Analysis and Design of an On-Chip Retargeting Engine for IEEE 1687 Networks

Ahmed Ibrahim, Hans G. Kerkhoff
Testable Design and Test of Integrated Systems Group (TDT),
Centre of Telematics and Information Technology (CTIT), University of Twente,
Enschede, the Netherlands
a.m.y.ibrahim@utwente.nl and h.g.kerkhoff@utwente.nl

***Abstract*—IEEE 1687 (iJTAG) standard introduces a methodology for accessing the increasing number of embedded instruments found in modern System-on-Chips. *Retargeting* is defined by iJTAG as the procedure of translating instrument-level patterns to system-level scan vectors for a certain network organization. The analysis and the design of an on-chip retargeting engine is presented in this paper. Performing retargeting on-chip enables the execution of life-time dependability procedures using embedded instruments. The proposed engine is capable of retargeting instruments' patterns using an optimized model of the network, which is extracted by resolving the dependencies between the data registers of the instruments and the network-state registers.**

***Keywords*—IEEE 1687, iJTAG, embedded instruments, retargeting, dependability.**

I. Introduction

The increasing System-on-Chip (SoC) complexity enabled by the continuous technology scaling, became a major challenge for testing and debugging. Consequently, an increased number of embedded instruments became integrated in modern SoCs. Access to those instruments was usually done in an ad-hoc manner, however, with the introduction of the IEEE 1687 standard (iJTAG) [1] instrument access-methods became standardized, and more instruments could become connected to the iJTAG instruments network in a systematic manner.

A subset of embedded instruments could be accessed on-chip for maintaining the life-time dependability of SoCs, as well as the off-chip access for test and debug. For example: health monitors, environment sensors, fault detectors, and others, collectively referred to as dependability instruments [2-4]. An on-chip dependability manager could reuse the existing iJTAG network as a standardized, scalable and low cost on-chip access network to the dependability instruments in order to execute dependability operations. For example, frequency downscaling using a phase locked loop instrument, when a certain path delay monitor instrument indicates a persistent delay fault due to aging.

iJTAG uses reconfigurable scan networks in order to achieve an optimum access path to the required instrument register with minimum path overhead. Such networks grow in complexity, and often multiple access vectors are required to be generated in order to configure the network to access a certain instrument. For example, in figure 1, three scan vectors are required in order to access R2, the first is for setting the register SCB1, which will enable accessing both R1 and SCB2 in the next cycle, then the second for setting SCB2 to '1' enabling the access to R2 in the third cycle. The process of translating an instrument-level pattern to several network-level vectors is called the *retargeting* process.

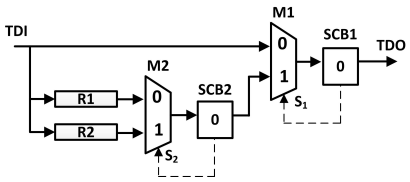


Figure 1: Example of a reconfigurable scan network.

Optimum pattern retargeting was previously investigated in [5] by mapping it into a pseudo-boolean optimization problem, then using iterative SAT solving to find an optimum scan sequence. Such work is useful for retargeting tools, or for generating a set of retargeted patterns corresponding to a certain access procedure. However, for on-chip access via a dependability manager, *dynamic retargeting* is required, as the access procedures will be dependent on the runtime data.

In this paper we present the analysis and the design of an on-chip retargeting engine. The engine uses an optimized network model for retargeting which enables a simplified processing compared to the other models used in software tools. The paper is organized as follows: Terminologies and background are given in section II, then the analysis for the on-chip model is presented in section III. In section IV the architecture of the retargeting engine is presented. Then in section V experimental verification by means of SoC test benchmarks is discussed. Conclusions are given in section VI.

II. Background and Terminologies

iJTAG introduces a reconfigurable scan network connected between the TDI and the TDO ports of the Test Access Port (TAP), for efficiently accessing the Test Data Registers (TDRs) of embedded instruments. A *scan segment* can be defined as any valid organization of scan registers and network multiplexers (i.e. ScanMuxs) connected between two nodes in the network. The scan network consists of several scan segments branching at *branching nodes*, and merging at ScanMuxs.

