the FSM benchmarks. A direction for future work is to develop an algorithm for simultaneous partitioning and state encoding for the mixed synchronous/asynchronous architecture in order to find out if the more complex algorithms will pay off in reduced power.

6 References

1. L. BENINI, G DE MICHELI: ‘Dynamic power managment : Design techniques and CAD tools’ (Kluwer Academic Publishers, 1998)

2. A. ABDOLLAHI, F. FALLAH, M. PEDRAM: ’ Leakage Current Reduction in CMOS VLSI Circuits by Input Vector Control’, IEEE Tans. On VLSI, 2004, **12**, pp. 140-154

3. M. ALADINA, J. MONTEIRO, S. DEVADAS, A. GOSH : ‘Precomputational-based sequential logic optimization for low power’, *IEEE Trans. on VLSI*, 1994, **2**, pp. 426-436

4. J. MONTEIRO, S. DEVADAS, A. GHOSH : ‘Sequential logic optimization for low power using input-disabling precomputation architectures’, *IEEE Trans. on CAD*, 1998, **17**, pp. 279-284

5. L. BENINI, G. DE MICHELI, A. LIOY, E. MACII, G. ODASSO, M. PONCINO : ‘Synthesis of power-managed components based on computational kernal extraction’, *IEEE Trans. On CAD*, 2001, **20**, pp. 1118-1131

6. S-H. CHOW, Y-C. HO, T. HWANG: ‘Low-power realization of finite-state machines – a decomposition approach’, *ACM Trans. on design automation of electronics systems*, 1996, **1**, pp. 315-340

7. L. BENINI, P. SIEGEL, G. DE MICHELI: ‘Automatic synthesis of low-power gated-clock finite-state machines’, 1996, **6**, *IEEE Trans. on CAD*, pp. 630-643

8. B. OELMANN, K. TAMMEMÄE, M. KRUUS, M. O’NILS: ’Automatic FSM synthesis for low-power mixed synchronous/asyncrhonous implementation’, *Journal of VLSI Design - Special issue on low-power design*, 2001, 12, pp. 167-186

9. B. OELMANN, M. O’NILS: ‘Asynchronous control of low-power gated-clock finite-state machines’, *Proceedings of IEEE International conference on electronics, circuits, and systems*, 1999, pp. 915-918

10. C. CAO, B. OELMANN: ‘Mixed synchronous/asynchronous state memory for low power FSM design’, *Proceedings of the EUROMICRO symposium on digital system design*, 2004, pp. ???

11. C. CAO, M. O’NILS, B. OELMANN: ‘A tool for low-power synthesis of FSMs with mixed synchronous/asynchronous state memory’, 2004, *IEEE Proceedings of the Norchip Conference*, pp. ???

12. B. OELMANN, M. O’NILS: ‘A low power hand-over mechanism for gated-clock FSMs’, *Proceedings of the European conference on circuit theory and design*, 1999, pp. 118-121

13. S. YANG: ‘Logic synthesis of optimization benchmarks – user guide version 3.0’, *MCNC Technical report*

14. Synopsys inc.: ‘http://www.synopsys.com’, company homepage.

15. United Microelectronics Corp: ‘http;//www.umc.com.tw’, company homepage

16. G. VENKATARAMAN, S. M. REDDY, I. POMERANZ: ‘GALLOP: Genetic Algorithm based Low Power FSM Synthesis by Simultaneous Partitioning and State’ Assignment’, The Sixteenth International Conference on VLSI Design, 2003, pp. 533-538

Figure captions:

Figure 1. Structural decomposition of FSM

Figure 2. Example, a) Monolithic FSM with state partition indicated, b) coupled-states introduced

Figure 3. Example, Coupled-state table

Figure 4. State table

Figure 5. Pseudo code for bundling of the coupled and the free states

Figure 6. Example of a partition FSM with high *c*

Figure 7. Optimized coupled-state table

Figure 8. Pseudo code for g-state merging

Figure 9. State encoding in re-ordered state table

Figure 10. Pseudo code for optimized state encoding

Figure 11. Power reductions for partitioned FSMs

Figure 12. Power reductions in the sub-FSMs

Figure 13. Power reductions versus number of state memory bits

Table 1.Structural information from the decompositions

Figures:

