Università degli Studi di Pavia
Facoltà di Ingegneria

Dottorato di Ricerca in Microelettronica
VIII nuova serie (XXII ciclo)

Application Specific Instruction-set Processor for Hard Disk Drive Servo operation: the Microprogrammed Servo Sequencer

Advisor: Prof. Carla VACCHI

Ph.D. Dissertation
of Paola BALDRIGHI

to Stefano

Contents

1 Introduction

The Hard Disk Drive is the most common device used for data storage. The introduction of Perpendicular recording provides an increasing of hard disk user data density. The direct consequence is the increasing complexity of the hard disk drive read/write channel logic: in particular also the operation of track seek and track following must be more accurate in respect of new media specs. In fact each Hard Disk Drive read/write system needs a control logic for the head positioning to identify the correct user data sector that has to be read or written. These positioning operations are controlled by the Servo Subsystem that is part of the Hard Disk Drive R/W channel. Servo Subsystem realizes the correct head positioning on the track decoding servo sectors (track servo data are recorded along the tracks).

Servo Subsystem is usually implemented with an hardwired approach: a Finite State Machine. The presence of different Hard Disk Drive Market Segments, characterized by various kinds of media supports, with the necessity of peculiar Read/Write channel systems, needs different Servo sectors and consequently different Servo subsystem. The Finite State Machine approach reaches the best performance, but loses in terms of flexibility, since the behavior of the Servo subsystem couldn't be modified after Integrated Circuit (IC) fabrication if some changes of the design are needed: the hardwired Finite State Machine implementation, and so its behavior, cannot be modified.

The proposed ASIP, the Micro-programmed Servo Sequencer (MSS), maintains the same behavioral characteristics of the Servo Subsystem considering also same performances in terms of Servo System elaborating frequency, adding an important feature: its flexibility. The MSS Instruction Set Architecture (ISA) has been customized to emulate a generic Servo System behavior and takes into consideration different Hard Disk Drive market segments. The ISA contains all the fundamental instructions for the correct description of the Servo System behavior. The simple ASIP architecture and some embedded operations allow the achievement of the appropriate performance, needed to implement also the most

critical control systems. The Microprogrammed Servo Sequencer flexibility characteristic concerns two aspects. The first one involves the parametric feature of the RTL source code that describes Microprogrammed Servo Sequencer: it is possible to dimension all Instruction Set fields (except the operating code that identify instruction type) to fit different Servo system features. The second aspect involves the reprogramming feature of the MSS Instruction Memory, MSS Register Memory and MSS Reconfigurable I/O ports: the reprogramming process allows initializing the MSS behavior with the appropriate firmware according to the HDD market segment, so it is possible to make changes of this behavior after IC fabrication reprogramming the microcontroller with a specific firmware, thus reducing design costs of different Servo System.

This work proposes a complete front end design for the description of Servo System.

In Chapter 2 the Servo System operations are explained and the hardwired approach is compared with the programmable one.
In Chapter 3 the Microprogrammed Servo Sequencer is presented: it is composed by an elaborating core, embedded counters, programmable I/O ports and a Reprogramming Finite State Machine; the MSS instruction set is described and custom instructions, such as COUNTER one, are explained in details.
Chapter 4 begins with an introduction to the verification process, then Assertion Based Verification explanation and an example of this verification applied to JUMPCS instruction are shown.
In Chapter 5 the Microprogrammed Servo Sequencer is customized on a real case of Servo System. A dedicated firmware has been written, the MSS has been reprogrammed, its correct behavior has been verified and synthesis results are compared with real Servo System.

2 Hard Disk Drive Servo operations

Each Hard Disk Drive (HDD) needs a control logic, that realizes the heads alignment, for the correct user data reading or writing. This logic is called Servo Subsystem and it is realized by means of a hardwired Finite State Machine. The Application Specific Instruction-set Processor (ASIP) programmable approach is an alternative implementation for reducing the non-recurring design, verification, layout and test costs: it is possible to map different generations of Servo FSM onto the same ASIP.

2.1 The Hard Disk Drive

The Hard Disk Drive (HDD) is a mass storage device formed by a spindle which holds one or more flat circular disks called platters, made from a non-magnetic material (for example glass or aluminum alloy) and coated with a ferromagnetic layer on which user data are recorded by means of a magnetization operation (Figure 2.1.1).

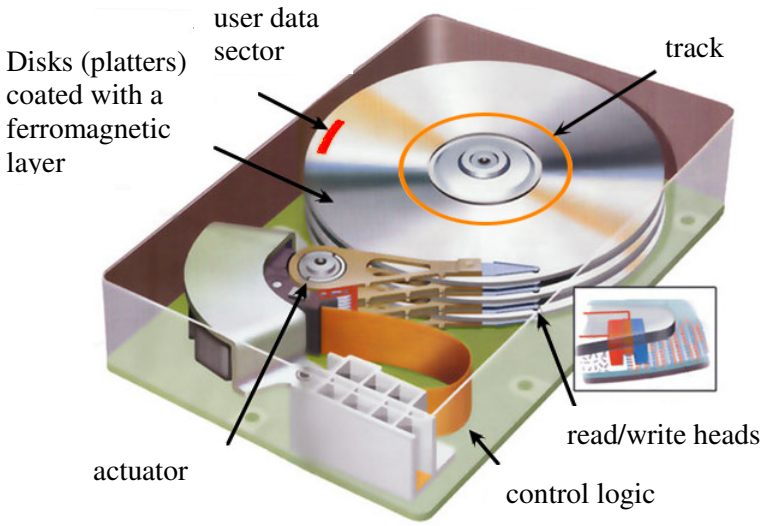

Figure 2.1.1 Hard Disk Drive

The platters are spun at high speed and different heads, moved by means of a mechanic arm over the track that has to be read/written, realize the data read and write operations reading or modifying the magnetization of the ferromagnetic material. Each platter is organized in tracks, concentric rings, separated by interspaces called *gaps*. Each track is organized in many sectors that generally contain 512 bytes each and are separated by spaces called *intersector gaps*. Due to the Hard Disk geometry the more external tracks contain more sectors than the more internal ones. The read and write operations are realized moving the heads over the correct track; thanks to the disk rotation it is possible to read or write the desired sector. Each platter needs two heads for reading and writing process. The correct head alignment over the correct track is achieved by means of servo information reading [2.1], [2.2].

2.2 Servo data and operations

Each Hard Disk Drive (HDD) read/write system needs a control logic for the head positioning to identify the correct user data sector that has to be read or written. These positioning operations are controlled by the Servo Subsystem that is part of the Hard Disk Drive R/W channel [2.3], [2.4]. Not only at the read/write system startup and every time there is a track change, but also during track following, the Servo Subsystem is conti-

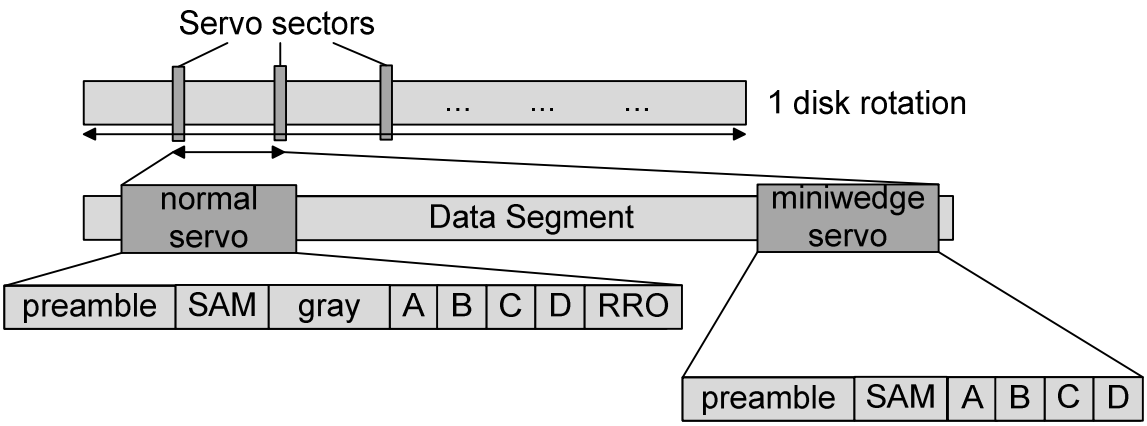

Figure 2.2.1 User, normal servo and miniwedge servo data

nuously running and aligns the head over the track itself.

The servo data are recorded along each track, distributed in each wedge of the Hard Disk (Figure 2.2.1). When the read/write head moves from one track to another (track seek) it is necessary to read a *normal servo* sector that contains information about the signal phase and gain (*preamble*), the end-of-synchronization recognition sequence (*Servo Address Mark*), the encoded servo sector location on the hard disk (*graycode*), the head perfect positioning on the hard disk track (*A-D* bursts) and eventually the hard disk track eccentricity (*Repetable Run Out*).

During recording or retrieval of user data while staying on the same track it is necessary to regulate the head position (track following). The *miniwedge servo* sector supplies this information: it contains only the *preamble*, the *Servo Address Mark* and the *bursts* fields [2.5], [2.6] [2.7].

2.3 Hardwired vs. Programmable approach

The Servo Subsystem realizes the correct head positioning on the track decoding the servo sectors by means of hardwired Finite State Machines (FSM). The main one, the arbiter, decides on the behavior of the Servo Subsystem: Regular (for correct head positioning) or Spiral (Repeatable Run Out writing process). The Regular process is implemented by the Regular Servo Sequencer (RSS) Finite State Machine that realizes the correct head positioning with the support of other Servo system embedded blocks. A hardwired FSM approach would reach the best performance, but losing in terms of flexibility, since after Integrated Circuit (IC) fabrication the behavior of the Servo subsystem couldn't be modified if some changes of the design are needed.

The scaling down process increases the design complexity, so non-recurring design and manufacturing costs (Figure 2.3.1). It is possible to integrate more transistors on the same die and their number is exponentially high. This complexity forces the introduction of different computer-aided design (CAD) tools, more expensive to acquire and maintain, to better manage the hierarchical block level designs. In addition to the digital part, it is

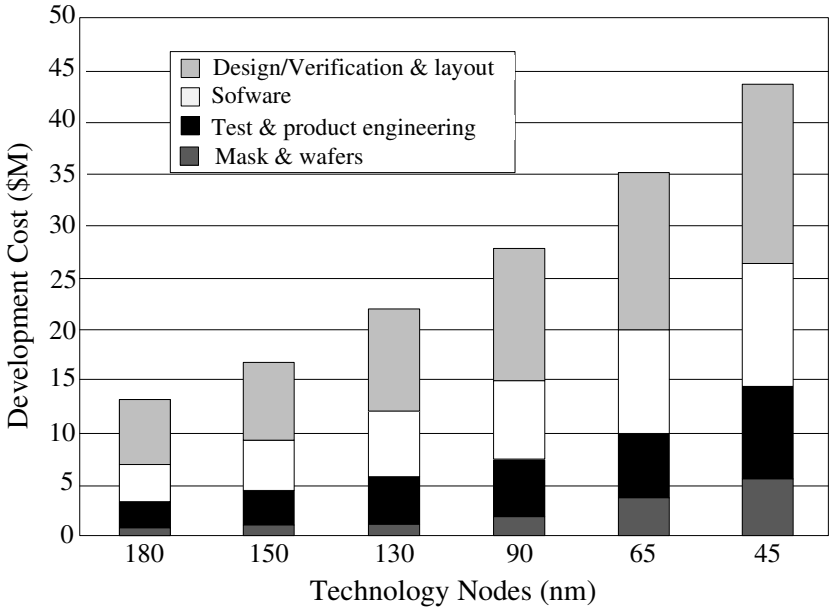

Figure 2.3.1 ASIC non-recurring design and manufacturing costs

possible to integrate also analog and mixed signal components on the same die increasing also design, verification and layout costs. Besides the chance of silicon failure is quite high causing higher test and product engineering costs. At last the cost of a mask set for sub-100nm designs is multi-million dollar [2.8].

These high non-recurring design and manufacturing costs imply either larger break even volumes at fixed per-unit costs, or prohibitive per-unit costs at fixed volumes. The programmable approach is an alternative implementation to ASICs that is rapidly emerging. An example is the Application Specific Instruction-set Processors (ASIPs). The programmability of these devices enables the mapping of different generations of an application onto the same ASIP reducing so the non-recurring design, verification, layout and test costs. A programmable approach provides also a much lower risk because for different application generations it is necessary to write and debug only firmware, not working hardware.

The firmware solutions on an ASIP cause a productivity benefit, but also a loss of design quality (measured in area, delay, power). This disadvantage is acceptable because they are more flexible. The ASIP approach allows designing an embedded device for a specific application maintaining the ASIC performances with a programmable characteristic.

The presence of different Hard Disk Drive Market Segments, characterized by various kinds of media supports, with the necessity of peculiar Read/Write channel systems, needs different Servo sectors and consequently different Servo subsystem. A programmable architecture, an Application Specific Instruction-set Processor (ASIP) [2.9],[2.10] reduces the high cost of Servo Subsystem design. The proposed ASIP, the Microprogrammed Servo Sequencer (MSS), maintains the same elaborating frequency of the Regular Servo Sequencer and offers an incomparable degree of flexibility.