Network reconfiguration is performed by controlling the *address bits* (S) of the network ScanMuxs. The source of the

978-1-4673-9659-2/16/$31.00 ©2016 IEEE

address bits is a set of updatable network registers referred to as the ScanMux Control Bits (SCBs). A *Segment Insertion Bit* (SIB) is a special configuration of a ScanMux and an SCB for including/excluding segments. For example, M1-SCB1 configuration in figure 1 is considered to be a SIB. SIBs allow a hierarchal organization of the iJTAG networks.

We define the set of *Scan Paths* (SP) to include all possible scan paths between TDI and TDO, each corresponds to a certain state of the network SCBs. An *active path* $\in SP$, is the currently configured scan path. A *scan access* is performed by executing the Capture-Shift-Update (CSU) cycle defined by the IEEE 1149.1 TAP controller. During this cycle, data from the instruments are captured into the corresponding TDRs. Then a new scan vector with the same length of the active path is shifted into the network, while simultaneously data from the active path is shifted out. The scanned-in vector could consist of both network configuration bits and instruments data. Finally the CSU cycle ends by updating both the updatable TDRs and the SCBs with the new instruments patterns and the network configuration bits.

In order to access a certain instrument via its TDR, the network should be configured first to an active path that includes this TDR via one or more scan accesses. We define the set of *Access Paths* (AP) of a certain register to include each scan path $p \in SP$ that includes this register. A network register (TDR or SCB) is considered to be *selected* when the active path is $\in AP$.

The selection of a register is dependent on a minimum set of multiplexers that when configured properly, an active path $\in AP$ is formed. We define the *Selection Dependency* of a register ($Sel(Reg)$) as a boolean relation in terms of a subset of the ScanMuxs address bits, which evaluates to true only when the corresponding path is $\in AP(Reg)$. For example in figure 1, $Sel(R1) = S_1 \wedge \neg S_2$, $Sel(R2) = S_1 \wedge S_2$, $Sel(SCB2) = S_1$, $Sel(SCB2) = True$.

A register could have several disjunctive selection clauses. For example, in figure 2, $Sel(R2) = (S_1 \wedge \neg S_2 \wedge S_3) \vee (S_1 \wedge S_2)$. Which means that accessing R2 could be done by configuring the network to satisfy either clause (i.e. setting $S_1S_2S_3$ to '101' or to '11X'). Therefore, in general $Sel(Reg)$ is given in a Disjunctive Normal Form (DNF), where each clause represents a different minimum network configuration possibility for forming an access path, and with the literals are multiplexers address bits. We define a *multi path register* as any register with more than one clause in its selection dependency. A register is said to be *inaccessible* when substituting the address bits with their corresponding SCB sources, its selection equation is reduced to False.

III. An Optimized Model for On-chip Retargeting

Executing retargeting on-chip requires maintaining a model of the iJTAG network. One possible model is similar to what was introduced in [6] or in [7] where the network is represented as a directed graph. In this model, each node in the graph corresponds to a certain component (TDR, SCB or ScanMuxs), and maintains a set of predecessor ($Pred$) and a set of successor ($Succ$) nodes. As this model only holds the connectivity information, generating the retargeted patterns on-chip for a certain register-access requires resolving the register selection dependencies each time it is being accessed, which is an expensive tasks in terms of the required clock cycles.

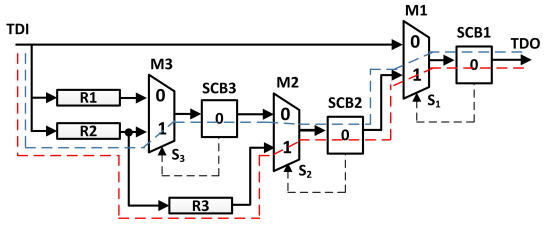


Figure 2: A register (R2) with two disjunctive selection dependencies.

In this section we will present the analysis for an optimized iJTAG network model for on-chip retargeting. The model will embed both the selection dependencies and the network connectivity, which will reduce the required time for generating the retargeted patterns, and also reduce the area required to store the model on-chip, as it will eliminate the connectivity references ($Pred$ and $Succ$) by embedding it in the sequential organization of the model.

