A Microprogrammed Approach for Implementing Statecharts

Javier Cereijo García
European Spallation Source
Lund, Sweden
Email: javier.cereijogarcia@esss.se

Roberto R. Osorio
Department of Computer Engineering, CITIC
University of A Coruña, Spain
Email:roberto.osorio@udc.es

Abstract—**Statechart diagrams allow specifying complex systems in which there may be several states active at the same time and a large number of events and transitions to evaluate. Statecharts have been found useful in the design and implementation of control systems in research facilities, such as particle accelerators. Automatic tools may convert statechart-based specifications into hardware descriptions. During the development of one of those tools, the convenience of implementing statecharts as microprogrammed control systems was considered. In this work, we propose a method for implementing generic microprogrammed architectures that support statecharts upgradable on the field. This approach is evaluated showing its advantages and disadvantages.**

I. Introduction

Control systems in critical applications require fast and jitter-free synchronization in order to trigger a large number of actuators at the right time in a variety of situations. The required control signals are better produced and transmitted by dedicated circuits, instead of programmable processors, in order to prevent undesired delays.

Statecharts are a design tool that extends traditional finite state machines. They have been found to excel in designing complex control systems with concurrent decision making. Hardware implementation of statecharts by hand, however, could be time consuming and error prone. Therefore, automatic tools have been proposed in the past and in recent works.

Those tool produce bespoke circuits that implement most of the functions described in the statecharts. Whereas those tools have some limitations that will be briefly described in Section II, they produce HDL (hardware description language) code that may be synthesized onto ASICs or FPGAs.

In the developing of one of those tools, the convenience of implementing a generic statechart architecture that could be upgraded through firmware came out. Such an architecture would incur in a significant overhead, but it would also bear some advantages, not only over ASICs, but also FPGAs:

- configuration changes could be deployed in a short time skipping logic synthesis
- updating the firmware does not depend on the version of the synthesis software, making the control system easier to maintain. Ever-changing software versions, expiring licenses, and devices that are no longer supported, are serious concerns when planning a infrastructure that must last for decades.

Hence, we propose a method to create microprogrammed architectures [1] [2] that implement statecharts with a high degree of upgradability. Microprogramming was widely used inside microprocessors some decades ago. At that time, computer aided design was not developed, and design errors were quite common. Those errors were often hidden in corner cases and would only show up after the computer was put into the market. Microprogramming allowed vendors to issue control updates and correct those mistakes even in-field. Statecharts may be implemented by mapping concurrent processes (called super-states) into a microprogram. As for microprocessors, the main advantage consists of being able to update the control by just loading a new configuration.

In this work we describe statecharts' characteristics; propose a methodology to convert them into a microprogram; describe de hardware components of the architecture; carry out an evaluation based on a complex example; and present the final conclusions. [1]

II. Statecharts

Statecharts were introduced by Harel [3] in 1987 as a tool to overcome the limitations of Finite State Machines. (FSM) in describing the behaviour of complex systems [4]. FSMs are state-based models where only one state is active at any given time, which can be changed by external inputs or internal conditions. The change between states is called transition.

The aspect that limits the usability of FSMs is that they can greatly grow in complexity when adding states. Statecharts deal with this issue by extending the conventional state-transition diagrams allowing for hierarchy and nested states, concurrency, and better communication among the states. This allows for more compact, expressive and modular diagrams, that can describe complex behaviour with smaller diagrams when compared to FSMs. As such, Statecharts are a visual formalism for describing states and transitions. At the same time Statecharts maintain all of the characteristics of FSMs, such as conditions, outputs, etc. Their main contributions are:

[1]This work was funded in part by the Ministry of Economy and Competitiveness of Spain, Project TIN2016-75845-P (AEI/FEDER, UE), Xunta de Galicia and FEDER funds of the EU under the Consolidation Program of Competitive Reference Groups (ED431C 2017/04), and under the Centro Singular de Investigación de Galicia accreditation 2016-2019 (ED431G/01).

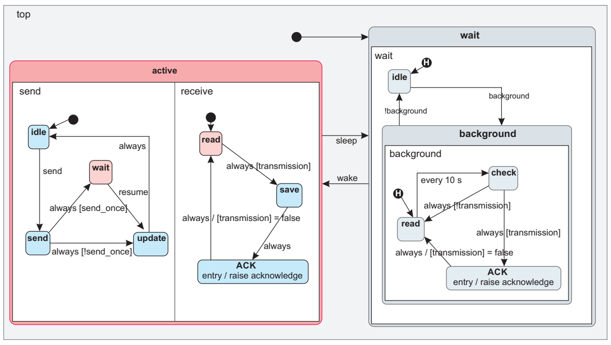


Fig. 1. Simple Statechart with superstates, actions and conditions

- Orthogonality: as opposed to classical FSMs, where only one state can be active at a time, Statecharts can have more than one state active concurrently. These are called AND-states, while the traditional approach are called OR-states. Orthogonality is very useful for describing subsystems.
- Depth: there is a hierarchy in the state structure, allowing for states or even complete FSMs or sub-Statecharts to live inside other states, connected with inter-level transitions. In the nested structure the state containing other states is called super-state. Depth allows for great modularity, clustering, and ease of movement between levels of abstraction by zooming in or out. It is also possible to define entry and default states, and have history in the states, as explained in Section III-B.
- A super-state has History if it can remember its present state and return to it later after being disable for a while. A super-state without History, however, will always resume its activity starting at its initial state.