The MSS Instruction Set Architecture (ISA) has been customized for the emulation of a generic RSS behavior and takes into consideration different Hard Disk Drive market segments. The ISA contains all the fundamental instructions for the correct description of the Regular Servo Sequencer. The simple ASIP architecture and some embedded operations allow the achievement of the appropriate performance, needed to implement also the most critical control systems, like RSS [2.11]. The reprogramming process allows initializing the Micro-programmed Servo Sequencer behavior with the appropriate firmware according to the HDD market segment, so after IC fabrication it is possible to make changes of this behavior reprogramming the microcontroller with a specific firmware, thus reducing design costs of future RSS.

2.4 Bibliography

[2.1] C. D. Mee and E. D. Daniel, "Magnetic Storage Handbook", McGraw-Hill Professional, New York, USA, 1996.

[2.2] Marco Maurizio Maggi, "Progetto di un controllore riconfigurabile per la gestione dei segnali servo nell'hard disk drive", Tesi di Laurea Specialistica in Ingegneria Elettronica, Università degli Studi di Pavia, a.a. 2006/2007.

[2.3] S. R. Tawfeic, "Track seeking control for hard disk drives using the approaching index switching algorithm", *Proc. 2008 International Conference on Computer and Communication Engineering*, pp. 172 – 175, May 2008.

[2.4] H. Yada, T. Yamakoshi, H. Ishioka, and N. Hayashi, "Synchronous servo scheme using maximum-likelihood detectors", *IEEE Transactions on Magnetics*, Vol. 39, no. 6, pp. 3593 – 3603, Nov. 2003.

[2.5] H. Yada, H. Ishioka, T. Yamakoshi, Y. Onuki, Y. Shimano, M. Uchida, H. Kanno, and N. Hayashi, "Head positioning servo and data channel for HDDs with multiple spindle speeds", *IEEE Transactions on Magnetics*, Vol. 36, no. 5, pp. 2213, Sep. 2000.

[2.6] T. Hamaguchi, H. Maeda, K. Usui, and K. Shishida, "An alternating DC track servo pattern for perpendicular recording", *IEEE Transactions on Magnetics*, Vol. 41, no. 10, pp 2872, Oct. 2005.

[2.7] Al-Mamun, T.H. Lee, G.X. Guo, W.E. Wong, W.C. Ye, **"**Measurement of position offset in hard disk drive using dual frequency servo bursts", *IEEE Transactions on Instrumentation and Measurement*, Vol. 52, no. 6, pp. 1870 – 1880, Dec. 2003.

[2.8] D. R. Martinez, R. A. Bond, M. M. Vai, "High Performance Embedded Computing Handbook: A Systems Perspective", CRC Press Inc, June 2008.

[2.9] K. Keutzer, S. Malik, A. R. Newton, "From ASIC to ASIP: The Next Design Discontinuity", *Proceedings of VLSI in Computers and Processors*, pp. 84-90, 2002.

[2.10] S. Saponara, L. Fanucci, S. Marsi, G. Ramponi, D. Kammler, E.M. Witte, "Application-Specific Instruction-Set Processor for Retinex-Like Image and Video Processing", *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 54, Issue 7, pp. 596 – 600, July 2007.

[2.11] R. Leupers, K. Karuri, S. Kraemer, M. Pandey, "A design flow for configurable embedded processors based on optimized instruction set extension synthesis", *Proceedings of Design, Automation and Test***,** Vol. 1, pp. 6, March 2006.

3 The Microprogrammed Servo Sequencer

The Microprogrammed Servo Sequencer (MSS) is an Application Specific Instruction-set Processor (ASIP). A typical RISC processor, the MIPS (Microprocessor without Interlocked Pipeline Stages), has been studied to derive MSS elaborating core architecture. The MSS Instruction-set implements Servo control logic operation. To achieve better performances some embedded logic which communicates with MSS are introduced. A reprogramming process by means of a Finite State Machine can change the behavior of MSS.

3.1 MIPS

The MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC processor with 4 pipeline stages (Figure 3.1.1) born in 1981 at Stanford University with professor Hennessy. Each instruction has to go through the Instruction Fetch, the Instruction Decode, the execute, the memory access and the write back. The first MIPS processors have a 32-bit architecture; the last ones have a 64-bit architecture. In the following paragraphs the Microprogrammed Servo Sequencer Instruction-Set Architecture has been described in comparison to MIPS features.

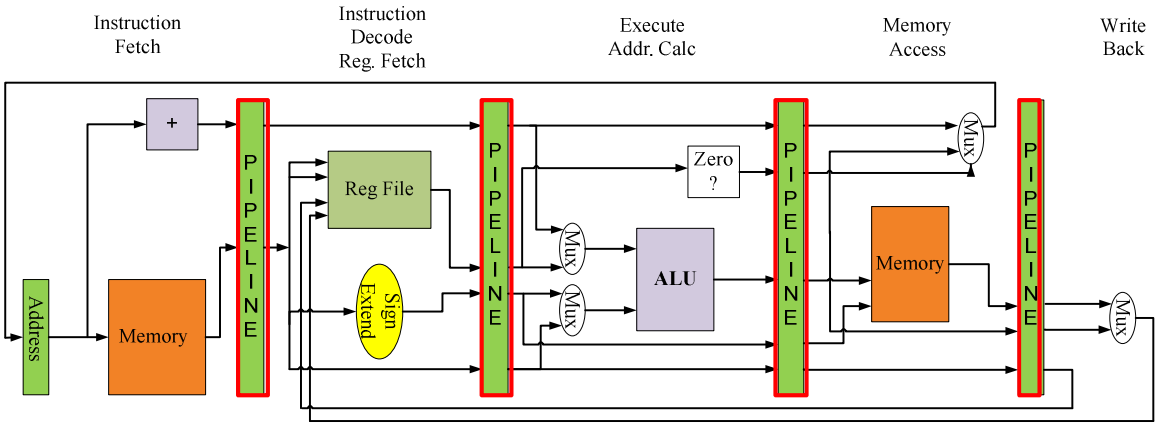

Figure 3.1.1 MIPS architecture

3.2 Application Specific Instruction-set Processor architecture

The ASIP Architecture is derived from the Microprocessor without Interlocked Pipeline Stages, which presents more than 64 different 32 bit instructions [3.1]. The MIPS instruction-set is a Reduced Instruction-set Computer (RISC) approach. The characteristic of a RISC approach are:

- ✓ high clock cycle frequency;
- ✓ low instruction execution time;
- ✓ fixed instruction length which involves simple instruction-set architecture;
- ✗ large firmware because the instruction-set is composed by simple instructions.

The alternative to RISC approach is the CISC architecture that presents these features:

- ✗ reduced clock cycle frequency;
- ✗ high instruction execution time due to complex instructions;
- ✓ not fixed instruction length;
- ✓ small firmware because CISC instructions implement complex operations.

The choice of a RISC approach brings to simpler microcontroller hardware architecture than a Complex Instruction-set Computer (CISC) one. The memory architecture is Harvard (Figure 3.2.1): there are two memories for data and instruction, so it is possible to access in one clock cycle both to data memory and to instruction memory. The Von

Neumann (Princeton) memory architecture implies only one memory for both data and instructions decreasing microcontroller performances: the execution of a single instruction may need at least two clock cycles increasing firmware execution latency (Figure 3.2.2).

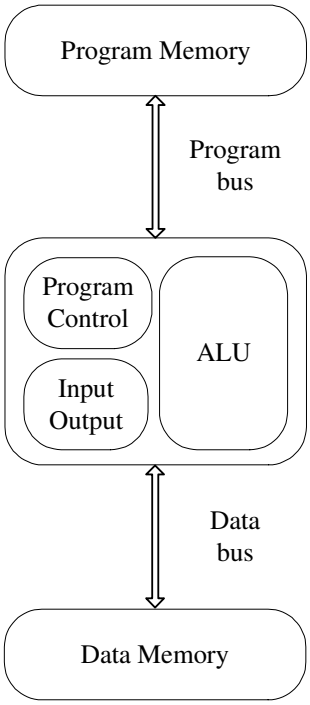

Figure 3.2.1 Harvard memory architecture

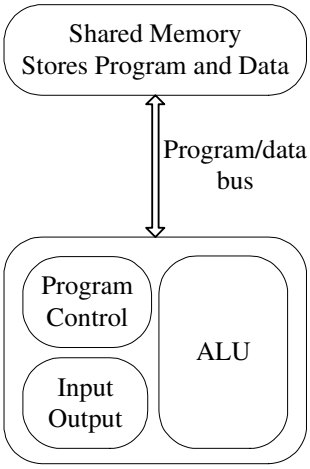

Figure 3.2.2 Von Neumann memory architecture

Commercial microcontrollers have 16, 32 or 64 bit architecture. The Microprogrammed Servo Sequencer has an 18 bit architecture customized for the specific application.

3.3 Microprogrammed Servo Sequencer Architecture

The Microprogrammed Servo Sequencer Architecture is composed by an elaborating Core, programmable ports, 4 fixed module embedded counters for Servo operations and a generic programmable one, managed by COUNTER instruction (Figure 3.3.1). MSS is described in RTL VHDL language. Its code is parametric to easily change registers and instruction fields' length. In the MSS reprogramming phase the Servo designer writes the firmware for registers and signals initialization (signal mapping), and the instructions that have to be executed by the Processor according to the RSS behavior.

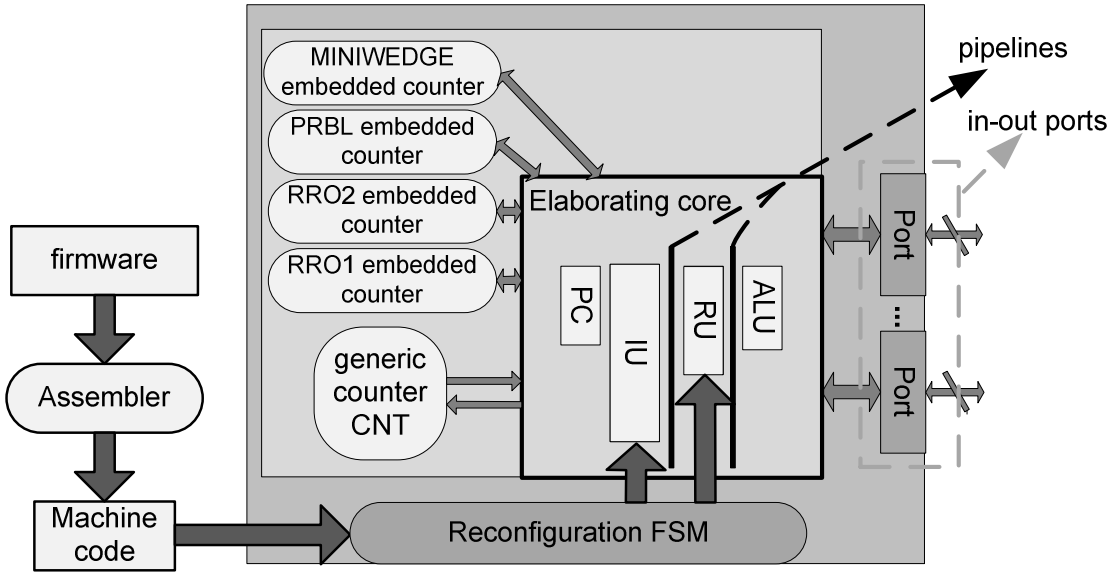

Figure 3.3.1 Microprogrammed Servo Sequencer architecture

The most important characteristic of this Application Specific Instruction-set processor is its I/O ports *programmability*. The MSS has to be used to substitute different Servo subsystem belonging to different Hard Disk Drive system, so the input and output signals may change from Hard Disks to others. By means of firmware reprogramming it is possible to define precise correspondences between hardware and signal labels recording this information in the data memory. This process is called signal mapping and it is explained in 3.4.1.

3.3.1 The elaborating core

The elaborating core (Figure 3.3.2) is based on adapted MIPS architecture: it is composed of a Program Counter (PC), an Instruction Unit (IU), a Register Unit (RU) and an Arithmetic Logic Unit (ALU). The MSS architecture performs the Instruction Fetch through the PC and the IU, the Instruction Decode through the RU and the Execute by the ALU. The Memory Access process is not performed: the Microprogrammed Servo Sequencer

has only Register Unit and Instruction Unit, but not external memory. So that it can have only two pipeline stages in spite of four achieving high performance. The Write-Back process is substituted by the STORE instruction, because it's not necessary to memorize all the executed instructions results.