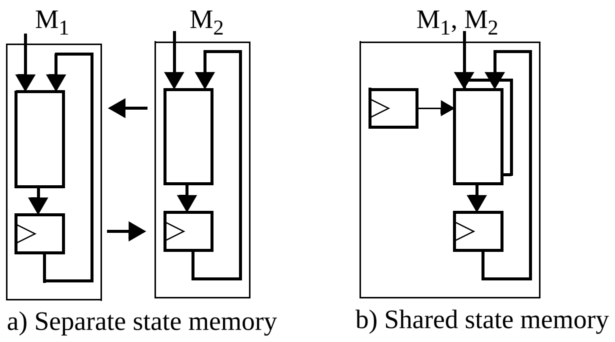

FIGURE 1. Structural decomposition of FSM

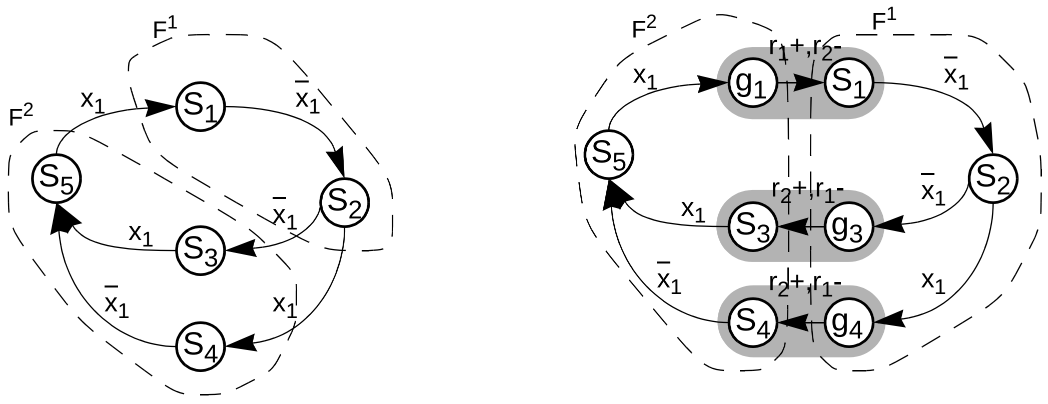

FIGURE 2. Example, a) Monolithic FSM with state partition indicated, b) Coupled states introduced

B:	b_1	b_2	b_3
F^1	g_1	s_3	s_4
F^2	s_1	g_3	g_4

FIGURE 3. Example, Coupled state table

```
struct subFSM {
    set of int S, G, Q;
}

set of struct subFSM F;
int sb[n, $max(\lceil \log_2 |U^m| \rceil)$] ← null;

assignCoupledStates(set of struct subFSM F, int sb)

    int i,j ← 1;
    for all f ∈ F {
        for all q ∈ f.Q {
            i ← indexOf(f);
            sb[i,j] ← q;
            for all ft ∈ F\f {
                for all g ∈ ft.G {                    //g states in other subFSMs
                    if (indexOf(g) = indexOf(q))
                        sb[indexOf(ft),j] ← g;
                }
            }
            j ← j +1;
        }
    }
}
assignFreeStates(set of struct subFSM F, int sb)
{
    for all f ∈ F {
        int j ← 1;
        i ← indexOf(f);
        for all s ∈ f.S \f.Q {
            while (sb[i,j] ≠ null)
                j ← j +1;

            sb[i,j] ← s;
        }
    }
}
```

FIGURE 4. Pseudo code for bundling of the coupled (assignCoupledStates) and free states (assignFreeStates)

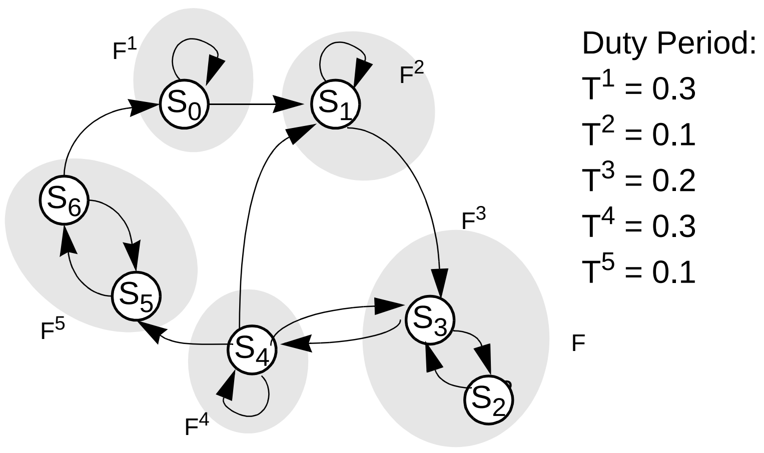

FIGURE 5. Example of a partitioned FSM with high *c*.