In Figure 1, a Statechart is shown. At the *top* level, there are 2 *OR* super-states, because either *active* OR *wait* super-states may be running at a given time. The *active* super-state is made up of 2 super-states, *send* and *receive*. This illustrates the concept of hierarchy, as one super-state may be made up of several ones. In this case, both super-states are running in parallel, allowing to describe concurrent processing. This is called an *AND* state (denoted by the divider line). Contrarily, when *wait* is active, either *idle* or *background* are running, but not both at the same time. A black dot and an arrow point at the initial node for each super-state. Moreover, 2 super-states are denoted to have *history* (an H within the dot). Therefore when processing returns to *wait*, it remembers whether it was running in *idle* or *background* and, in the latter case, in which of the 3 nodes.

The complexity of an statechart can greatly grow, but its functionality still be understood with little effort. In the original paper by Harel, a digital watch example is proposed, which we recreate in Figure 2, and will use in the remaining of the paper as it is complex enough to illustrate all the implementation aspects. On it, the *main* superstate is managed concurrently with the *status*, *light* and *power* superstates, forming an AND state. The hierarchy is quite deep, mainly as OR superstates, but new AND states are used in *regular-beep-test* and *stopwatch*. History is also used in several superstates, some of then within other superstates that also use history themselves.

III. Mapping a statechart into a microprogram

A number of papers have been published on statechart synthesis [5] [6] [7]. To the best of our knowledge, however, microprogrammed implementations have never been proposed. In this section, a number of implementation challenges are addressed.

A. *Supporting concurrency and hierarchycal structure*

The microprogrammed architecture must implement a number of processing elements able to run AND states concurrently and communicating among them. This will be addressed by implementing several independent micro-programs running, each of them, one AND state. How those micro-programs run on micro-memories is addressed in detail in Section IV-A. In Figure 2, 5 AND states are contained into *main*, but there are additional ones within *display*.

Also, the hierarchical structure must be mapped onto the microprogram. Moving the execution from one state to another, or between OR states, is similar to jumping to another process in any sequential piece of software. However, jumping to an AND state creates the problem of spawning another process. Hence, some computing resources should be idle waiting to be activated from a different superstate.

The straightforward way to implement this possibility implies that micro-instructions must be able to transition to a new one in a different superstate and, at the same time, send a message to another micro-memory triggering the execution of a given micro-instruction. However, that made the micro-instruction format too verbose, so the following solution was preferred instead. Inner AND superstates are supported by one or more ghost micro-program that mimic the superstate transitions of main micro-programs doing nothing until one of those transitions falls into an AND-superstate. Despite this scheme seems to waste resources, ghost micro-programs often contain significant parts of useful code.

Looking at the left side in Figure 2, let's consider *regular*, *beep-test* and, within *stopwatch*, *display* and *run*. The whole *displays* superstate could be partitioned into 2 microprograms as shown in Figure 3. Each microprogram starts at *time* and *beep-test* respectively. If button *a* is pressed, both microprograms jump to *out*, only that the micro-instructions at the main microprogram are fully functional, while those at the ghost one just apply state transitions without updating any variable of producing any output. If button *b* is pressed at state *stopwatch/zero*, the ghost microprogram will transition to superstate *run*, which instructions are fully functional. If button *a* is the pressed, the main microprogram will transition to *regular*, and the ghost one to *beep-test*.

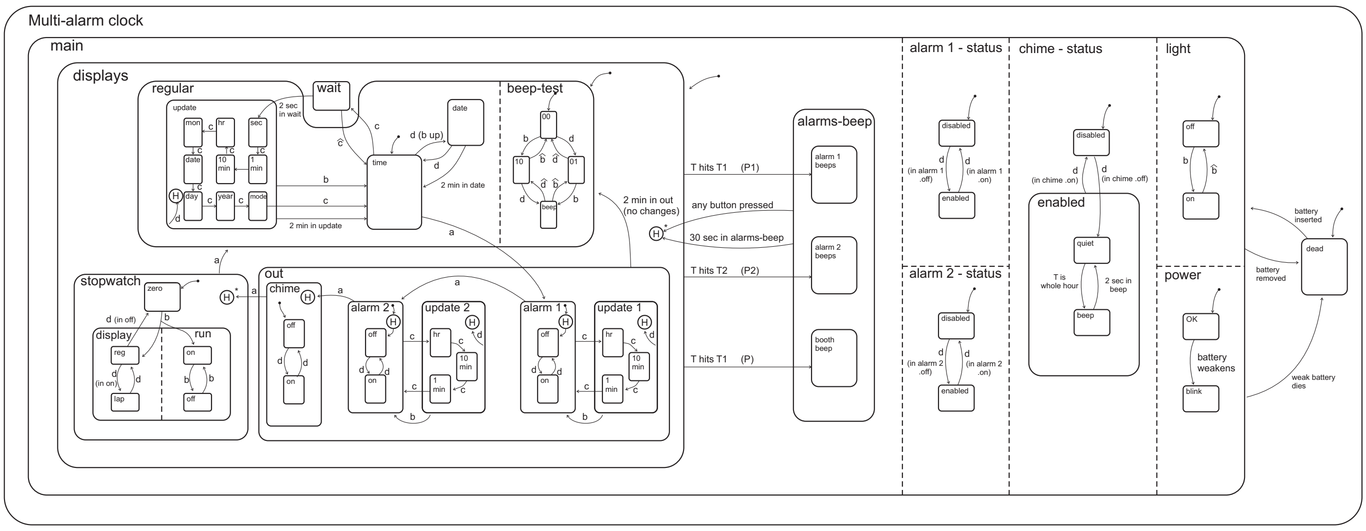
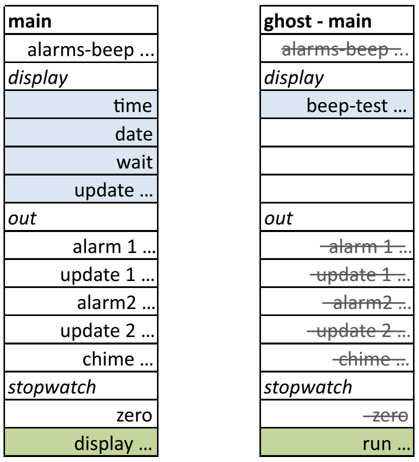