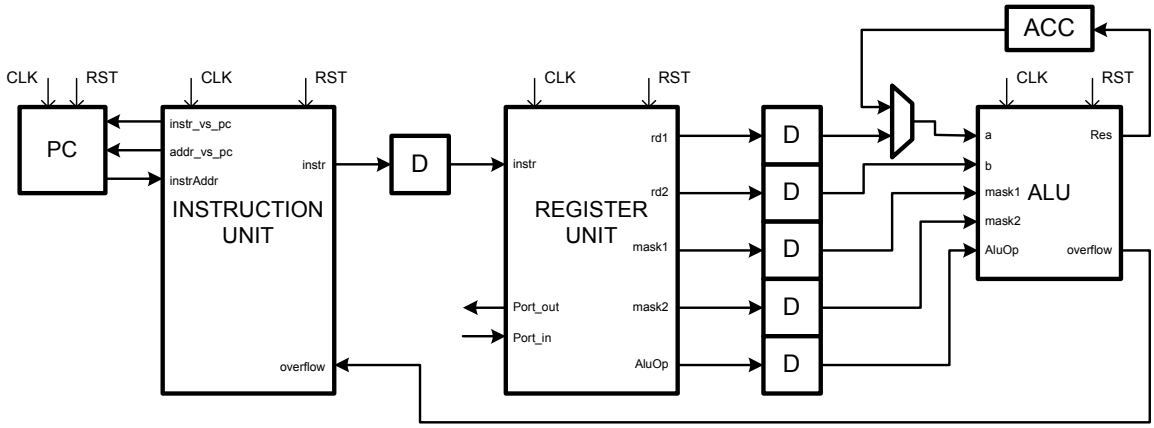

Figure 3.3.2 MSS elaborating core

The PC provides the address *instrAddr* of the instruction that has to be read in the IU; it can be programmed with the number of instructions that are contained in the IU. The instructions are collected in sequence by means of a counter. It is possible also to have an out of order execution caused by J instruction, such as JUMP. The configuration signals *instr_vs_pc* and *addr_vs_pc* manage this operation.

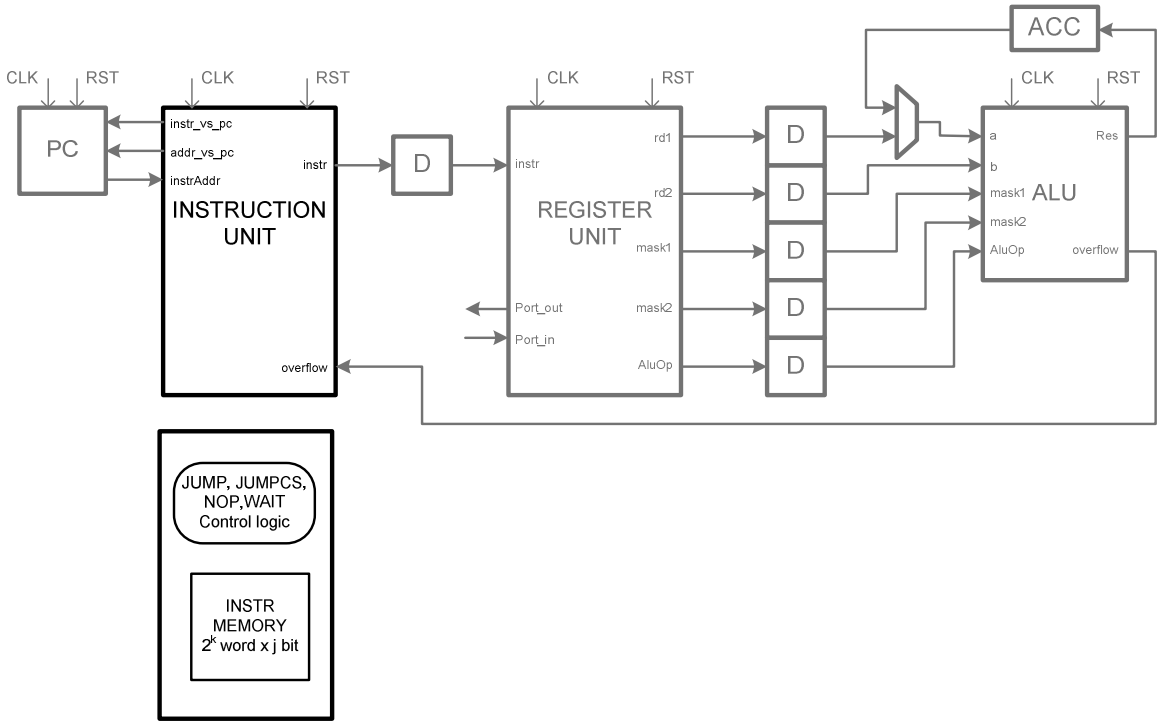

Figure 3.3.3 Instruction Memory details

The Instruction Unit (Figure 3.3.3) contains a RAM single port for the firmware instructions memorization. The input signal *instrAddr* from PC provides the address of the instruction that has to be collected from RAM. The output signal *instr* represents the instruction which has to be executed. PC and IU manage also the control of J instructions through an embedded handshake protocol by means of *instr_vs_pc* and *addr_vs_pc* signals.

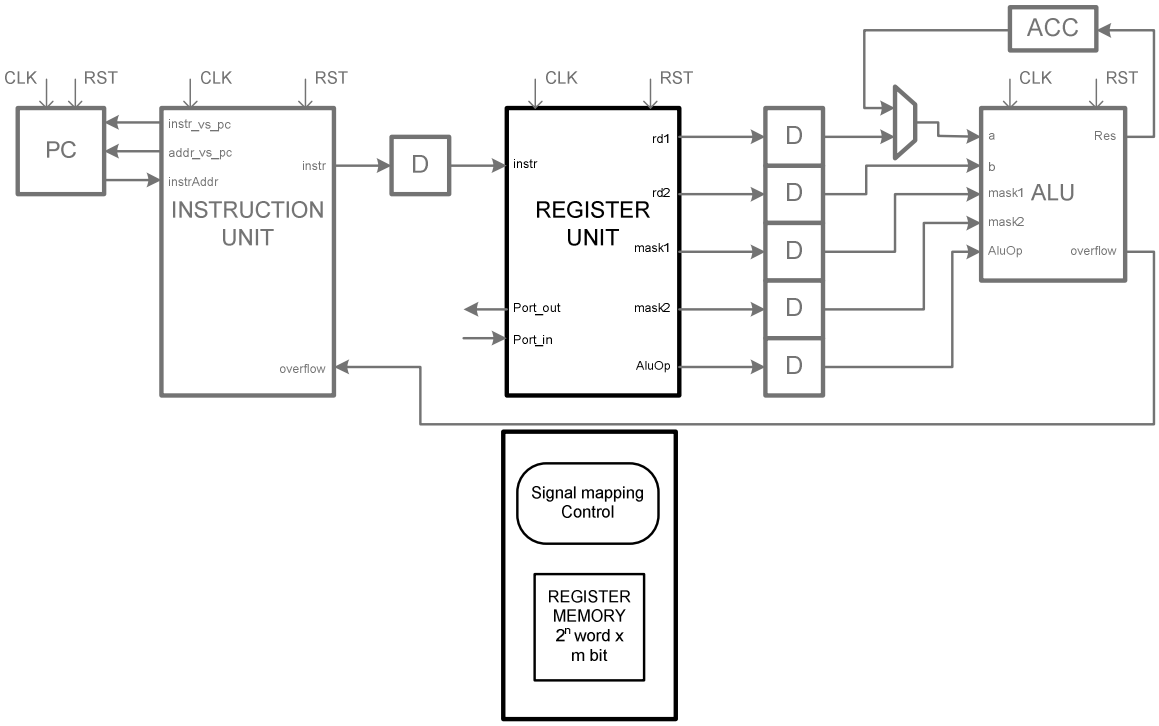

Figure 3.3.4 Register Unit details

The Register Unit (Figure 3.3.4) is composed by a RAM dual port memory (if an R-type instruction has to be executed it is necessary to read two different values from RAM) and a control logic for the management of signal mapping. A unique addressing space for internal memory registers and I/O ports signals have been defined. In Figure 3.3.5.a and Figure 3.3.5.b two addressing space examples are shown: in the Servo application version 1, for example related to a low end market segment Servo, it is necessary to define more information about signal mapping (due to the need of more in/out signals) than in the Servo application version 2, for example related to a high end market segment.

The Arithmetic Logic Unit (ALU) executes logic and arithmetic operations between *a* and *b* coming from RU. If *rd1* and/or *rd2* are related to *rs1* and *rs2* signal labels *mask1* and *mask2* input contain information about the significant bits of these two signals (*rd1* and *rd2* dimension is 12 bit, but a signal dimension can have lower length). Whenever the ALU executes an operation, its result is memorized in the embedded register accumulator *Acc* so that during firmware execution every instruction can access to the previous calculated value. The output signal *ovf* is generated for the communication with RU concerning instruction such as COMPI, ADD and COMPNI.

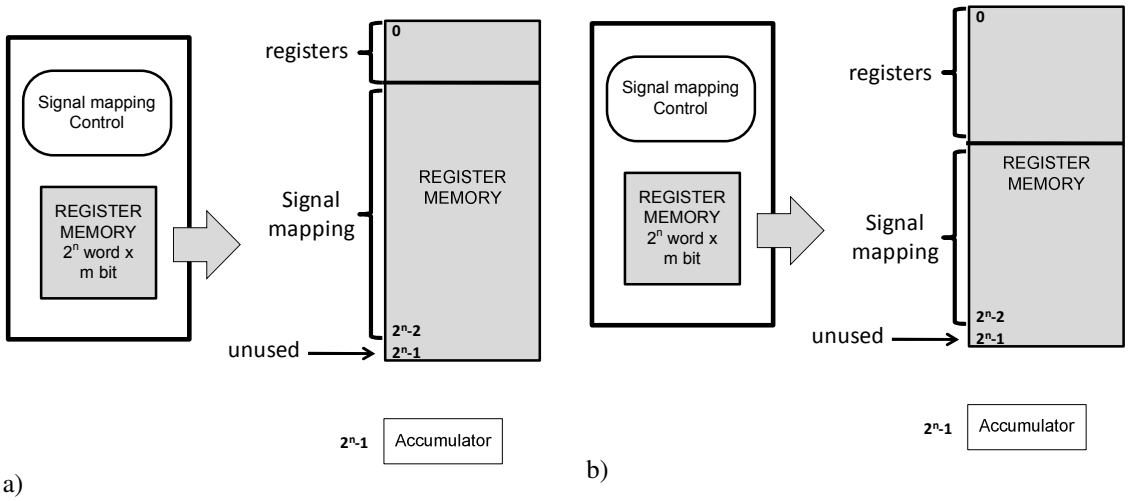

Figure 3.3.5 a) Addressing space Servo version 1 example 1; b) Addressing space Servo version 2 example

3.4 Instruction-Set Architecture

The MSS ISA is derived from the MIPS ISA (Figure 3.4.1) and it has been reduced from more than 64 to 16 instruction types: only the necessary ones are maintained and some of them are modified to customize the architecture for Regular Servo Sequencer emulation. It presents three different kinds of instructions: Register (R), Immediate (I) and Jump (J) instructions (see Figure 3.4.2), but the instructions fields number and length are reduced.

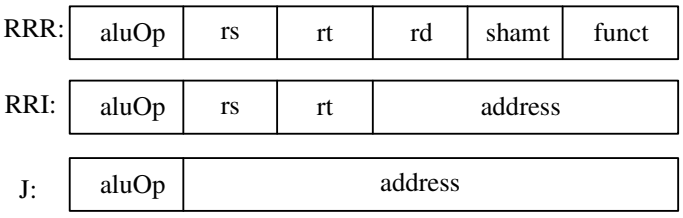

Figure 3.4.1 MIPS instruction-set

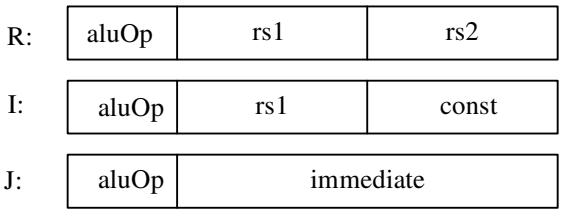

Figure 3.4.2 MSS Instruction-set

In fact there are only three fields (two in J instruction) with instructions codification in the *aluOp* field (Arithmetic Logic Unit operation). This original implementation is due to the need of reduce instruction bit length to preserve block size (and power dissipation). In fact a 4 bit instructions encoding in spite of MIPS ISA 6 bit one is introduced. The MIPS ISA (Figure 3.4.3) instruction length is reduced from 32 bit to a lower length: *rd*, *shamt*, *funct* fields are eliminated because they are not necessary for the dedicated MSS IS; *ad-*

dress and *immediate* fields are reduced because analyzing different Servo applications their firmwares have less than 1 million instructions (20 bits are enough for instructions addressing). In Figure 3.4.4 an example of MSS dimensioning has been shown: the MIPS