a) Initial coupled state table

B:	$\mathbf{b_0}$	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$	$\mathbf{b_4}$
F^1	s_0	g_1	-	-	-
F^2	-	s_1	g_3	-	-
F^3	-	-	s_3	g_4	-
F^4	-	g_1	g_3	s_4	g_5
F^5	g_0	-	-	-	s_5

b) Sorted table

B:	$\mathbf{b_0}$	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$	$\mathbf{b_4}$
F^1	s_0	g_1	-	-	-
F^4	-	g_1	g_3	s_4	g_5
F^3	-	-	s_3	g_4	-
F^2	-	s_1	g_3	-	-
F^5	g_0	-	-	-	s_5

c) After merging coupled-state

B:	$\mathbf{b_0}$	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$	$\mathbf{b_4}$
F^1	s_0	g_1	-	-	-
F^4	s_4	g_1	g_3	g_5	-
F^3	g_4	-	s_3	-	-
F^2	-	s_1	g_3	-	-
F^5	g_0	-	-	s_5	-

d) Final coupled state table

B:	$\mathbf{b_0}$	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$
F^1	s_0	g_1	-	-
F^4	s_4	g_1	g_3	g_5
F^3	g_4	-	s_3	-
F^2	-	s_1	g_3	-
F^5	g_0	-	-	s_5

FIGURE 6. Optimized state table

```
struct subFSM {
    set of int S, G, Q;
}

set of struct subFSM F;
int sb[n, $max(\lceil \log_2 |U^m| \rceil)$];  //state bundle table
double probBundle[numberOf(F.G)]; //sum of static state probability of states in each state bundle
mergeCoupledStates(set of struct subFSM F, int sb, double probBundle)

    sort(sb);
    g_n ← numberOf(F.G);
    for (i ← 1; i < g_n; i ← i+1){
        max_gain ← 0;
        opt_b  ← 0;
        for (j ← i+1; j ≤ g_n; j← j+1){
            row  ← 1;
            while (sb[row, i]=null ||sb[row,j]=null)
                row  ← row+1;
            if (row=n){                    //column i and j can be merged
                gain ← probBundle [i]+probBundle [j];
                if (gain > max_gain){
                    max_gain ← gain;
                    opt_b ← j;
                }
            }
        }
        if (opt_b > 0){                    //find column obt_b can be merged into column i
            for (k ← 1; k ≤  n; k ← k+1){
                if (sb[k, i] = null)
                    sb[k, i] ← sb[k, obt_b];
            }
            “remove column opt_b in sb”;
            g_n ← g_n-1 ;
        }
    }
    sort(sb);
}
```

FIGURE 7. Pseudo code for g-state merging

B: C:	b_1 000	b_2 001	b_3 010	b_4 011	b_5 100	b_6 101	b_7 110	b_8 111	$\lceil \log_2 \lvert U^m \rvert \rceil$
F^1	s_1	g_4	s_2	s_3	-	-	-	-	2
F^2	s_6	s_4	s_5	g_7	-	-	-	-	2
F^3	g_1	-	-	s_7	-	-	-	-	2
F^4	g_1	s_8	g_5	s_9	s_{10}	s_{11}	s_{12}	s_{13}	3