Fig. 2. Recreation of Harel's description of an statechart to control a digital watch



main
alarms-beep ...
display
time
date
wait
update ...
out
alarm 1 ...
update 1 ...
alarm2 ...
update 2 ...
chime ...
stopwatch
zero
display ...

ghost - main
~~alarms-beep~~ ...
display
beep-test ...
out
~~alarm 1~~ ...
~~update 1~~ ...
~~alarm2~~ ...
~~update 2~~ ...
~~chime~~ ...
stopwatch
~~zero~~
run ...

Fig. 3. Main and ghost micro-programs for superstate *displays*. The microinstructions on the ghost program replicate all the transitions taken by the main one, but they only take actions when entering an AND state: at *beep-taste* and *run*. Idle microinstructions are struck-through

B. History

Implementing history may represent a significant challenge in some cases. In this sense, most papers [8] [9] [10] on automatic synthesis of statecharts, do not consider implementing History. Also, in all the cases we have checked, some restrictions apply. The main challenge consists of applying History correctly in a hierarchy of superstates. Let's take as an example superstate *displays*, which must be implemented with History so that, if transitioning back from *alarms-beep*, it should resume at the last executed state. This requires the microprogrammed control to be aware of the difference between deep and shallow superstates in the hierarchy.

For the sake of simplicity, we have chosen the following history scheme. History is kept track of at superstate level. Each superstate that implements History will remember which microinstruction it was running before jumping out, but no one will remember which superstate in the hierarchy was running. Coming back to Figure 2, this means that if returning to superstate *displays* from *alarms-beep* we must choose an specific superstate to return to. We could choose *time* and *beep-test*, for example, despite the user could actually be setting the *chime*. History, however would still be usable, as when moving to *chime*, the superstate will remember which microinstruction it was running. Despite this choice may not fit all the needs, engineers should be able to find workarounds to avoid undesirable situations in most design cases.

C. Microinstruction format

At the beginning of this section, we have addressed the main challenges in mapping a statechart into a microprogram, and suggested a number of restrictions that make implementation feasible. Now, we will propose a microinstruction format and analyse the limitations that the format imposes on the microprogram.

The proposed microinstruction format is divided in to parts: condition evaluation in order to decide which microinstruction to execute next; and actions to take. The latter ones may consist of producing an output of updating an internal variable. Actions are associated to the microinstruction itself, not the transitions. Therefore, it behaves like a Moore automate.

The format is shown in Figure 4. The first part consists of a number of conditions that are evaluated in order. Hence, if the first condition is fulfilled, the remaining ones are ignored. Conditions are based on the values of inputs and/or internal variables, implemented as counters. Conditions may be *and-ed* using the chaining bit (D_i). A condition with the chaining bit *on* is only valid if the next one (or ones) are also valid. Conditions may be *or-ed* by specifying the same target address for 2 or more conditions. Thus, $(a+b)\cdot c \rightarrow target$ is actually implemented as 2 different conditions: $a \cdot c \rightarrow target$ and $b \cdot$

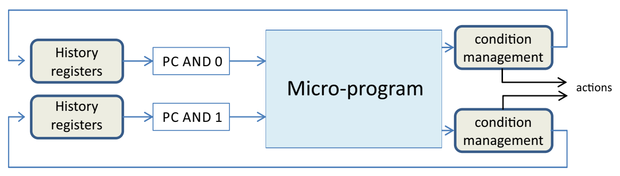


Fig. 6. Circuit that runs up to 2 AND super-states. Dual port memory is concurrently addressed by 2 Program Counters. Conditions are evaluated and actions are taken independently for both superstates. Transitions are refined using the History and new values for PC are produced every cycle.

$c \rightarrow target$. A number of slots for specifying conditions exist. This number is a design parameter, large enough to hopefully cope with any variation of the original design, but not too large to slow down condition evaluation in highly demanding applications.

In most cases, enough slots should be provided at design time to evaluate any condition. That would not be the case for an specific application, the designer must consider whether it is acceptable to evaluate conditions in more than one step. That should usually be possible unless time restrictions are exceptionally tight. Hence, a first microinstruction will evaluate a limited number of general conditions diverting the program flow to more specific microinstructions that will refine decision taking. An example is shown in Figure 5. Instead of evaluating 6 multi-variable conditions in a single microinstruction, only 4 single-variable conditions are evaluated at most. As it can be seen in this example, 2 transitions are made in a single step, whereas the remaining 4 need 2 steps.