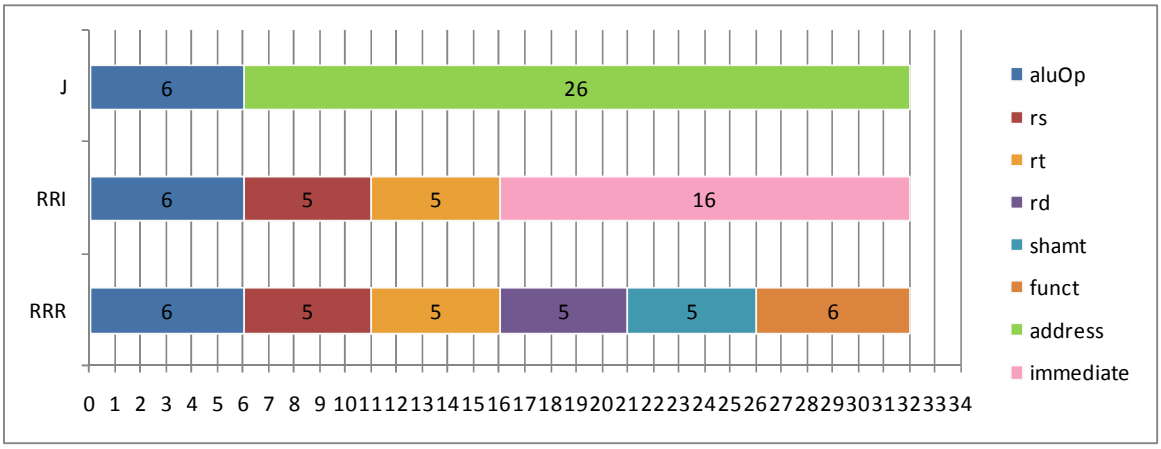

Figure 3.4.3 MIPS Instruction-set

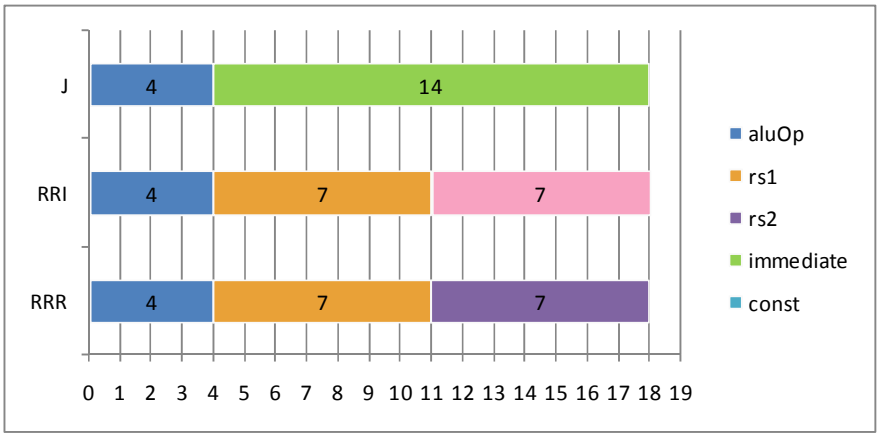

Figure 3.4.4 MSS Instruction-set

Table 3.4.1 Instruction-set architecture

Instruction Type					
R		I		J	
AND	*rs1&*rs2	COMPI	If *rs1=const -> ovf = 2	JUMP	branch un-conditioned
OR	*rs1|*rs2	WAIT	ctrl SG, HALT	NOP	No op
ADD	*rs1+*rs2	COMPNI	If *rs1!=const -> ovf = 2	JUMPCS	branch condi-tioned
CONCAT	*rs1&*rs2	SLLR	*rs1<<imm		
COUNTER	generic counter	STORE	*rd=acc		
SET	*rs1=sig	SET_V	*rd=const		
NOT	Not (rs1)				

ISA instruction length is reduced from 32 bit to 18 bit length [3.2]. The Instruction Set definition is derived from the behavior of the Regular Servo Sequencer: the firmware program based on the MSS IS substitutes the RSS operations. Each RSS state manages control signals that drive embedded control system blocks, like detectors, for the recognition of Servo pattern. The Microprogrammed Servo Sequencer firmware describes this state diagram behavior.

In Table 3.4.1 the R, I and J instructions have been shown in details. The R instructions manage two operands *rs1* and *rs2* which can be two registers, two signals or a register and a signal. It is possible to have also the accumulator *Acc* in spite of *rs1* or *rs2*. These instructions are:

- AND operates the "and" bitwise of two values:
  - *rs1* and *rs2* registers from data memory,
  - *rs1* and *rs2* input signals,
  - *rs1* register and *rs2* input signal or vice versa,
  - *rs1* register or input signal and *Acc* accumulator or vice versa;
- OR operates the "or" bitwise of two values (the same types of AND instruction);
- ADD operates the sum operation of two values (the same types of the above instruction);
- CONCAT operates the concatenation of two values: *rs1* and *rs2* input signals. The operation is correct only if the sum of the two signals bit length is smaller or equal than the registers bit length;
- SET initializes *rs1* with *rs2* value: *rs1* must be an output signal and *rs2* could be an input signal, a register or the accumulator *Acc*;
- COUNTER is a customized instruction: it manages an embedded programmable counter. The details of this instruction are described in paragraph 3.4.3.

The I instructions include the operand address, *rs1*, which can be a register or a signal, and a constant, *const*. They are:

- NOT operates the negation of *rs1* value: *rs1* can be an input signal, a register or the accumulator *Acc*;
- COMPI operates the equality test on *rs1* value and *const* constant: *rs1* can be a register, an input signal or the accumulator *Acc*;
- COMPNI operates the inequality test on *rs1* value and *const* constant (the operand *rs1* is of the same types of the COMPI instruction);
- SLLR operates the shift logical left of *rs1* value of *const* positive constant position, if *const* is a negative value SLLR operates the shift logical right (the operand *rs1* is of the same types of the above instruction); SLLR substitutes the instructions Shift Logical Left (SLL) and Shift Logical Right (SRL) to reduce the instruction types number, so maintaining *aluOp* field of 4 bit wide without any instruction length modification;

- STORE saves the result of the previous instruction at the address *rs1* (write back operation): *rs1* could be a register or an output signal;
- SET_V initializes *rs1* with *const* value: *rs1* must be an output signal;
- WAIT makes the ASIP idle until *rs1* assumes the value indicated by *const* value. The WAIT instruction has also a particular configuration for the activation of the Servo Gate control; this operation is described in 3.4.4.

The J instructions are the jump instructions: the *immediate* field contains the destination address. They are:

- NOP is a no operation instruction: the *immediate* field is the iterations number;
- JUMP is an unconditioned branch: after a JUMP the next instruction to be executed is at the *immediate* address in the instruction memory;
- JUMPCS is a conditioned branch that is realized through the series of two instructions: the first provides the overflow information and the second is the JUMPCS (Figure 3.4.5). The ADD, COMPI and COMPNI instructions provide the overflow information by means of the *ovf* signal. The Jump operation is executed only if the *ovf* signal, generated by the instruction associated, is at '10'. This kind of approach allows JUMPCS to be combined with different kind of test: not only an equal test (COMPI and COMPNI), but also, for example, an add overflow test (ADD). The JUMPCS instruction is explained in details in 3.4.2.

```
COMPI   rs1   rs2
JUMPCS  imm
```

Figure 3.4.5 JUMPCS example

The effect of each instruction may be memorized through STORE operation that substitutes the write back phase typical of commercial microcontrollers. This causes an overhead latency that it is accepted because few firmware instructions need the write back process for this specific application.

3.4.1 Signal Mapping

The signal mapping provides a correspondence between each firmware label and one or more signals. Each label represents the port address and mask information of the respective signal. When the MSS has to execute an instruction containing a signal label operand (a firmware label), the RU extracts the correct signal label value masking the port at which the signal/signals is/are connected (Figure 3.4.6). The complex mechanism of signal mapping has been introduced for two reasons:

- Flexibility: it may be that in different firmware versions related to different Servo systems some signal label may change their dimension or significance. Signal mapping provides a way to have different interpretations of signals la-

bels with a complete disconnection between signals connected to ports and signals labels in firmware. In Figure 3.4.6 *sig_a* label has different interpretations in each Servo firmware versions.

- Power consumption reduction: the signal mapping mechanism lead to consider only significant information that are necessary for a particular Servo system, not all signals connected to ports. So the ALU operations are executed on limited values, only the necessary ones (Figure 3.4.6).

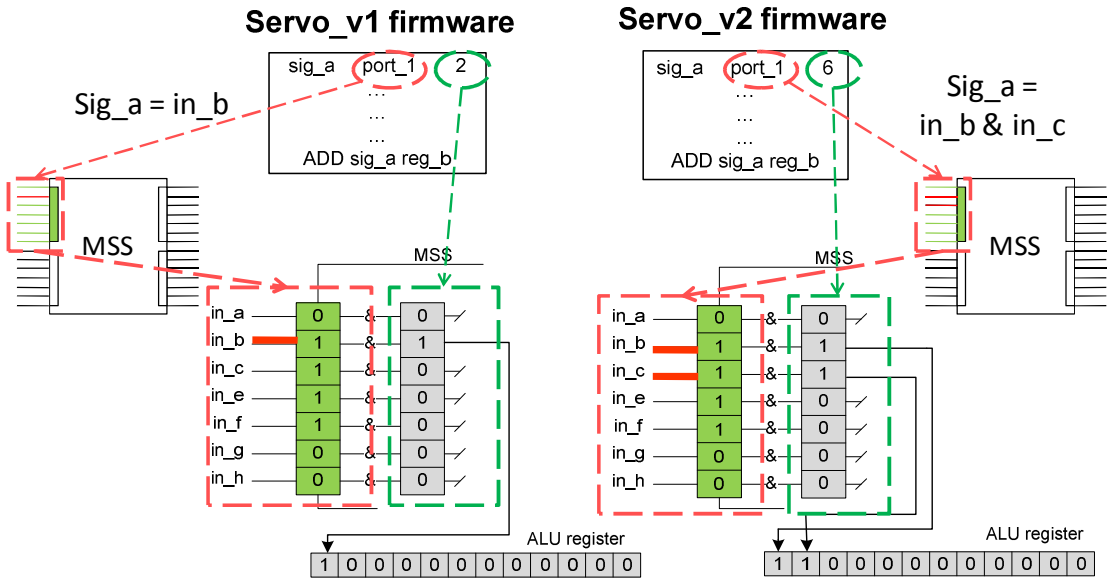

Figure 3.4.6 Signal Mapping mechanism example

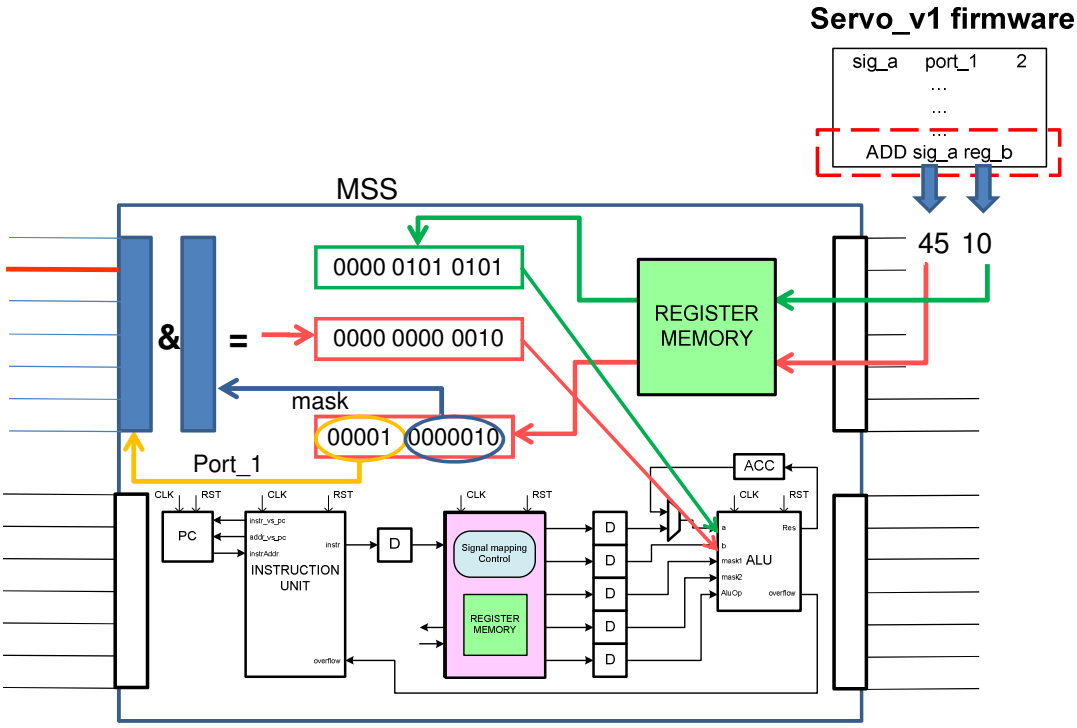

Figure 3.4.7 Signal Mapping instruction execution example

When the Microprogrammed Servo Sequencer has to execute an instruction containing a signal operand, the Register Unit:

1. extracts from data memory the signal port and mask;
2. applies the signal mask to the correspondent port value;
3. send the obtained value with the signal mask to the ALU.

Then the ALU executes the operations described in the RSS ISA. In Figure 3.4.7 an example is shown: the RU executes an ADD instruction with register *reg_b* and signal *sig_a* operand. *Sig_a* value is obtained masking the port number one.