A. The Selection Dependency Graph

In order to resolve the selection dependencies of the registers on the address bits of the different ScanMuxs, we define the Selection Dependency Graph (SDG) as a directed graph $G = (V, E)$, where an edge $e = (v_{src}, v_{dst}) \in E$ represents a certain selection dependency of the destination node on the source one, and its direction is the opposite of the direction of the scan path between v_{src} and v_{dst}. The set of nodes (V) is partitioned into four main node types: 1) V_{Reg} represents a network register (TDR or SCB), 2) V_{I0} represents a '0' input port of a 2-to-1 ScanMux, 3) V_{I1} represents a '1' input port of a 2-to-1 ScanMux, 4) V_{Br} represents a branching node in the scan path. Two auxiliary nodes for TDI and TDO are also included. A node $v \in V_{Reg}, V_{I0}$ or V_{I1} has only one predecessor node $v_{pr} \in V$ in the SDG, while a branching node $v \in V_{Br}$ has multiple predecessor nodes.

In this analysis we represent an m-to-1 ScanMux with n bit address (S_{n-1} ... S_0) using the equivalent n-stage cascaded organization of 2-to-1 ScanMuxs, such that each of the 2-to-1 ScanMuxs would have a unique address bit literal in the selection relation. Figure 3 shows the equivalence of a 4-to-1 ScanMux.

Constructing the SDG is done by parsing a network representation written in RTL or in Instrument Connectivity Language (ICL) that is defined by iJTAG. The graph is constructed starting from TDO and tracing the network backward towards TDI, while replacing the registers, branching nodes and each input of a ScanMux ($I0$ & $I1$) with corresponding nodes. For the sake of simplicity we assume that SCBs are always separate registers. Figure 4 shows the corresponding SDG to the network in figure 2.

The selection of a node $v \in V$ is computed from the selection of its predecessors according to its type. A similar

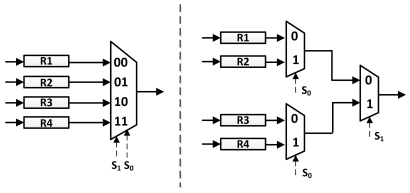


Figure 3: Example of 2-to-1 Mux equivalence.

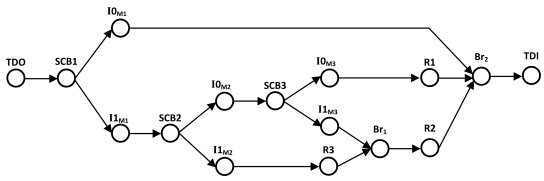


Figure 4: The selection dependency graph corresponding to figure 2.

analysis was introduced in [5] for the SAT predicates. The selection is deduced from SDG here as follows:

$$Sel(V_{\mathrm{TDO}}) = Sel(V_{\mathrm{TDI}}) = TRUE \tag{1}$$

$$Sel(V_{Reg}) = Sel(v_{pr}) \tag{2}$$

$$Sel(V_{I0}) = Sel(v_{pr}) \wedge \neg S \tag{3}$$

$$Sel(V_{I1}) = Sel(v_{pr}) \wedge S \tag{4}$$

Eq. (1) implies that both auxiliary nodes (TDI and TDO) are always selected. This is used as an initial condition for the selection dependency resolving algorithm. Eq. (2) indicates that a register is selected when its predecessor node in SDG is selected. While eq. (3) indicates that a '0' input port node is selected when its predecessor is selected and the ScanMux address (S) = '0'. Similarly in eq. (4) for the '1' input port node except that S should be = '1'.

A branching node is selected (i.e. accessed in the active path) when at least one of its predecessor nodes is selected. Eq. (5) represents the selection equation of a branching node. A reduction by factoring is applied to the resulting disjunctive clauses in order to remove the dependency on the ScanMuxs that had their multiplexed segments originated from this branching node.

$$Sel(V_{Br}) = Fact(\bigvee_{v_{pr} \in pred(V_{Br})} Sel(v_{pr})) \tag{5}$$

B. Resolving the Selection Dependencies

After constructing the SDG from the network representation, equations (1-5) could be used to calculate the selection of all the registers by traversing the graph from TDO to TDI and resolving the selection of each node based on its predecessor's. In order to embed the connectivity information in the on-chip model of the network, we propose a sequential tracing algorithm that assigns orders to the registers representing the register position in the on-chip model while calculating the nodes selections.