The second part of the format consists of a list of actions, which are of 2 types: outputs and updates. A number of outputs are defined in the architecture, with a set of predefined values (more about this on Section IV-E. A microinstruction may select an output specifying the index of the output and the code of the value. Regarding updates, internal variables are managed as counters. As it is explained in Section IV-B, those counters may be updated in several ways. Actions on counters are encoded as the index of the counter followed by the action code. Outputs and counters indexes are consecutive in the same list, so any mix of both is allowed in the list of actions.

It must be understood that, in the case of a large number of conditions and/or actions, the length of the microinstructions will grow accordingly. On the one hand, designers may decide to limit the maximum number of conditions or actions simultaneously encoded in a microinstructions. On the other hand, using tens of memory blocks from any modern FPGA should not be a problem. Therefore, it should be possible to create long microprograms made of wide microinstructions without serious cost concerns. In Section VI, the cost of microinstruction decoding will be addressed.

IV. Architecture

We now describe a generic architecture focusing on the following aspects: storing and accessing the microprogram; evaluating and updating counters, inputs and conditions; generating outputs; and loading the configuration.

A. Storage

A number of RAM blocks are used to store the microprogram. Generally, several blocks are laid out *horizontally* forming a row so that, the more the blocks, the longer the microinstruction format. Each row of RAM blocks may be accessed using 2 ports. Therefore, it is possible to fetch 2 microinstructions in the same cycle. This allows implementing, for example, 2 AND states. However, more AND states could be needed. Therefore, several rows must be instantiated in order to allow further concurrency.

Hence, the microprogram is stored in a set of rows of RAM blocks. Each row is long enough to allow long microinstructions. Each row hosts (potentially) 2 AND states, although one or both could be idle.

Each AND state is accessed using one PC register (program counter). Each AND state is built of one or more states or super-states. As RAM blocks in FPGAs support hundreds or thousands of data words, any AND state should fit in it allocated space even if it embraces a significant number of sub-states. PC is used to address the memory directly but, most importantly it must be updated allowing the implementation of history when switching between different OR superstates. This is done using the circuit in Figure 7. Up to 8 history registers are proposed, as it is not expected that an AND state contains a larger number of OR states. Each of those resisters stores the target address for entering a given OR state.

Every time a microinstruction is evaluated, a proposed next value of PC is produced (newPC). If newPC is lower than 8, it signals jumping to a new OR state. In that case, the history register of the current OR state is updated, and the target address for the new OR state is retrieved from the history register. These selection operations are performed using the multiplexers and de-multiplexers in the figure. Naturally, those superstates that do not support history, will not update their history register. This scheme is simple, and only 8 lines in the micromemory are wasted.

B. Counter operation

Counters are implemented as 32-bit registers connected to a 32-bit adder, as shown in Figure8. By selecting the inputs at the multiplexer, it is possible to update the content of the counter in several ways: reset; increment and decrement; set to $ref0$ or $ref1$; add $ref0$ or $ref1$; and set to $ref0 + PC$, which has special uses as it will be shown in Section V.

The value of each counter is always compared with its 2 reference values. The outcome of those comparisons will control the execution of the microprogram. Reference values are both used for comparison (as a proper counter), and to set new values (as a variable). Normally, each counter will use reference values in only one of those ways.

C. Input evaluation

Similarly to counters, inputs are evaluated by means of comparisons. However, inputs cannot be updated by the statechart, so the circuit for evaluating inputs consists of

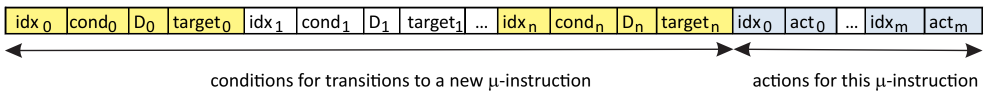
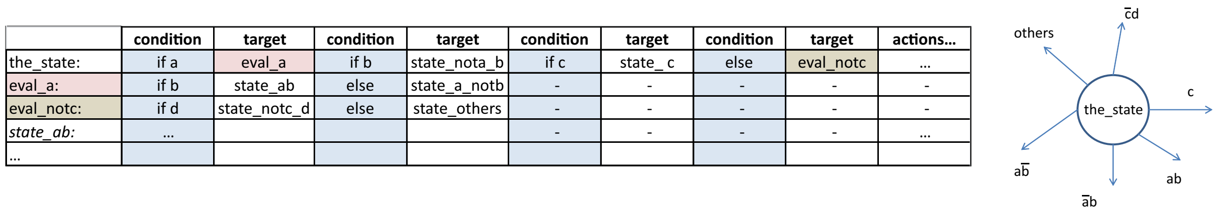


Fig. 4. Microinstruction format showing n+1 conditions and m+1 actions. The length of most fields may vary depending on the number of allowed counters and inputs and outputs; or the maximum number of microinstructions in an AND superstate



	condition	target	condition	target	condition	target	condition	target	actions...
the_state:	if a	eval_a	if b	state_nota_b	if c	state_ c	else	eval_notc	...
eval_a:	if b	state_ab	else	state_a_notb	-	-	-	-	-
eval_notc:	if d	state_notc_d	else	state_others	-	-	-	-	-
state_ab:	...				-	-	-	-	...
...									

the_state
others
c̄d
c
ab
āb
ab̄

Fig. 5. Splitting a variety of state transitions in two step. Auxiliary states *eval_a* and *eval_notc* help to implement a larger number of transitions than those directly supported by the format.