3.4.2 JUMPCS instruction

The JUMPCS is part of a conditioned branch that is realized through the sequence of two instructions: the first one provides the overflow information and the second is the JUMPCS. The Jump operation is executed only if the *ovf* signal, generated by the associated instruction, is '10': ('00' indicates no information, the ASIP waits; '01' stays for false, the Jump is not executed). This kind of approach allows JUMPCS to be combined with different kind of test: not only have an equal test like COMPI and COMPNI, but also, for example, an add overflow test (ADD) and the Servo Gate control (*WAIT SG 0*).

Due to the two microcontroller pipeline stages, one RAM dual port register memory latency and one extra clock cycle latency for reading instruction in the instruction

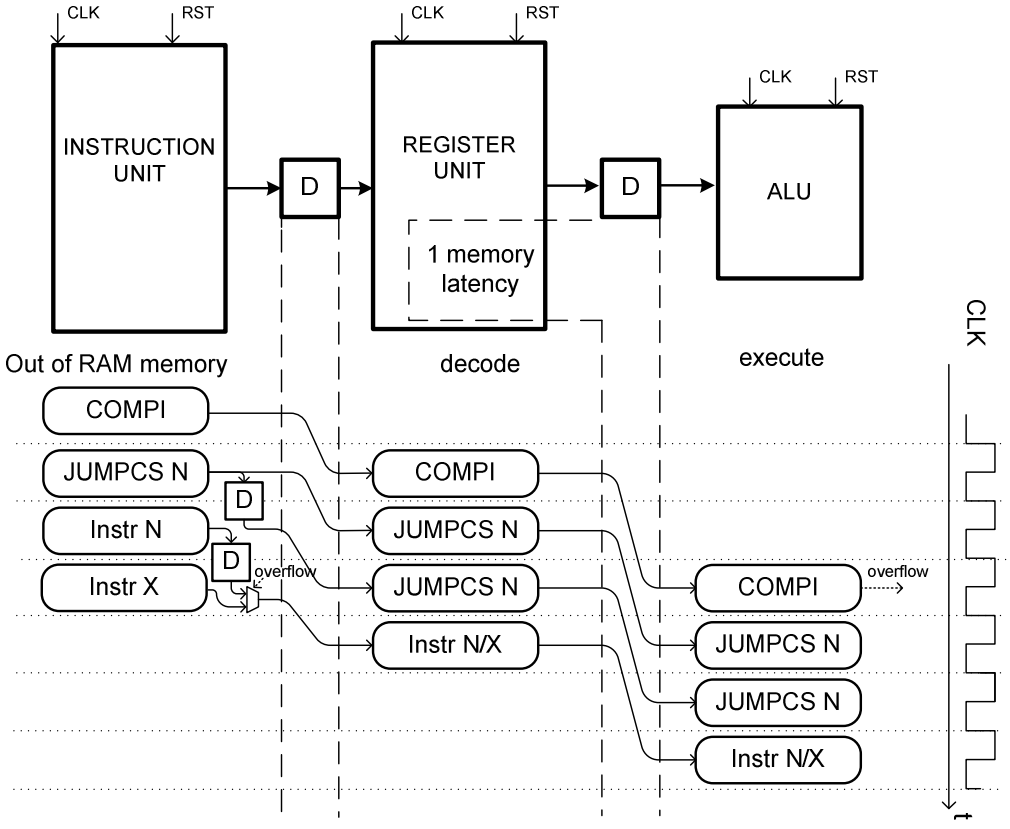

Figure 3.4.8 JUMPCS execution

memory it isn't possible to collect from instruction memory the correct instruction that has to be executed after JUMPCS without delay. To achieve a better performance the prefetch technique has been introduced: the instruction corresponding to the jump destination is anyway read from the instruction memory during the first cycle of the JUMPCS fetch; in the second clock cycle the jump destination is memorized in a register, while the instruction following JUMPCS in the firmware code is read. In the third clock cycle the correct one, chosen between the two prefetched instructions, is executed. This technique allows the reduction of the JUMPCS latency from 3 to 2 clock cycles. Since the MSS ASIP requires many JUMPCS in the firmware, the latency reduction cuts down significantly the total firmware execution time.

In Figure 3.4.8 it is shown an example of JUMPCS execution where the *ovf* signal is generated by a COMPI instruction. This signal is ready after 2 clock cycles after JUMPCS decoding and during COMPI execution.

3.4.3 Generic Counter

COUNTER instruction manages the programmable counter *CNT*. In Figure 3.4.9 it is shown the COUNTER configuration for *CNT* initialization: COUNTER CNT reg1 sets generic counter to *reg1* value, whereas instruction COUNTER CNT 126 starts the decrement of the counter with *end_CNT* signal value announcing the end of this operation. COUNTER is a background operation: the firmware execution continues with the instructions following COUNTER. The end of the counter decrement must be verified with a WAIT instruction on the *end_CNT* signal.

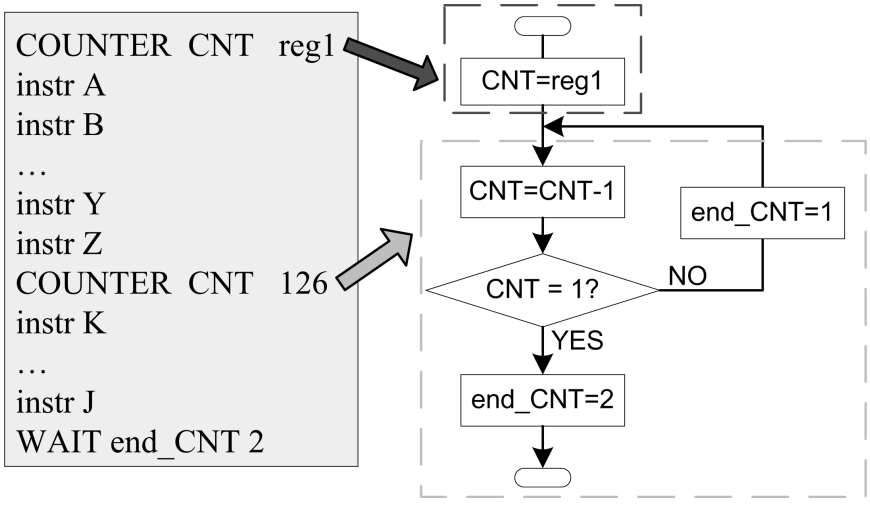

Figure 3.4.9 COUNTER instruction

3.4.4 Servo Gate control

The WAIT instruction has a particular configuration for the activation of the Servo Gate control: if the first field, *sig1*, assumes the special value (SG), a dedicated address is gen-

erated to recognize the Servo Gate control operation. The Servo Gate is an input enable signal whose value must be periodically verified to correctly activate or terminate the Servo operations: a custom logic, designed to achieve better performance, is activated to perform this operation through the WAIT instruction. In Figure 3.4.10 the Servo Gate WAIT configuration is shown: the *WAIT SG 0* instruction activates the Servo Gate control and the associated JUMPCS instruction waits for the positive or negative conclusion of this operation. If this control has no success the firmware execution jumps to *failure_state*.

```
WAIT      SG      0
JUMPCS    failure_state
```

Figure 3.4.10 Servo Gate WAIT

3.4.5 Embedded counters

The MINIWEDGE, PRBL, RRO1 and RRO2 counters are fixed module embedded counters that communicates with the MSS by means of internal embedded signals. These four counters cannot be described by means of the other generic counter instruction because their activations involve too much input signals, so the generic counter approach would have penalized MSS performances.

3.5 Reprogramming process

The Reprogramming process achieves the MSS behavior modification to fit different market Segment Servo Sequencer characteristic. This process is executed only during

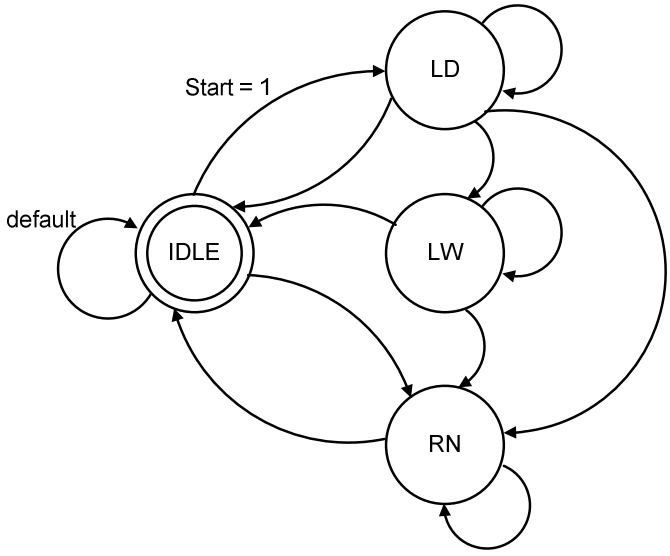

Figure 3.5.1 Reprogramming Finite State Machine

each Hard Disk Drive system bootstrap: Regular Servo Sequencer industrial implementation has to be initialized by means of configuration information contained in a memory on hard disk drive channel chip board; in this memory can be loaded also the firmware data to allow the Microprogrammed Servo Sequencer reprogramming.

The reprogramming process has been implemented by means of a Finite State Machine (Figure 3.5.1) that monitors the Microprogrammed Servo Sequencer states. This process (Figure 3.5.2) is based on an object code file, the Machine Code (MC), to be read during this phase. The MC contains information about signal mapping, register memory and instruction memory and it is generated by a custom assembler written in C language.

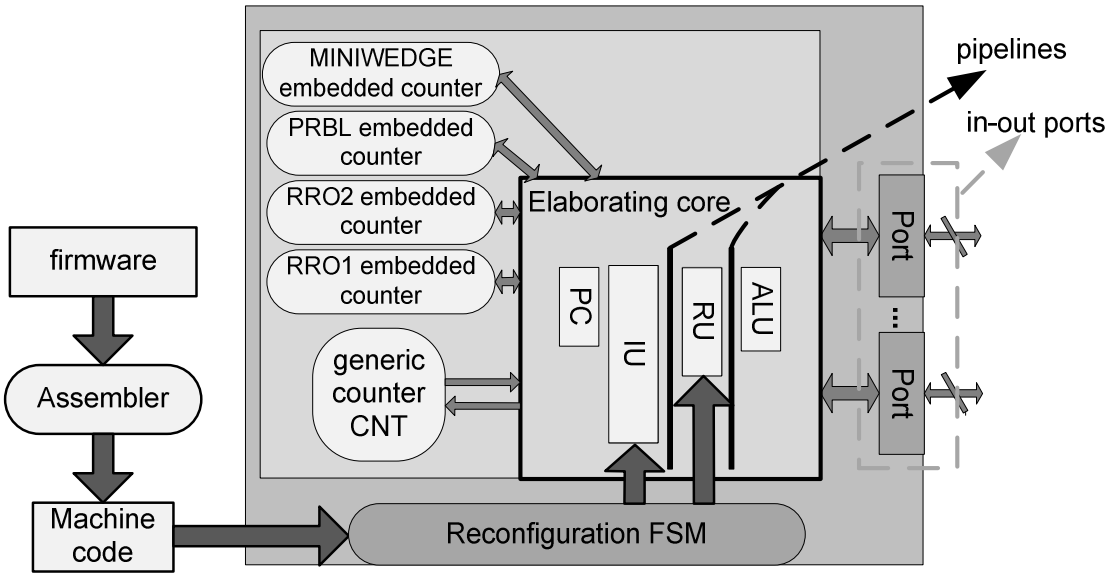

Figure 3.5.2 Reprogramming process

The firmware, an assembly program, is written by the Servo designer and it is based on the MSS ISA. The firmware consists of three sections (Figure 3.5.3 and Figure 3.5.4):

1. Initialization of registers: the registers are initialized with an hexadecimal value and a label is associated to each of them;
2. Initialization of signals: it is defined each signal mapping by associating to the signal label its port and mask;
3. Program: it describes the behavior of the microcontroller, in particular the part of the firmware code showing the sequence of operations that must be executed using the instructions available from MSS ISA. In this part of the assembly code it is possible to refer to registers and signals using the labels associated to them during previous initializations. Other labels may be used for some code lines, so these labels may be used for jump instructions for example.