FIGURE 8. State encoding in re-ordered state table

```
int old_sb[n, $max(\lceil \log_2 \lvert U^m \rvert \rceil)$];            //state bundle table before optimization
int new_sb[n, $max(\lceil \log_2 \lvert U^m \rvert \rceil)$]← null;   //state bundle table after optimization
double b_matrix[numberOf(mergedCoupledState),numberOf(mergedCoupledState)];
optimiseCoupledStates(int old_sb, double b_matrix, int new_sb)

    int b[numberOf(mergedCoupledState)];          //state bundles
    struct sub_b;    //subset of state bundles
    for (i ← 1; i ≤ numberOf(mergedCoupledState); i ← i+1)
        b[i] ← the ith column of old_sb;
    lock(b[1]) ;
    for (i ← 1; i ≤ n; i ← i+1)
        new_sb[i, 1] ← b[1];
    for (i ← 1; i ≤ n; i ← i+1){
        sub_b ← ∅;
        for (j ← 1; j ≤ numberOf(mergedCoupledState); j ← j+1){
            if (old_sb[i, j] ≠ null)
                sub_b ← sub_b ∪ b[j];
        }
        b_n ← least state bits needed for sub_b in new_sb;
        for unlocked state bundle b[x]∈ sub_b{
            for each locked state bundle  b[y] in b
                "find b_matrix[$x_i y_i$] with maximal state bundle transition probability";
        }
        for (j ← 1; j ≤  $2^{b\_n}$; j ← j+1)
            "find m is the column index of b[$y_i$] in new_sb,such that
             Hammingdistance(binaryCode(m),binaryCode(j)) is minimal";
        for (k ← 1; k ≤ n; k ← k+1)
            new_sb[k, j] ← b[$x_i$];
        lock(b[$x_i$]);
    }
}
```

FIGURE 9. Pseudo code for optimized coupled state encoding

a) Final coupled state table after optimization

B: **C:**	**b_0 00**	**b_1 01**	**b_3 10**	**b_2 11**
F^1	s_0	g_1	-	-
F^4	s_4	g_1	g_5	g_3
F^3	g_4	-	-	s_3
F^2	-	s_1	-	g_3
F^5	g_0	-	s_5	-

b) Final state table after free state optimization

B: **C:**	**b_0 00**	**b_1 01**	**b_3 10**	**b_2 11**	bits
F^1	s_0	g_1	-	-	1
F^4	s_4	g_1	g_5	g_3	2
F^3	g_4	s_2	-	s_3	2
F^2	-	s_1	-	g_3	2
F^5	g_0		s_5	s_6	2

c) State table before state encoding optimization

B:	**b_0 000**	**b_1 001**	**b_2 010**	**b_3 011**	**b_4 100**	**bits**
F^1	s_0	g_1	-	-	-	1
F^2	-	s_1	g_3	-	-	2
F^3	s_2	-	s_3	g_4	-	2
F^4	-	g_1	g_3	s_4	g_5	3
F^5	g_0	s_6	-	-	s_5	3

FIGURE 10. Comparison of state bundle table before and after optimization

```
struct subFSM {
    set of int S, G, Q;
}

set of struct subFSM F;
int sb[n, $max(\lceil \log_2 |U^m| \rceil)$];  //state bundle table before free states assignment
double s_matrix[numberOf(S),numberOf(S)];  //state transition probability matrix

optimizeFreeStates(set of struct subFSM F, int sb, double s_matrix)
{
    int b_n[n];     //minimun state code length in each subFSM
    int sb_backup[n, $max(\lceil \log_2 |U^m| \rceil)$];
    sb_backup $\leftarrow$ copy(sb);
    assignFreeStates(F,sb_backup);
    for (i $\leftarrow$ 1; i $\le$ n; i $\leftarrow$ i+1)
        b_n[i] $\leftarrow$ minimumLengthCode(sb_backup[i]);
    for all f $\in$ F {
        i $\leftarrow$ indexOf(f);
        A $\leftarrow$ f.Q $\cup$f.G;                //assigned states ,g states included
        D $\leftarrow$ f.S \f.Q;                     //unassigned states
        do{
            count $\leftarrow$ numberOf(D);   //unassigned state number
            if (count>0){
                for all a $\in$ A {
                    for all d $\in$ D
                        "find s_matrix[$a_i$,$d_j$] with highest state transition probability";
                }
                k $\leftarrow$ 1;
                while (sb[i,k] $\neq$ $a_i$)
                    k $\leftarrow$  k+1;
                for (m $\leftarrow$ 1; m $\le$  $2^{b\_n[i]}$; m $\leftarrow$ m+1){
                    if(sb[i, m] $\neq$ null)
                        "find position $m_i$ with minimal Hammingdistance(binaryCode($m_i$ -1), binaryCode(k-1));"
                }
                sb[i, $m_i$]  $\leftarrow$ $d_j$;
                A$\leftarrow$ A $\cup$ $d_j$;
                D$\leftarrow$ D\$d_j$;
                count $\leftarrow$ count-1;
            }
        }while (count>0)
    }
}
```