First we define the selection of a node $v \in V$ to be a DNF relation with n clauses, where each clause represents a minimum network configuration for accessing the node.

$$Sel(v) = \bigvee_{i \in 1 \ldots n} C_i(v) \tag{6}$$

A clause ($C_j(v)$) of certain node ($v \in V$) has one source clause in a predecessor node $v_{pr} \in pred(v)$, from where $C_j(v)$ was originally derived. The source of $C_j(v_{I0})$ ($v_{I0} \in V_{I0}$) could be resolved by substituting equation (6) for $v = v_{pr}$ ($v_{pr} = pred(v_{I0})$) in equation (3), the resulting selection relation $Sel(v_{I0})$ will be given as ($\bigvee_{i \in 1 \ldots n}(C_i(v_{pr}) \wedge \neg S_{pr})$). It is clear that the j^{th} clause in $Sel(v_{I0})$ (i.e. $C_j(v)$) was derived from the j^{th} clause in $Sel(v_{pr})$ (i.e. $C_j(v_{pr})$), therefore:

$$Src(C_j(v_{I0})) = C_j(v_{pr(v_{I0})}) \tag{7}$$

The same is true for a clause in $Sel(v_{I1})$ and $Sel(v_{Reg})$.

In order to resolve the sources of the clauses in $Sel(v_{Br})$, a traversing algorithm is developed in Algorithm 1, which returns a set of source clauses for all the clauses in SDG ($Src(C_{i=1..n}(v_{\in V}))$). The algorithm traverses the graph starting from TDO towards TDI by calling $Traverse(G, v_{\mathrm{TDO}})$. It proceeds by always choosing the $I1$ nodes (line 6). When a branching node is encountered and not all of its predecessors selections were resolved, the algorithm backtracks the same path until the first I1 node, then traverse forward from the corresponding I0 node (line 17). Every time a branching node is accessed the selection relation is iteratively updated with the predecessor set of clauses as follows:

$$Sel(v_{Br}) = Fact(Sel(v_{Br}) \vee Sel(v_{pr})) \tag{8}$$

If two clauses were reduced by factoring while calculating (8), the source of the resulting clause (C_{reduced}) is assigned to the clause in v_{pr}. The algorithm ends by reaching v_{TDI}.

We next order the clauses in the registers selection relations. Algorithm 2 performs the ordering by traversing the graph starting from TDO by calling $Order(G, v_{\mathrm{TDO}}, C_1(TDO))$. Algorithm 2 traverse forward by selecting V_{I1} when the SDG branches. While traversing the graph, one clause is selected starting from $C_{\mathrm{current}} = C_1(\mathrm{TDO})$ (i.e. $TRUE$) and selecting the clause in the successor node where C_{current} is its source. When a branching node $v \in V_{Br}$ is reached, the algorithm only proceeds forward if $\mathrm{C_{current}}$ is a source of a clause in Sel(v), if not (i.e. C_{current} was reduced) the algorithm backtracks like in Algorithm 1. During this sequence, while visiting a register node, an order is assigned to C_{current} corresponding to the order of visiting it w.r.t all other registers clauses. A multi path register with n number of clauses will be visited n times during the traversal, where each clause will be assigned with a different order. Figure 5(a) shows the selection relations for the network in figure 2 along with the clauses ordering resulted from Algorithm 2.

C. Generating the HArray

The resulting set of $Registers\ Clauses$ (RC) is used to construct the on-chip network model. A multi-path register

Algorithm 1 Setting the Source of Clauses in SDG

```
1: procedure TRAVERSE(G, v)
2:     Calc. Sel(v) from (2), (3) or (4), (∀v ∈ {V_Reg, V_I0,V_I1})
3:     Set Src(C_{i∈1...n}(v)) as in (7), (∀v ∈ {V_Reg, V_I0,V_I1})
4:     Add v to trace_path
5:     if succ(v) = {v_I0, v_I1} then                    ▷A Mux
6:         Traverse(G, v_I1)                  ▷Choose v_I1 if G branches
7:     else if succ(v) = v_Reg then
8:         Traverse(G, v_Reg)
9:     else if succ(v) = v_Br then
10:        Calc. Sel(v_Br) from (8)
11:        if a reduction is possible then
12:            Src(C_reduced(v_Br)) ← C_src(v)
13:        else
14:            Src(C_new(v_Br)) ← C_src(v)
15:        if any edge ∈ (pred(v_Br), v_Br) is not visited then
16:            v_I0 ← Backtrack(G, v)
17:            Traverse(G, v_I0)
18:        else
19:            Traverse(G, v_Br)
20:    else if succ(v) = v_TDI then
21:        return Src
22: procedure BACKTRACK(G, v)
23:    remove v from trace_path
24:    if trace_path.end_node = v_I1 then
25:        return v_I0
26:    else
27:        Backtrack(G, trace_path.end_node)
```

with n disjunctive selection clauses is modeled via n instances in the model, each represents a register access via the access path enabled by satisfying the corresponding clause. Therefore an access to a multi-path register could be performed by accessing any of its instances. The decision of which instance to choose for the access is left for the scheduling algorithm.