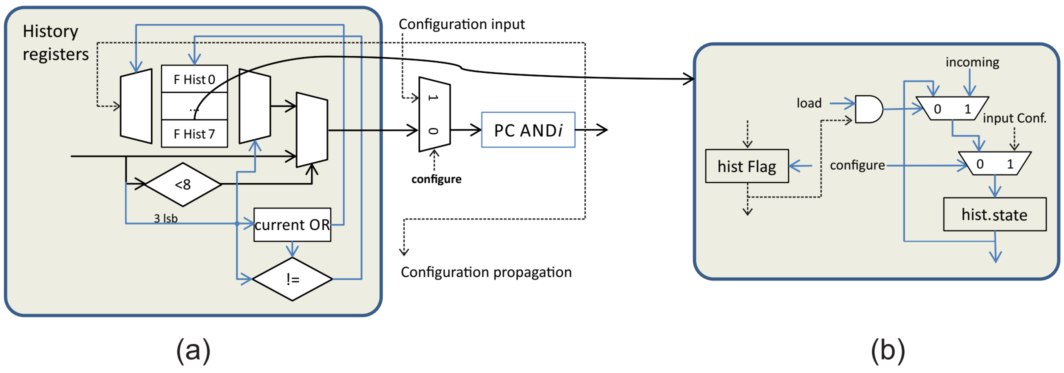


Fig. 7. History register contains the entry address for OR states within an AND superstate. In case of implementing history, special values of PC are substituted by those stored in the history registers. History is updated when leaving the current OR state. History is initialized at configuration time as shown in part (b) of the figure.

just 2 comparators plus the registers to load and hold the configuration. This scheme is shown in Figure 9.

D. Condition chaining

Transitions from one state to another depends on the evaluation of one or more conditions. The engineers must specify the conditions sorted by priority. Each condition consists of 4 fields. The first one is the index of the counter or input evaluated for that condition. The length of this field depends on the total number of counters and inputs. The second field is a code that selects the specific condition: equal/lesser/greater than the first/second reference value. An additional bit negates the condition, allowing to evaluate different, greater or equal and lesser or equal. The third field is a single bit (Di) that encodes if another condition must be *and-ed* or not. Finally, the fourth field is the address of the next microinstruction, if the condition is evaluated positively.

Condition chaining is implemented by the circuit shown in Figure 10. In this case, up to 8 conditions can be cascaded, but a simpler scheme could sufficient in most cases. Only one of the 8 output bits is true. If all are false, the same microinstruction will be evaluated next cycle. Those bits are *and-ed* with their corresponding address, and then *or-ed* together. This implements a multiplexer with a decoded selection signal.

E. Output selection

A number of different outputs are implemented, even if the system does not make use of all of them. Similarly to counters, outputs may use 2 reference values that are loaded at configuration time. The microprogram will select the value of a number of outputs in each microinstruction using an index that selects one input of a multiplexer. This is shown in Figure 11. The microprogram will select among the reference values and the content of several counters. Some outputs may be significant only at a given time, while others should hold its value until changed. This is implemented by using a register and an multiplexer. Similarly to what happens in FPGA's logic blocks, the output may be registered or not. We consider that this behavior does not change at run time, so a flag is loaded during configuration that rules the multiplexer.

F. Loading configuration

Configuration is to be loaded word by word using the data network and propagated through the hardware in a serial fash-

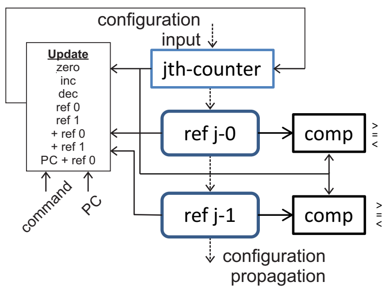


Fig. 8. Counter implementation. At configuration time, an initial value is loaded, together with 2 reference values. On normal operation, the counter will update according to commands sent by the active states. Also, it provides the result of comparing the current value with the reference ones.

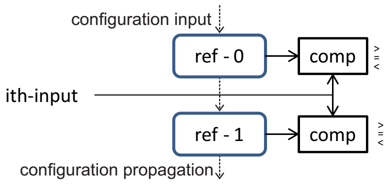


Fig. 9. Input checking. At configuration time, 2 reference values are loaded. On normal operation, the result of comparing the current value with the reference ones is calculated.