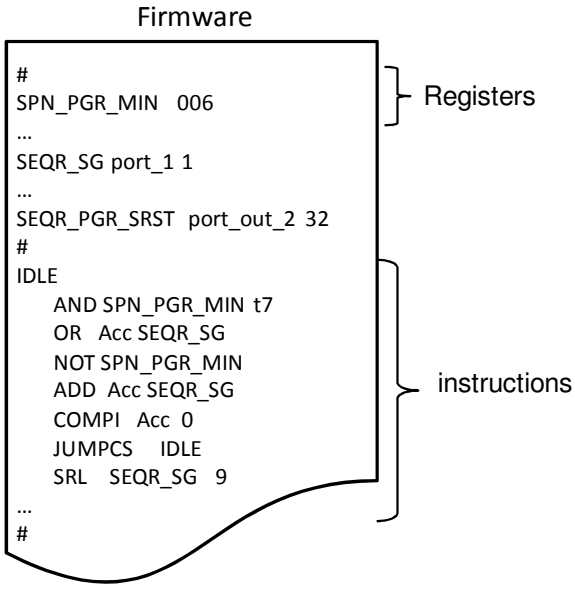

Figure 3.5.3 Firmware

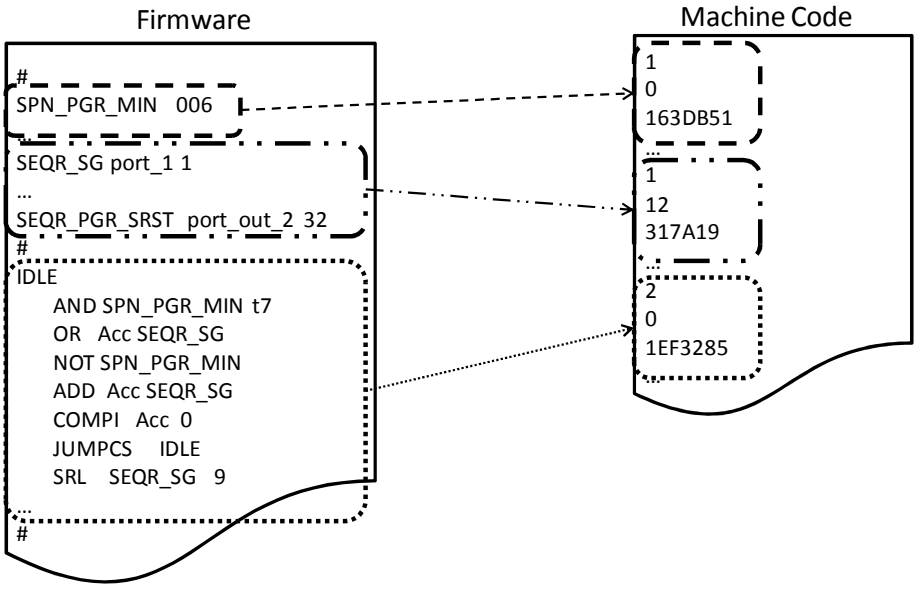

Figure 3.5.4 MSS Firmware and Machine Code: the MC I expressed in decimal coding

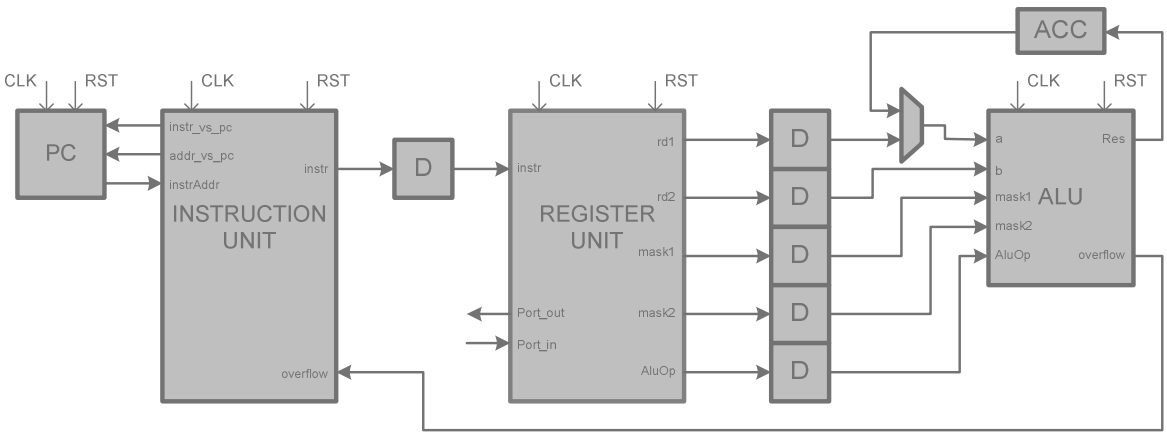

Figure 3.5.5 IDLE reprogramming phase: the MSS is inactive

At system startup the MSS is in the *idle* state (Figure 3.5.5) and the reprogramming process is triggered from the *start* control signal.

Figure 3.5.6 shows the *Loading Data* (LD) state: the register memory is initialized with register values and signal mapping (ports and masks corresponding to signal labels). The instruction memory is booted in the *Loading Word* (LW) state (Figure 3.5.7).

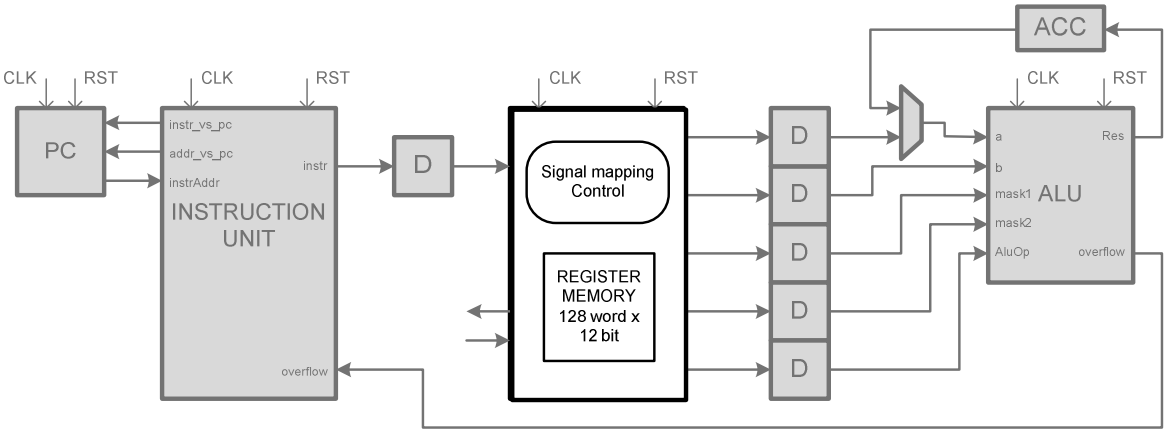

Figure 3.5.6 LD reprogramming phase

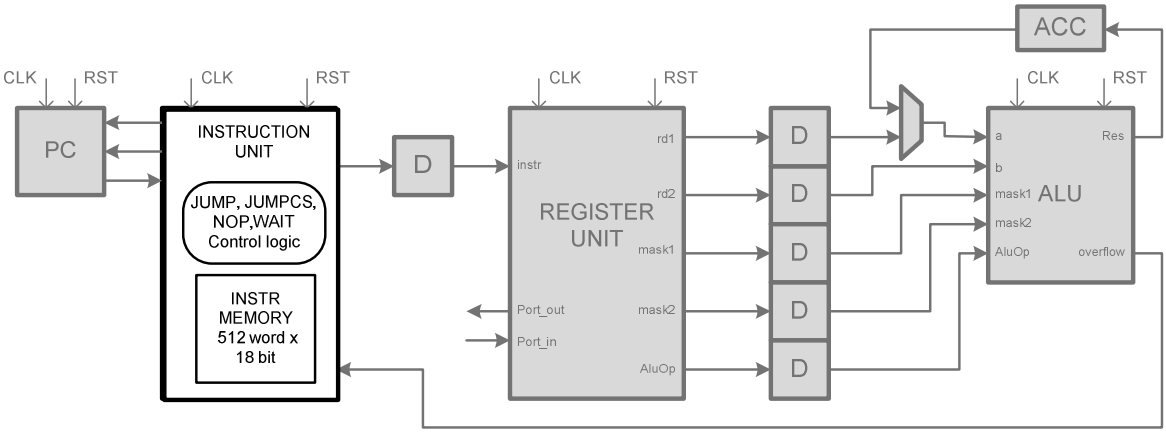

Figure 3.5.7 LW reprogramming phase

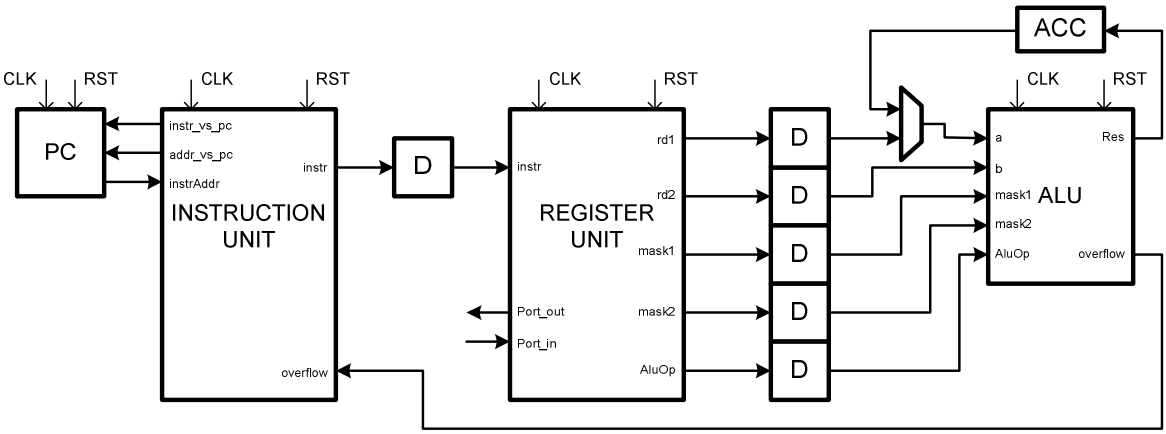

Figure 3.5.8 RN reprogramming phase: firmware execution

In the *RuNning* (RN) state the Microprogrammed Servo sequencer executes the firmware that describes the behavior of the Regular Servo Sequencer (Figure 3.5.8). When a head focus process is started and a Servo Sector must be read, the MSS is in *RN* state and executes the firmware; at the end of the elaboration the MSS is set back in *idle* state until the next head focus process.

3.6 Bibliography

[3.1] D.A. Patterson, and J. Hennessy, “Computer Organization and Design”, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2004.

[3.2] P. Baldrighi, M.M. Maggi, M. Castellano, C. Vacchi, D. Crespi, P. Bonifacino, **“**Implementation of Microprogrammed Hard Disk Drive Servo Sequencer”, *Proc. 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools,* IEEE, pp. 442 – 446, Sep. 2008.

4 The verification process

The CMOS scaling down process has generated multi-million gate ASICs with necessary circuit complexity increasing. In the last few years SOCs and SOPs, which consist of a complex system composed of different blocks such as elaboration cores, memories, communication modules integrated in a single chip, are often adopted. These systems have better performances and reduced chip area than simpler ASICs, but the main disadvantage is the complexity of these systems: during RTL design the number of functional bugs increases exponentially with system complexity, so the process of verification becomes essential to reduce bugs number before the manufacturing process.

4.1 The importance of verification

In Figure 4.1.1 the digital Design and Manufacturing flow has been shown. The first phase consist of chip specs definition; starting from design specs the RTL description is generated; after a first verification process the Synthesis of RTL creates the netlist ready for the prototype phase; another verification process to check RTL source code and netlist correspondence is needed before realizing prototype; the prototype must be tested to control chip features correctness; if the test is successful the manufacturing process goes on and product chips are realized; before selling them a final test is needed for rejecting those that don't work. Every verification process is needed for the bugs elimination. If the bugs are resolved before prototype phase chip cost doesn't grow too much, but if some bugs are found during prototyping or manufacturing chip costs grows exponentially (Figure 4.1.2). That's why the verification process is so important. Moreover the increasing complexity of chip leads to an increasing number of functional bugs, so that the verification process is essential [4.1], [4.2].