We define the on-chip model to be an array $H[k]$ of k records named as the Hierarchy Array ($HArray$). Each record maintains a type field $Type$ and one or more auxiliary fields Aux according to the type. The type field could have a $value \in \{TDR, SIB, I0, I1, SCB\}$. The auxiliary fields are: Aux(TDR) = TDR length (len), Aux(SIB, I0 and I1) = relative pointer to the corresponding SCB (SCB_ptr) and number of nodes (NN). Aux(SCB) = pointer to the corresponding location in the SCB State Vector (SV_ptr), the state vector will be discussed in the next section.

A register selection clause ($C(v_{Reg})$) is a conjunction of m literals, eq. (9). A literal represents a certain address value of a multiplexer from the set (k) of m multiplexers on which this clause is dependent.

$$C(v_{Reg}) = \bigwedge_{i \in 1...m} (\neg)^{x_i} S_{k(i)}, \quad x \in \{0, 1\} \tag{9}$$

The HArray consists of several nested $dependency\ sections$, where registers located in a section are dependent on a corresponding ScanMux. Those sections are indicated by a header element representing a SIB, $I0$ or $I1$, and the depth is indicated by the NN auxiliary field. The traversing algorithm ensures that an $I0$ section is immediately followed by the $I1$ section of the ScanMux. A SIB is a special case of an $I0$-$I1$ pair with an empty $I0$ section.

Algorithm 2 Ordering the registers clauses

```
1: procedure ORDER(G, v, C_current)
2:     Add v to trace_path
3:     find C_i(v) such that Src(C_i(v)) = C_current
4:     C_current ← C_i(v).
5:     if v ∈ V_Reg then
6:         C_current.order ← order; order++
7:     if no Clause found then    ▷Occurs only when v ∈ V_Br
       and C_current is reducible
8:         v_next ← Backtrack(G, v)
9:     else
10:        if succ(v) = {v_I0, v_I1} then
11:            v_next ← v_I1              ▷Choose v_I1 if G branches
12:        else
13:            v_next ← succ(v)
14:    Order(G, v_next, C_current)
```

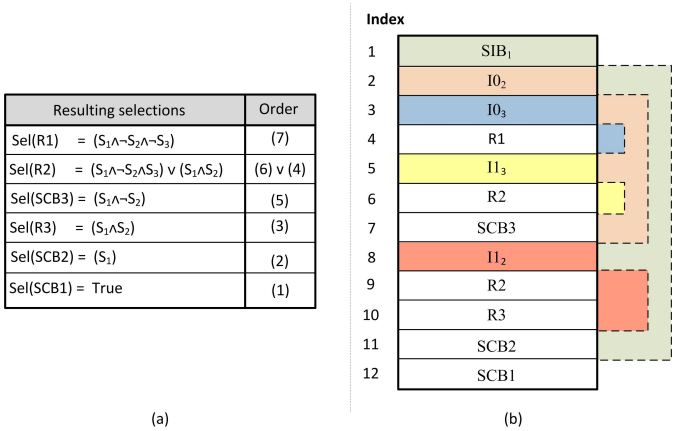
A literal $\neg S_{k(i)}$ in $C(v_{Reg})$ represents an $I0_j$ dependency section, while $S_{k(i)}$ represents an $I1_j$ or a SIB one.

The construction of the HArray is performed as follows: Let $HSel$ be an initially empty FIFO buffer that holds the intermediate values of the dependencies. For example if HSel =[$I0_1$,$I1_2$] it means that the current element in HArray is selected by ($\neg S_1 \wedge S_2$). We choose the clause with the highest order ($C_h \in RC$), then we compare each of the elements in HSel with the literals in C_h of the same order starting from the left most one. If an element in $HSel$ doesn't represent the dependency of the corresponding literal in C_h, then all the sections represented by the remaining elements in HSel (including the conflicting one) are considered to be closed at this location, then we add the new dependency sections represented by the remaining literals in C_h by adding an $I0$, $I1$ or SIB elements to the HArray. A SIB is added when a new S literal is found, while an $I1$ is added when an S literal is found where the corresponding HSel element is the $I0$ of the multiplexer represented by the literal. An $I0$ is added for a new $\neg S$ literal. The register instance is finally included.