FIGURE 11. Pseudo code for free state encoding optimization

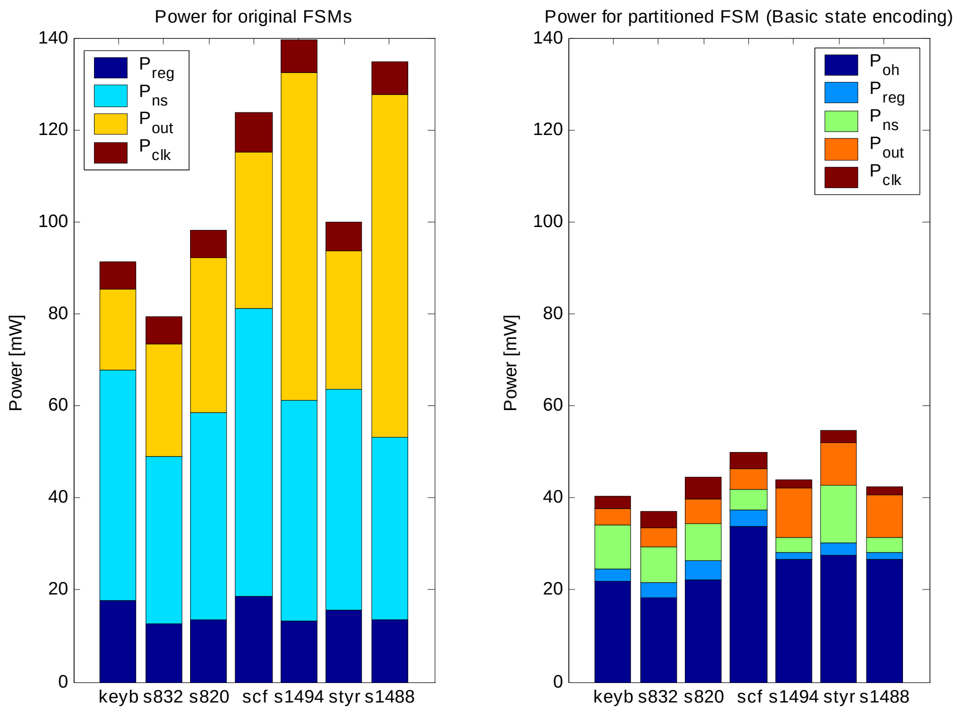

FIGURE 12. Power reductions for partitioned FSMs

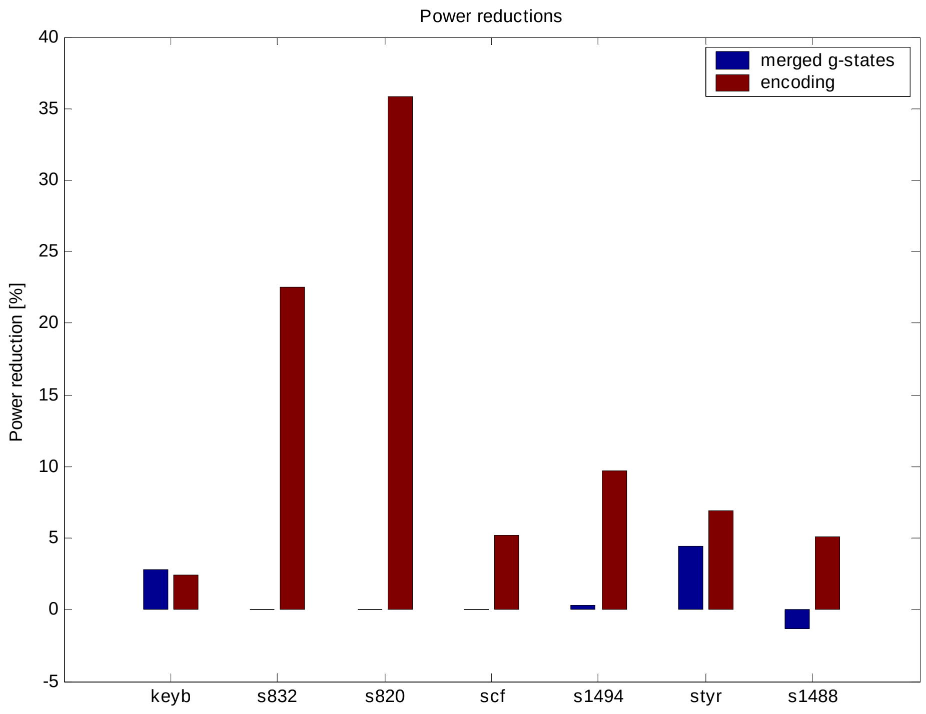

FIGURE 13. Power reductions in the sub-FSMs

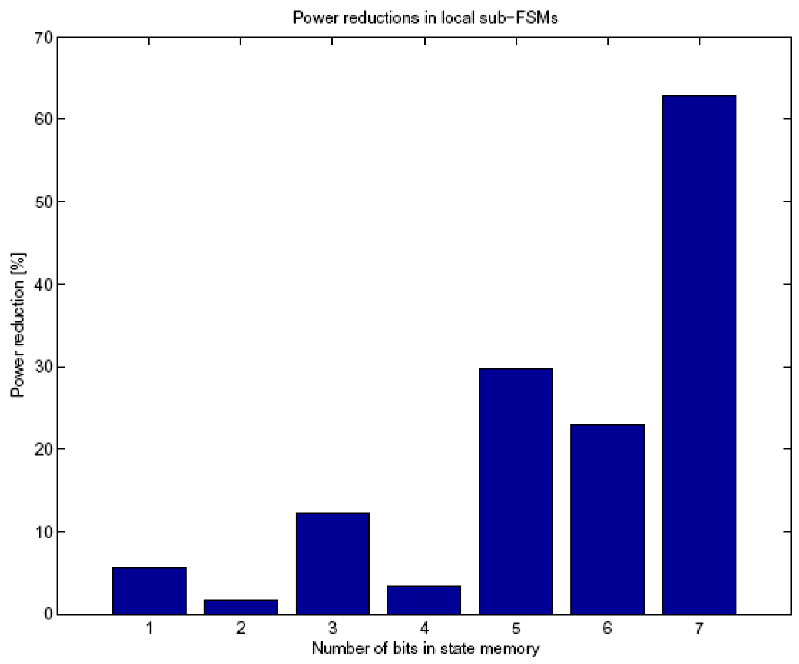

FIGURE 14. Power reductions versus number of bits in the state memory

TABLE 1. Structural information from the FSM decomposition

FSM	keyb	s832	s820	scf	s1494	styr	s1488
$\|S^1\|$	1	4	4	4	1	1	1
$\|U^1\|$	4	5	5	5	2	4	2
$\|PI^1\|$	3	6	6	1	3	5	3
$\|PO^1\|$	1	5	9	12	12	6	13
$\|T^1\|$	0.99	0.99	0.99	0.96	0.91	0.85	0.91
$\|S^2\|$	1	21	4	4	1	1	1
$\|U^2\|$	3	24	7	8	4	4	4
$\|PI^2\|$	6	18	9	3	3	5	3