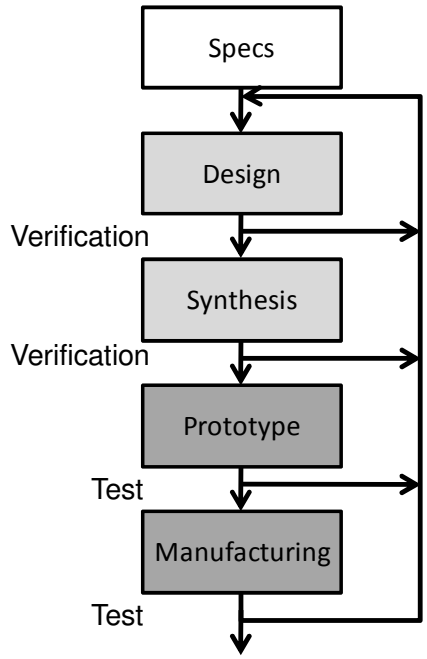

Figure 4.1.1 Design and Manufacturing Flow

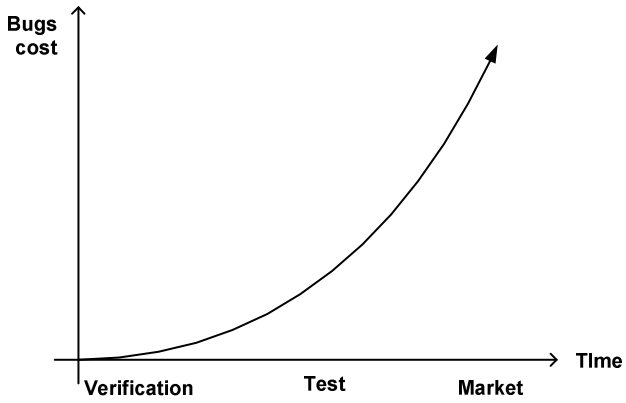

Figure 4.1.2 Chip cost

During chip design and manufacturing processes it is necessary to consider not only that specs have to be met, but also three different factors, which can determine the chip success or failure [4.3]:

a) Time to market: chip success depends on how much time elapses from chip specs definition to product availability on market: the longer is this period the fewer will be chip revenue. In fact it is important that the chip becomes a

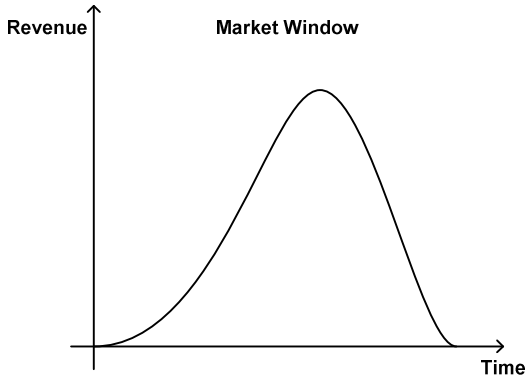

Figure 4.1.3 Market Window Revenue

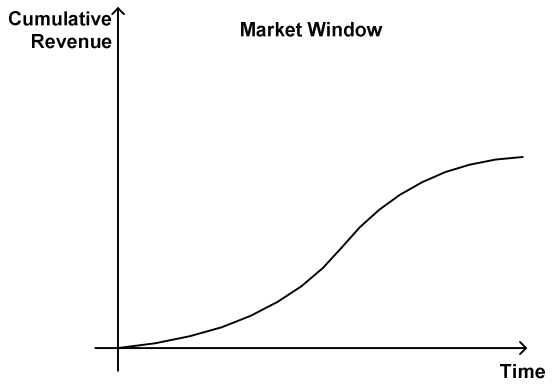

Figure 4.1.4 Market Window cumulative revenue

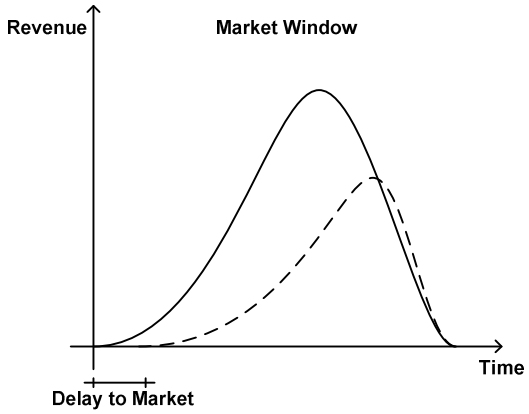

Figure 4.1.5 Loss of Revenue due to Delay to Market

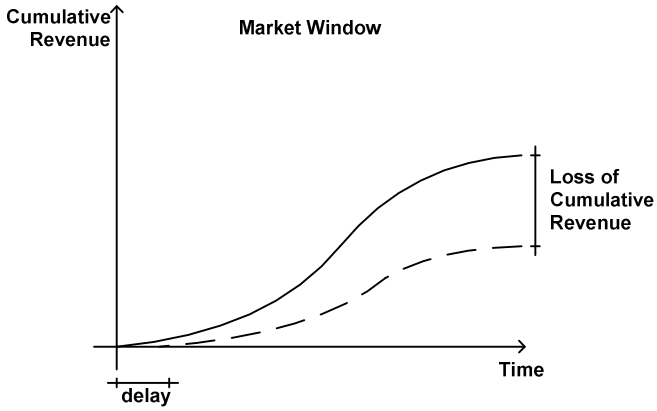

Figure 4.1.6 Loss of cumulative Revenue due to delay to market

product during the *Market Window*, a temporal window in which there is this product demand (Figure 4.1.3 and Figure 4.1.4). If the product delay to Market increases, the sales decrease a lot causing fewer revenue (Figure 4.1.5 and Figure 4.1.6). The verification process is able to find more quickly design bugs for the Delay to Market reduction.

b) Costs: the most important focus for a microelectronic society is to have the highest revenue by means of reducing design and manufacturing costs. With the CMOS scaling down process for the realization of integrated circuit the transistor dimension reduction allows to integrate more transistors in one chip increasing chip density. This phenomenon leads to the chip cost reduction considering product high volume, but the scaling down process causes also an increasing of masks cost for the integrated circuit manufacturing. If some bugs are found during chip test process, they have to be resolved, making necessary modifications, doing a review of the entire circuit with chip respin. In the last years respins due to functional bugs are increased (Figure 4.1.7). This aspect together to the increasing masks cost lead to the necessary verification process introduction and improvement. The goal of the verification process is to reverse

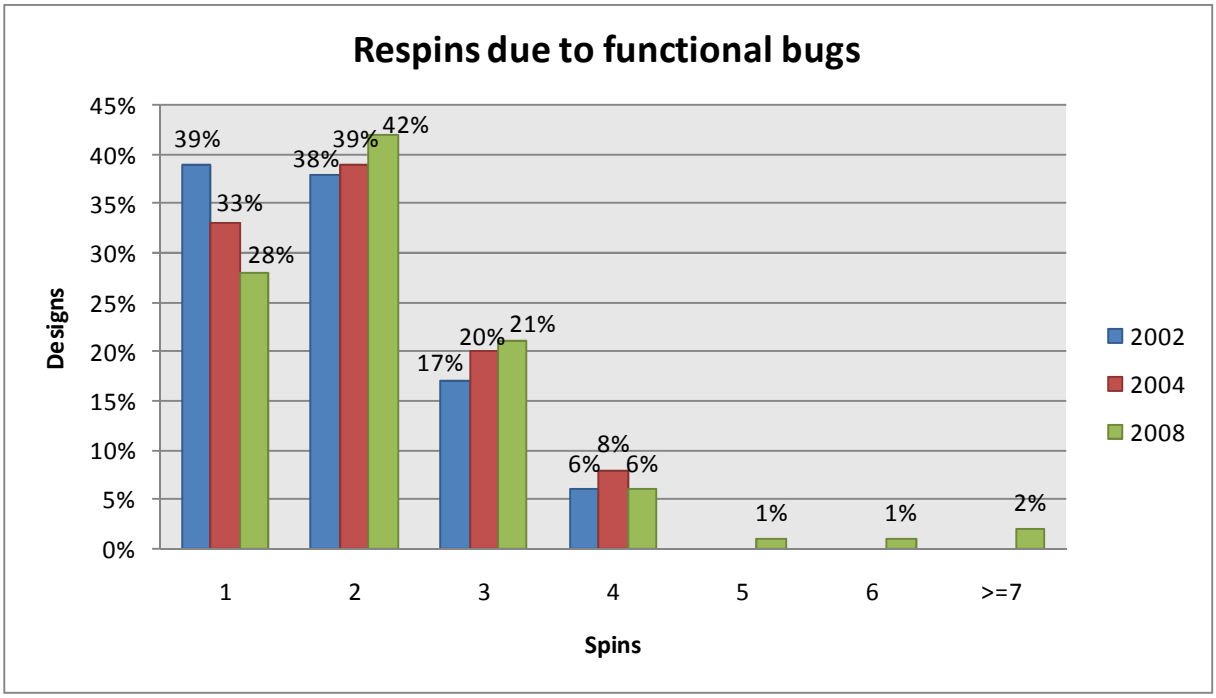

Figure 4.1.7 Respins due to functional bugs [4.4]

chip respins tendency resolving functional bugs before prototyping and manufacturing processes reducing Non-Recurring Engineering (NRE).

a) Quality: it is important to realize bug free chips. The bug not free chips diffusion in the market could cause other than a loss of customer satisfaction also a warranty cost increasing. In the worst cases this bug not free chips lead to the damage of the corporate image till customers loss.

In the last years Design for Verification has became an important aspect of the entire process of Design, Prototype and Manufacturing: a designer may think about verification during circuit design to facilitate the verification process increasing the probability of finding more functional bugs in the first phase.

The functional verification controls design correctness before its manufacturing. Relating to digital circuits this kind of verification consists of two technology kinds:

- Static technology: generally it is called formal verification. It consists of analyzing RTL code to find bugs such as, for example, unreachable lines of code. It controls the syntactical aspect of RTL source code.
- Dynamic technology: it is called simulation-based or emulation-based verification. This kind of approach requires not only the source code, but also the testbench, that is an example of a possible environment in which the design will work. This verification phase is not automatic: it is necessary to build the entire testbench and the checkers for verifying design correctness.

The process of verification is necessary not only for masks cost saving, but also for the efficiency in findings bugs sources.

Functional verification uses three approaches:

- Black-box approach: in this case the design is treated as a black box. It communicates with verification environment only with input and output pins. So initializing input pins only output pins, that are very few considering all chip signals, can be controlled (Figure 4.1.8). In case of bugs it is very difficult, almost impossible in today's large designs, to locate the sources of the problems. The unique advantage is that this kind of approach does not depend on implementation.

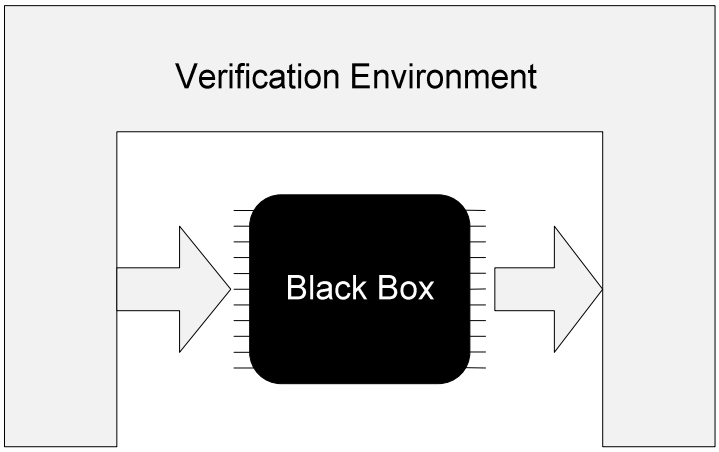

Figure 4.1.8 Black Box approach

- White-Box approach: the design is full observable and controllable, so that internal signals, structure and implementation are visible (Figure 4.1.9). In case of bugs found the problems sources are easily located. The disadvantage of this approach is that any changes in design implementation cause the verification environment modification. It is useful to verify low level implementation specific features. Assertions in RTL code are ideal for this purpose.

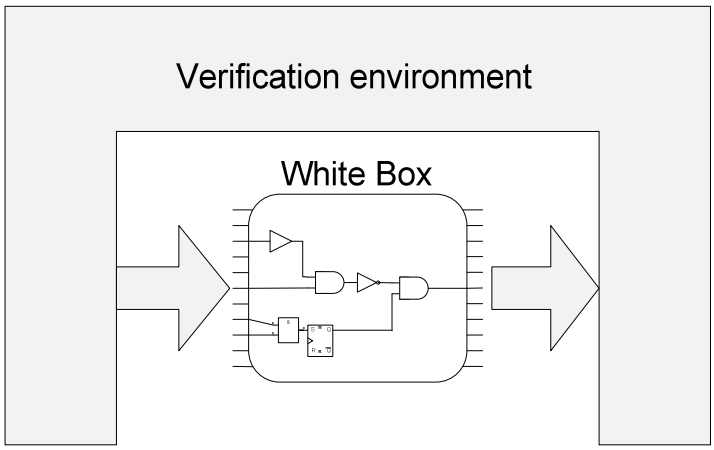

Figure 4.1.9 White Box approach

- Grey-Box approach: some non-functional modification, such as additional registers to control internal states, are added to the design to improve controllability.

In today's large designs the increasing complexity of the circuit lead to a more important verification process able to find greater number of functional bugs so that only white-box and grey-box approach can be used.

The verification process is different from test process (Figure 4.1.10): the first has to find functional bugs in the design; the second is necessary to assure that the product chip corresponds perfectly to the netlist and chip mass manufacturing has a good yield. The increasing verification importance lead to an increasing attention on the verification instrument.

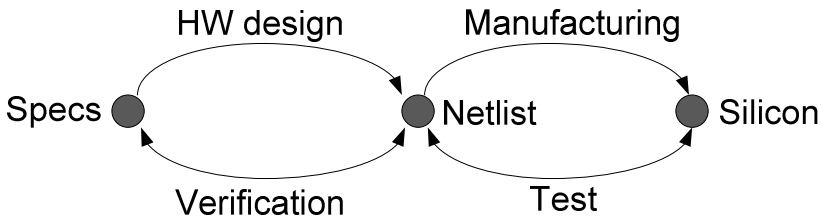

Figure 4.1.10 Verification and test processes

4.2 Assertion based verification

The Dynamic functional verification is based on the Design Under Verification (DUV) behavior control during simulation and emulation. The increasing complexity of today's circuit makes impossible the brute force approach: to stimulate input with all combinations of values verifying all signals behaviors correctness. So that a design is verified with a subset of chip functioning situations.