After the dependency sections in HArray become defined along with the TDRs/SCBs locations, the auxiliary information (NN and SCB_ptr) are added to the corresponding nodes. Figure 5(b) shows the corresponding HArray to the clauses in figure 5(a) and their ordering. The dependency sections are shown to the right. Note that the multi-path register (R2) has two instances in locations 6 and 9 each in a different section.

IV. Architecture of on-chip Retargeting Engine

The proposed retargeting engine architecture is shown in figure 6. The engine receives a request for concurrently accessing a number of registers from an external instructions scheduler. A register could be accessed for a read or a write operation. The current concurrent access group is stored in a buffer that holds the IDs of the registers (which corresponds to the register location in the HArray), a read/write flag, the value for a write access, and finally a valid bit which indicates that the corresponding register was not yet accessed. The buffer is ordered w.r.t the IDs. The engine maintains the dynamic state of the network configuration by including a State Vector (SV) which holds the current values of the SCBs. The SV_ptr



Resulting selections	Order
Sel(R1) = ($S_1 \wedge \neg S_2 \wedge \neg S_3$)	(7)
Sel(R2) = ($S_1 \wedge \neg S_2 \wedge S_3$) v ($S_1 \wedge S_2$)	(6) v (4)
Sel(SCB3) = ($S_1 \wedge \neg S_2$)	(5)
Sel(R3) = ($S_1 \wedge S_2$)	(3)
Sel(SCB2) = (S_1)	(2)
Sel(SCB1) = True	(1)

(a) (b)

Figure 5: The registers selection (a) corresponding to figure 2 and the resulting HArray (b).

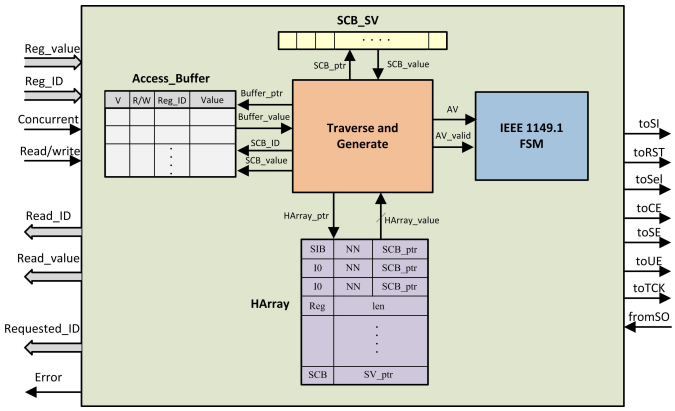


Figure 6: Architecture of the Retargeting Engine

auxiliary of the SCB entries in the HArray points to the corresponding location in SV.

The engine implements a $Traverse\ and\ Generate$ process, this process is responsible for the generation of the set of variable length Access Vectors (AV) that will be serially loaded to the network in order to access the required registers. An embedded IEEE 1149.1 TAP controller FSM is implemented to generate the physical iJTAG signals from the AVs according to the standard.

The $Traversal$ process traverses the HArray sequentially in the same order of the active path. This is performed as follows: While visiting a SIB, $I0$ or $I1$ entry, the corresponding SCB state is fetched from SV, where the SCB address is computed from the auxiliary data (i.e SCB_ptr), then the state bit location is known from the SCB auxiliary (i.e SV_ptr). A SIB is considered to be closed when the corresponding SCB = 0, while the $I1/I0$ port is not selected when the corresponding SCB = 0/1 respectively. If the SIB, $I0$ or $I1$ was found to be closed or not selected, the corresponding dependency section is skipped during the traversal by updating the $HArray\_ptr$. Otherwise, the next entry in the current traversal sequence will be the following entry to the SIB, $I0$ or $I1$. During the traversal, a pointer ($Buffer\_ptr$) is maintained to point to the location of the first entry in the access buffer with a register ID bigger than the $HArray\_ptr$ and with a $valid$ bit = 1.