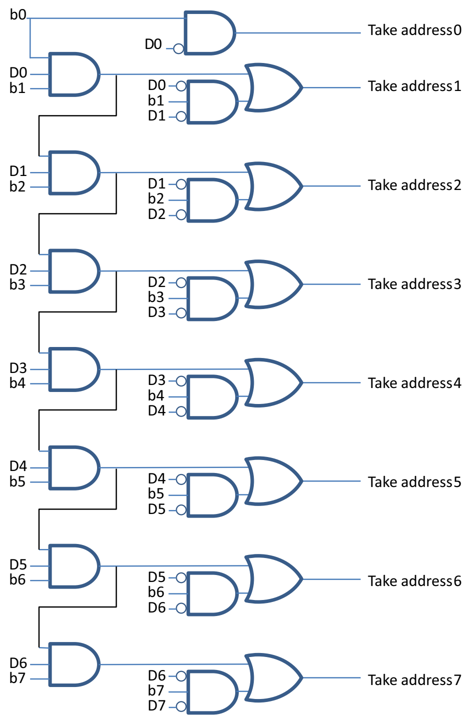


Fig. 10. Evaluation chaining for up to 8 conditions. The result of each individual evaluation is bi. The bit Di signals that the evaluation should be delayed and combined with the next one(s). Eventually, only one output bit will be active, signalling the selected state transition.

ion. This emulates the configuration of FPGAs through JTAG. All the memories and registers are chained in order to allow propagating the configuration, which includes: microprogram; the initial content of History registers; initial and reference values of counters; and reference values for outputs.

In Figure 12, it is shown how the microprogram is loaded. A configuration counter is kept that enables writing in selected RAM blocks. In this particular example, memory blocks are 32-bit wide, and input words are byte sized. Hence, 4 bytes have to be gathered before issuing the write signal. Each block stores 512 words, and there are 4 rows with 4 blocks each. Naturally, this layout can be easily changed. In Figure 13, a double-ported RAM block is shown. Each port serves one AND state, and the most significant bit is hardwired ('0' for port A, and '1' for port B) to differentiate the address space of each super-states. In configuration mode, however, port A is used to write all configuration words.

The configuration of History registers can be explained using Figure 7. For each potential OR super-state, and 8-bit register holds the target state, and one flag signals if that super-state makes use of History or not. The state registers are chained together, and the 8 flags are actually implemented as an 8-bit register chained with the other ones.

The same chaining mechanism is used to load the reference values of counters and outputs. For each of them, 1 initial value and 2 reference values are loaded, each of them 32-bit long. Also, for convenience, micro-memory is located at the end of the configuration chain so that when the microprogram is fully loaded, all the other configuration words are loaded as well.

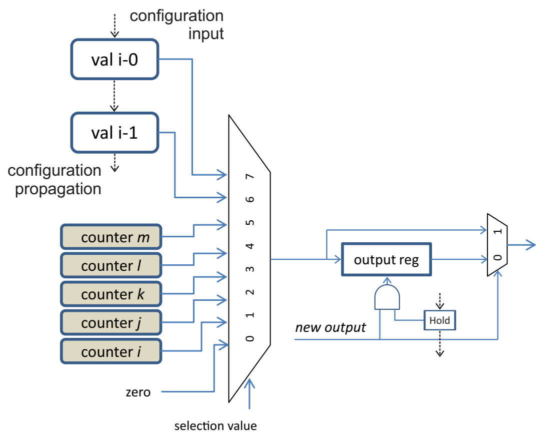


Fig. 11. Circuit for output selection. The selection value is obtained from the microinstructions, which selects the output from a set of counters and 2 reference values loaded at configuration time. The output may be registered or not.

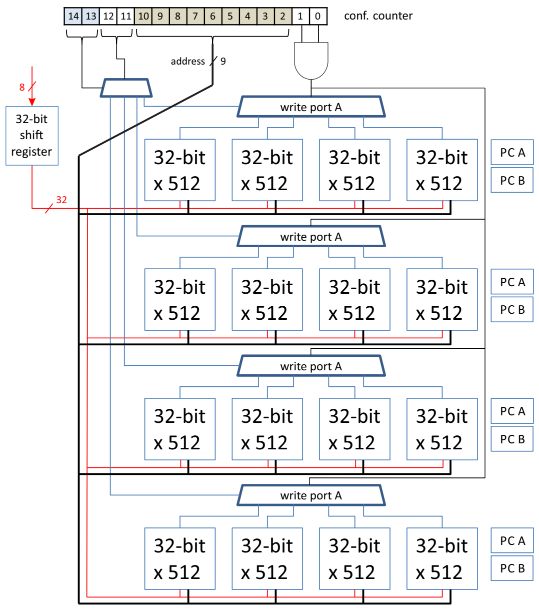


Fig. 12. Microprogram loading using 4 rows of 4 banks each. Configuration bytes are concatenated until 32 bit may be loaded into a memory block. A simple configuration counter enables loading each word in the right position, row and memory block.

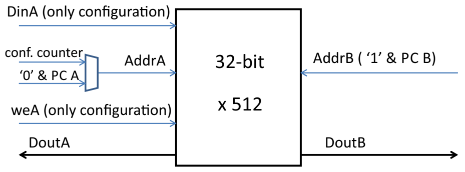


Fig. 13. Dual-ported RAM block. Two AND states may be addressed simultaneously. Configuration is loaded using port-A, exclusively.

V. Case Example

As an implementation example, we will now adapt the statechart in Figure 2 and look into some specific parts in detail. The statechart is unambiguously defined, but our architecture will leave some room for changes.