$\|PO^2\|$	0	17	10	8	7	2	7
$\|T^2\|$	0.27	0.03	0.03	0.08	0.20	030	0.20
$\|S^3\|$	1		17	110	1	2	1
$\|U^3\|$	4		23	8	4	3	4
$\|PI^3\|$	7		17	3	6	6	6
$\|PO^3\|$	1		12	8	13	1	12
$\|T^3\|$	0.18		<0.01	0.02	0.08	0.20	0.08
$\|S^4\|$	1				1	4	1
$\|U^4\|$	4				2	8	3
$\|PI^4\|$	7				0	5	1
$\|PO^4\|$	1				5	5	4
$\|T^4\|$	0.09				0.02	0.08	0.02
$\|S^5\|$	15				1	8	1
$\|U^5\|$	16				3	16	2
$\|PI^5\|$	6				1	7	0
$\|PO^5\|$	2				4	10	3
$\|T^5\|$					0.03	0.03	0.03
$\|S^6\|$					1	14	42
$\|U^6\|$					3	21	46
$\|PI^6\|$					2	6	8
$\|PO^6\|$					7	10	19
$\|T^6\|$					0.02	<0.01	0.02
$\|S^7\|$					42		1
$\|U^7\|$					46		3
$\|PI^7\|$					8		2
$\|PO^7\|$					19		5
$\|T^7\|$					0.02		0.02