The $Generate$ process is performed in parallel with the HArray traversal. For each traversed SIB, $I0$ or $I1$ entry in HArray, if the current register ID value indicated by the $Buffer\_ptr$ lies inside the dependency section indicated by the entry, a write access entry is added to the Access Buffer to write the corresponding selection value for the entry (i.e. '1' for $SIBs$ and $I1$, '0' for $I0$), then the buffer is reordered. If the register lies after the dependency section, the corresponding SIB, $I0$ or $I1$ is closed/deselected by writing the corresponding closing bit in the access buffer.

While visiting an SCB, if a write access was found for this SCB in the current buffer entry, a bit is generated in the AV with the corresponding write bit indicated in the buffer, otherwise the old SCB value indicated in the SV is rewritten in AV. The updated SCB value is written back in the SV. Visiting a register during the traversal means that the network at this point is configured to an access path of this register, and a corresponding write or read access should be provided by the instructions scheduler in the access buffer. If not, the retargeting engine informs the instructions scheduler that an undefined access to a register is met and a corresponding access information is required. A write to the register is performed by serially shifting the register value indicated in the access buffer to the AV. While a read is performed by observing the position of the register w.r.t the remaining of the active path during the HArray traversal, then the content of the register is extracted from the corresponding location in the buffered shifted-out network state. The valid bit of an accessed register or SCB is then set to 0.

V. Experimental Results

In this section we perform several experiments in order to verify the correctness of the HArray generation process, and the performance of the proposed retargeting engine. To our knowledge, there is no benchmark circuits for reconfigurable scan networks, therefore we follow the same approach in the verification of both [5] and [9] by using the ITC'02 SoC test benchmark circuits [8]. We consider the same iJTAG hierarchal network organization as proposed in [9]. Instead of using a single chain to act as the core chain, we consider each core to be wrapped using an IEEE 1500 Core-Test Wrapper (CTW) with a minimum required instructions as in figure 7(a). The CTW is used in order to introduce more complex access sequence than in accessing a single connected register. The corresponding sub network for the CTW is shown in figure 7(b) and the section of the HArray for one CTW was computed and is shown in 7(c). Due to space limitations, we only include results of the four circuits with the highest number of cores.

A Matlab implementation of the HArray generation process described in section III was done. The dependency graph representing each SoC in the benchmark set was generated using the corresponding graphs of the CTW and SIBs. A VHDL implementation of the retargeting engine was also developed. The resulting network model from the Matlab processing was used to implement the HArray of each test circuit. The third

Table I: Results for a complete SoC test schedule using the retargeting engine

Circuit name	no. of cores	HArray Size	SV length (bits)	No. of retargeted patterns	Test time (CC)	TP_CC	NC_CC	Retargeting Overhead (%)
p34392	19	272x12	79	66423	17500432	16372887	1127545	6.89
p22810	28	396x11	114	25157	8295510	7870218	425292	5.4
t512505	31	434x18	124	10634	165502627	165324413	178214	0.1
p93791	32	424x11	136	23163	31052141	30661291	390850	1.27

column in table 1 shows the sizes of the generated HArrays. The HArray depth depends on the number of cores (each core is allocated 12 entries) and the network SIBs organization. While the entry size depends on the pointers sizes and the TDR lengths. The SV length corresponds to the number of SCBs in the network.

We start by verifying the correctness of the HArray representation. The retargeting engine was set to execute one register access at a time for all registers in the p34392 test circuit, where a VHDL testbench acts as the external scheduler. Similar to [5], assertion-based VHDL simulations were performed to ensure each register access by checking the scanning of the continuity test pattern (11001100..) for each register (for the WBY and WIR, we observe a '1' and '11' respectively). Observing a successful register access implies the correctness of the dependency representation in the HArray for this register. For example, three different nested dependencies are required to be resolved for a correct retargeting to SC (figure 7(c)).