The inputs are: 4 buttons labeled *a*, *b*, *c* and *d*; plus status inputs labeled *chimeOnOff*, *oclock*, *batteryStatus*, *testT1* and *testT2*. Buttons' input values may be 0 (nothing), 1 (pressed) or 2 (released). The counters are *Timer1* and *Timer2* (their reference values may be set during configuration, allowing to fine-tune the behavior of the watch); *displayUpdate*; *alarmUpdate1* and *alarmUpdate2*; *chimeOnOff*; *countStopWatch*; and *countAlarm*. The outputs are: *light*, *beep*, *display* and *change*.

We have spotted 7 AND states from Figure 2. Rounding up to 8 will provide some room for future changes. Therefore, 4 rows of memory blocks will be needed. Super-states will be laid out as shown in Figure 14. In order to provide a significant, but not too long example, the implementation of *alarm1* and *update1* in *main/displays/out* is shown in Figure 15. Possible transitions are: time-out after 120 seconds without pressing any button (Timer2 is used); and pressing

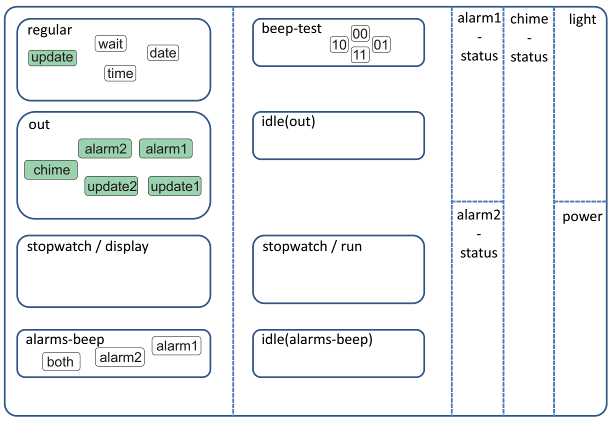


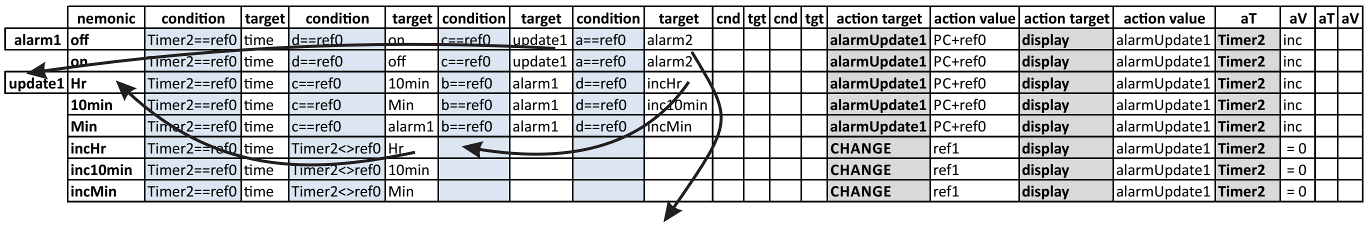
Fig. 14. Super-states layout for the example in Figure 2. Dashed lines separate AND superstates. The leftmost one carries out most of the tasks, while its neighbour is mainly a ghost superstate with the exception of *beep-test* and *stopwatch/run*. The remaining 5 are very simple but, hierarchically, are at the same level as the main ones.

one of the buttons, with different effects. Actions require more explanations: *alarmUpdate1* is a counter used to keep track of the current micro-instruction. It value is send to *display* (an output) so that the display of the watch reflects the settings currently edited. By configuring *ref0* for that counter to $X - PC(alarm1)$, *alarmUpdate1* will send codes X, $X+1$, $X + 2$, and so on to the display. Output *Change* is activated (*ref1 = 1*) when some value is updated.

Conditions are (potentially) evaluated over the value of 9 inputs and 8 counters, totalling 17. Rounding up to 32, 5 bits would be use to encode the condition index. A reasonable distribution could be 16 inputs and 16 counters. In this particular case, some microinstructions may jump to 4 different addresses, never more. However, we may consider to allow up to 6 different transitions in order to have a future-proof design. Each transition needs 4 bits for the index; 2 bits for the comparison (*equal*, *greater*, *lesser*); 1 bit for the reference value (ref_{j0} or ref_{j1}); 1 bit to invert the comparison; 1 bit for chaining conditions; and 8 bits for the destination address. In total, 18 bits times 6 conditions, equals 108 bits.

Also, up to 3 actions should be taken for the same microinstruction. Again, we extend this number to 4. Actions apply on counters and outputs. As using 16 counters was proposed, and there are 4 possible outputs, the number of outputs may be also extended to 16 so that 32 indexes are supported encoded as 5-bit numbers. The proposed number of operations on each counter or output is 8. Hence, each action is encoded as 5 + 3 bits, and the 4 actions are encoded using 32 bits. Added to the previous 108, 140 bits are needed per microinstruction. Using 32-bit words on each memory block, memory rows should be 5 blocks wide. In that case, the memory scheme would be very similar to the one in Figure 12 only that a slightly more sophisticated counter would be needed in order to deal with 5 memory blocks, as 5 is not a power of 2.

In summary, the microprogrammed control would be implemented using 4 rows of 5 memory blocks each; 16 inputs,



	nemonic	condition	target	condition	target	condition	target	condition	target	cnd	tgt	cnd	tgt	action target	action value	action target	action value	aT	aV	aT	aV
alarm1	**off**	Timer2==ref0	time	d==ref0	on	c==ref0	update1	a==ref0	alarm2					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
	on	Timer2==ref0	time	d==ref0	off	c==ref0	update1	a==ref0	alarm2					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
update1	**Hr**	Timer2==ref0	time	c==ref0	10min	b==ref0	alarm1	d==ref0	incHr					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
	10min	Timer2==ref0	time	c==ref0	Min	b==ref0	alarm1	d==ref0	inc10min					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
	Min	Timer2==ref0	time	c==ref0	alarm1	b==ref0	alarm1	d==ref0	incMin					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
	incHr	Timer2==ref0	time	Timer2<>ref0	Hr									**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		
	inc10min	Timer2==ref0	time	Timer2<>ref0	10min									**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		
	incMin	Timer2==ref0	time	Timer2<>ref0	Min									**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		

Fig. 15. Micro-code example for 2 selected superstates. The format supports up to 6 transitions and 4 actions. Some transitions are highlighted with arrows for the sake of clarity. Condition-chaining bits are not shown. Both superstates support History.

TABLE I
FPGA RESOURCE UTILIZATION PARTICULARIZED TO THE WATCH CONTROL EXAMPLE

Component	LUT	FF	BRAM
AND superstate	872	83	5/2
counter	247	96	
input	56	64	
output	225	97	
total	16203	4776	20

counters and outputs. Of those, 7 inputs; 8 counters; and 12 outputs would be available for upgrading the control algorithm. Also, and additional AND superstate would be supported and several microinstructions may be added to any of them.

VI. Evaluation

With respect to resource usage, this is chiefly related to the amount of memory blocks required to store the micro-program; and the number of inputs, counters and outputs. As the microprogrammed architecture is not tailored to implement a given application, but all the possible variations of a given one, the cost of implementing the individual components is presented in Table I. However, much of the implementation cost is due to large multiplexers, which size depends on the number of counters, inputs and outputs. Therefore, figures in Table I are particularized to the example case in Section V.

Pure logic (LUT, look-up tables) is the most used resource. Essentially, being able to select and address a large number of counters and outputs requires an large interconnect network that is implemented using wide multiplexers and demultiplexers. This particular design can be implemented using any Xilinx Artix-7 device with the exception of the 2 smaller ones (12T and 15T). Also, this design can be fitted at least 4 times in any Kintex-7 FPGA [11].

Despite we have not obtained any figures for a hardwired implementation of the same control system, we assume that there should be a difference of more than 1 order of magnitude. Clearly, a hardwired implementation would be preferred when resource utilization is an important factor. However, not all designs are as complex as the proposed one and, in any case, on-field programmability should be the main factor to decide between a microprogrammed or hardwired approach.

Finally, clock speed has been evaluated for Xilinx xC7A25T and xC7K70T devices using average speed grades. The minimum speed was 115 MHz, sufficient for our applications.

VII. Conclusion

Complex control systems can be clearly specified using statecharts. Mapping statecharts into hardware allows fast and jitter-free implementations in mission-critical applications. This work addresses a specific topic: implementing a statechart in hardware in such a way that it can be easily upgraded using firmware, without re-synthesising the architecture. That would allow maintaining the control systems even if the devices and/or design tools are no longer supported. Upgradability is crucial when ASICs are used, but it may be also important even when using reconfigurable hardware. A substantial hardware overhead must be expected. However, this issue may be a minor concern considering the substantial amount of resources currently offered even by de smaller devices. As future work, it is planned to develop a protocol that would allow upgrading the firmware using the data connection. Also, an automatic tool has been developed that converts an statechart specification into HDL code. We plan to extend that tool to generate firmware for the microprogrammed architecture.

References

[1] M. Milkes, “The genesis of microprogramming,” *IEEE Annals of the History of Computing*, vol. 8, pp. 116–126, 1986.

[2] W. G. Spruth, *The Design of a Microprocessor*. Springer-Verlag.

[3] D. Harel, “Statecharts: A visual formalism for complex systems,” *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[4] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Logic Optimization*. Springer, 1997.

[5] T. Muller-Wipperfurth and R. Hagelauer, “Graphical entry of fsmds revisited: putting graphical models on a solid base,” in *Proceedings Design, Automation and Test in Europe*, 1998, pp. 931–932.

[6] V.-A. V. Tran, S. Qin, and W. N. Chin, “An automatic mapping from statecharts to verilog,” in *Theoretical Aspects of Computing - ICTAC 2004*, 2005, pp. 187–203.

[7] S. Qin, W.-N. Chin, J. He, and Z. Qiu, “From statecharts to verilog: a formal approach to hardware/software co-specification,” *Innovations in Systems and Software Engineering*, vol. 2, no. 1, pp. 17–38, Mar 2006.

[8] R. Kol, R. Ginosar, and G. Samuel, “Statechart methodology for the design, validation, and synthesis of large scale asynchronous systems,” in *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996, pp. 164–174.

[9] K. Buchenrieder and C. Veith, “A prototyping environment for control-oriented hw/sw systems using state-charts, activity-charts and fpga’s,” in *EURO-DAC’94, European Design Automation Conference*, 1994, pp. 60–65.

[10] R. Findenig, T. Leitner, V. Esen, and W. Ecker, “Consistent SystemC and VHDL code generation from state charts for virtual prototyping and RTL synthesis,” 2011.

[11] “Xilinx Kintex-7,” https://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html, accessed: 2019-04-19.