Next we implement a complete test procedure for each test circuit. In this procedure each core is accessed individually. Each core-test includes an n number of core-accesses, where n is the number of ATPG patterns assigned to this core. In each care-access WBR_in, SC and WBR_out are concurrently accessed. In this experiment we assume an on-chip Test Patterns Generator to provide the write values. In order to measure the retargeting efficiency, we define the Retargeting Overhead (RO), as the ratio between the clock cycles required for the generation of the network configuration bits (NC_CC) to the clock cycles required for the generation of the cores test patterns (TP_CC). The TP_CC can be calculated as $(\sum_{all\ cores}(WBR_{in_i} + SC_i + WBR_{out_i})$ x $nPatterns_i)$. NC_CC can be calculated by observing the number of CC for the generation of the complete test data in the RTL simulation (which includes retargeting CC + actual patterns) then subtracting this value from TP_CC. Therefore $RO = (Observed\ CC - TP\_CC)/TP\_CC$.

As there was no prior work on on-chip retargeting, we couldn't compare our results with other works. The maximum RO was around 7%. It can be shown from the results that the retargeting overhead for the third circuit is very low, this is due to the very long scan chains that this circuit uses compared to the other circuits, therefore the retargeting engine spends most of the time in the register entry of the HArray. The retargeting engine was synthesized using TSMC 90 nm technology, the resulting area constituted to a 57748 um^2.

VI. Conclusions

In this work we presented the analysis and the design of an on-chip engine for dynamic retargeting. The engine is used for executing life-time dependability operations using iJTAG-connected instruments. We analyzed the iJTAG networks for the extraction of an on-chip model, by resolving the dependencies between the data registers and the network state registers. The proposed model enables a simple implementation of a retargeting engine, which retargets instruments patterns by traversing the model and iteratively opening dependency sections to access the target register. The retargeting engine along with the model were verified for a set of benchmark circuits, and it was shown that the retargeting overhead constitutes a maximum 7% of the actual required time for a SoC test.

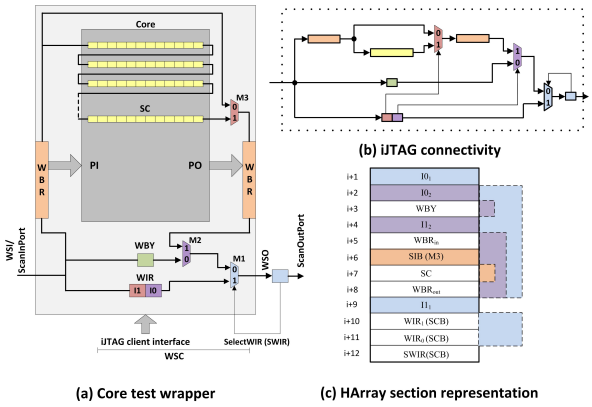


Figure 7: IEEE 1500 CTW

VII. Acknowledgments

This research was carried out within the FP7 BASTION project, financed by the European Committee (EC) and the Netherlands Enterprise Agency (RVO).

References

[1] IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device, IEEE Std 1687-2014, 2014.

[2] J.C. Vazquez, V. Champac, A.M. Ziesemer Jr., R. Reis, I.C. Teixeira, M.B. Santos and J.P. Teixeira, "Built-in aging monitoring for safety-critical applications", Int'l On-Line Testing Symposium (IOLTS) , pp.9-14, 2009.

[3] E. Karl, P. Singh, D. Blaauw, and D. Sylvester, "Compact In-Situ Sensors for Monitoring Negative-Bias-Temperature-Instability Effect and Oxide Degradation", Int'l Solid-State Circuits Conf. (ISSCC), pp.410-412, 2008.

[4] A. Ibrahim and H.G. Kerkhoff, "iJTAG integration of complex digital embedded instruments", Int'l Design & Test Symposium (IDT), pp.18-23, 2014.

[5] R. Baranowski, M. A. Kochte, and H.-J. Wunderlich, "Scan pattern retargeting and merging with reduced access time," European Test Symposium (ETS), pp. 17, 2013.

[6] R. Baranowski, M. A. Kochte, and H.-J. Wunderlich, "Modeling, Verification and Pattern Generation for Reconfigurable Scan Networks," Intl Test Conf. (ITC), 2012.

[7] F. G. Zadegan et al., "Design, Verification and Application of IEEE 1687," Asian Test Symposium (ATS), pp.93-100, 2014.

[8] E. Marinissen, V. Iyengar, and K. Chakrabarty, "A Set of Benchmarks for Modular Testing of SOCs," Intl Test Conf. (ITC), pp. 519528, 2002.

[9] F. G. Zadegan et al., "Access Time Analysis for IEEE P1687," IEEE Trans. Computers, vol. 61, no. 10, pp. 14591472, 2012.