The ABV is based on assertions, that are particular lines of code inserted in RTL source code. They can describe both low level design functionality (Implementation assertions) both high level characteristic (Specification assertions). The assertions have been used for decades in software implementation, and in the last decade they are being used also in hardware design. This phenomena is due to the simple use of the assertion: it is possible to insert them in the code both during the first design phase both during the last design phase without any changes in RTL source code. The assertions are considered as comments in the netlist generation process so that there isn't any non functionality feature added to silicon. Both designers and verification engineers can use assertions. Generally assertions are inserted gradually to the source code: they are created with new design functionality. There aren't limitations in assertions number: complex designs may have hundreds or thousands of assertions.

A great advantage of assertions use is their reusability characteristic: in every design, prototype and manufacturing phases they can be inserted without any modification (Figure

4.2.1). They can be used both in verification both in testing process to check that design behavior corresponds to design specs.

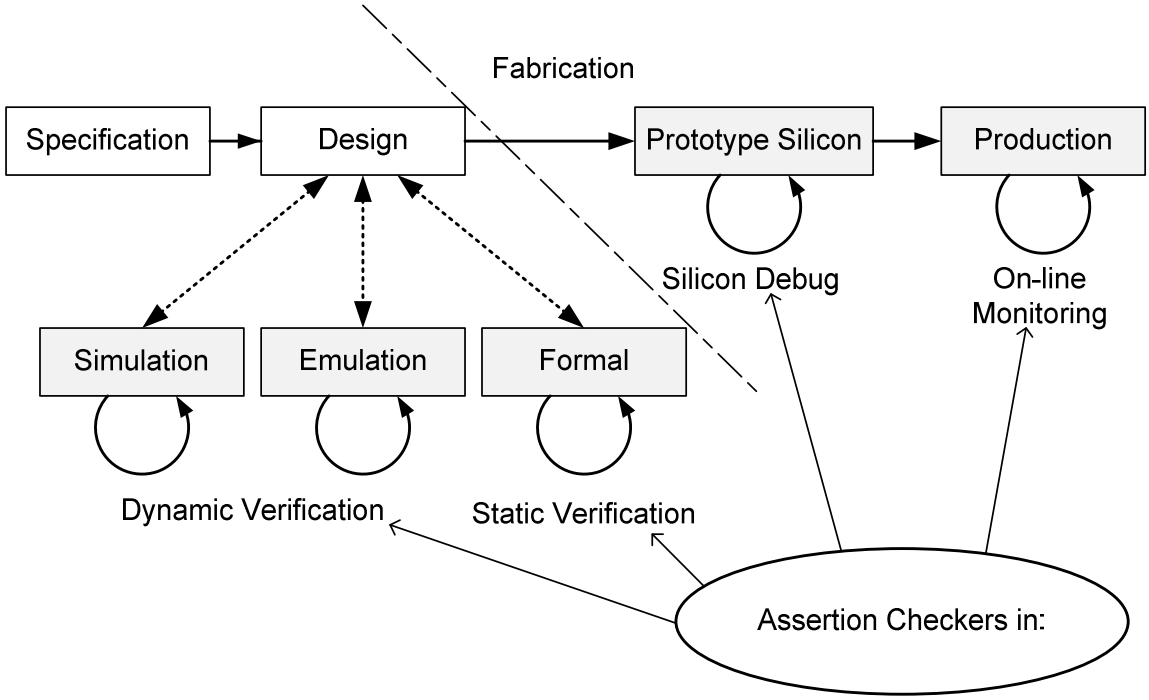

Figure 4.2.1 Design and manufacturing process whit assertion checkers

There are more advantages in assertion use in design process. The first is the design observability increasing: in a classic verification environment a stimuli generator testbench is created; it is applied to the Design Under Verification and then a receiver checks the output to find errors. During simulation to identify a bug by means of a testbench is necessary to generate opportune stimuli for design input so that the bug is stimulated (controllability) and also propagated to design output (observability), see Figure 4.2.2. In some cases it is possible that some bugs are not propagated to output pins causing a lack of verification information (false negative): in Figure 4.2.2 the checker finds only the bug that is propagated to output pin.

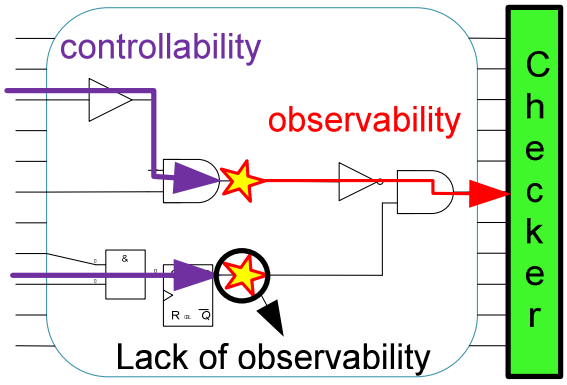

Figure 4.2.2 Not ABV approach

To increment the probability of findings bugs and locate their sources, the easiest way is to move the checkers nearest to the sources: if the checkers are located near bugs sources problems are immediately pointed out (Figure 4.2.3). In this case it is necessary only to care about controllability (input stimuli) so that the bugs are pointed out. Not only it is easier to find bugs, but also assertions add more information about the time the bugs occur and their locations in the source code. Without assertions the testbench is not able to

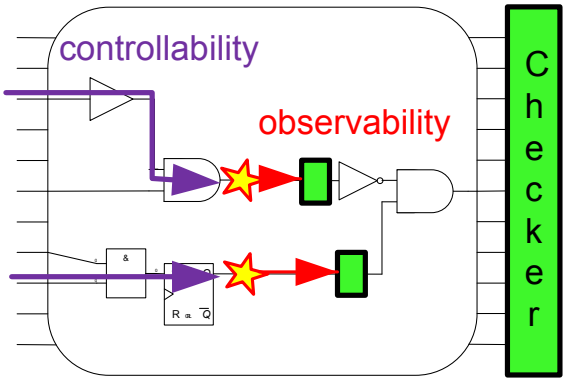

Figure 4.2.3 ABV approach

point out these information, only the stimuli sequence is known, so it is necessary to analyze the source code starting from output pins to reach bugs sources. In today's complex designs this kind of analysis is quite difficult: the design process includes a lot of persons and it may take a lot of time to find bug.

The Specification assertions can be used to check device interfaces. If during the first phase of design there are some discrepancies, the designer notices immediately this communication problem and he/she can resolve it quickly.

The assertions are used both at block level verification both at system level. In the last process there isn't any checkers for each specific block that compose system, but assertions in code continue to control each block behavior.

Assertion Based Verification is an important phase in design process. In literature there are two assertion language: PSL (*Property Specification Language*) [4.5] and SVA (*System Verilog Assertion*) [4.6], [4.7]. Both are IEEE standards with different origins, but they have similar characteristic. For Microprogrammed Servo Sequencer verification PSL assertion (Appendix 9.1) are used.

4.3 Critical operations

Due to the complexity of Microprogrammed Servo Sequencer Assertion Based Verification has been orientated to the more complex MSS functionality. The resulting critical operations are instructions that cause the out-of-order execution of firmware: JUMP, branch uncoditioned; JUMPCS, branch conditioned; NOP, no operation (idle condition);

WAIT, idle condition till a signal value change. Another critical feature of MSS is the Signal Mapping: it must be checked that the MSS reads the exact value for each signal. All these features involve Instruction Memory and Register Unit. In paragraph 4.3.1 will be explained the assertion based verification of JUMPCS instruction as example.

4.3.1 JUMPCS

In paragraph 3.4.2 JUMPCS behavior has been explained, but it is necessary to analyze in details this instruction to explain PSL assertion verification applied to it. When a JUMPCS is executed, during the following two clock cycles the branch instruction (Instr N in Figure 4.3.1) and the not branch instruction (Instr X in Figure 4.3.1) are collected from Instruction Memory so that when the overflow signal (the JUMPCS result) is stable the correct instruction is immediately ready for execution (prefetch technique).

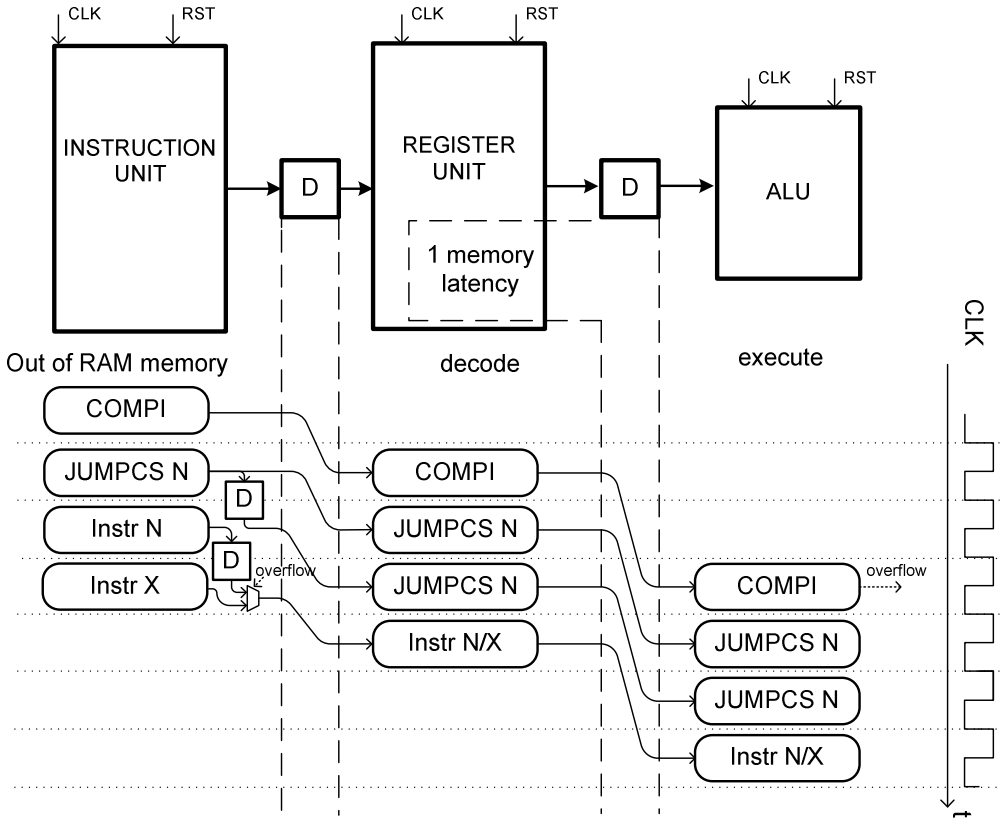

Figure 4.3.1 JUMPCS execution

The firmware example with JUMPCS instruction shown in Figure 4.3.2 is used to explain this instruction in detail: the Instruction Memory signals that are involved in JUMPCS operations are shown in Figure 4.3.3 and Figure 4.3.4. The first shows the JUMPCS branch not execution: *instrAddr* signal is the address information provided by Program Counter; *mem_Addr* signal is the effective address of the instruction collected from Instruction Memory, this value may be equal or different to *instrAddr* in case of out-of-order instructions execution; *data_out* signal is the instruction collected from IM, it may be that this instruction will not go on execution; *instr* indicates the instruction that must

```
...   ...
21    Instr 21
22    JUMPCS 36
23    Instr 23
24    Instr 24
...   ...
36    Instr 36
37    Instr 367
...   ...
```

Figure 4.3.2 Firmware example with JUMPCS instruction

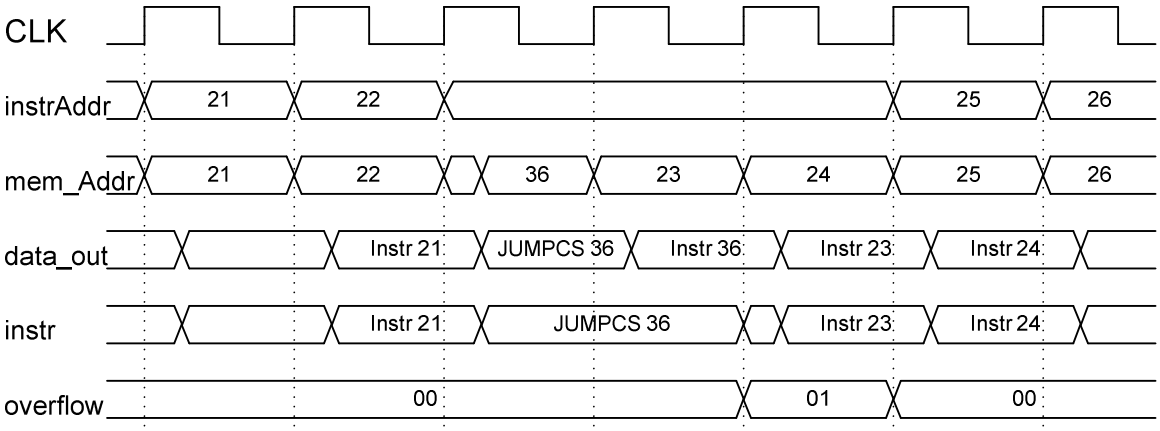

Figure 4.3.3 JUMPCS branch not execution signals evolution

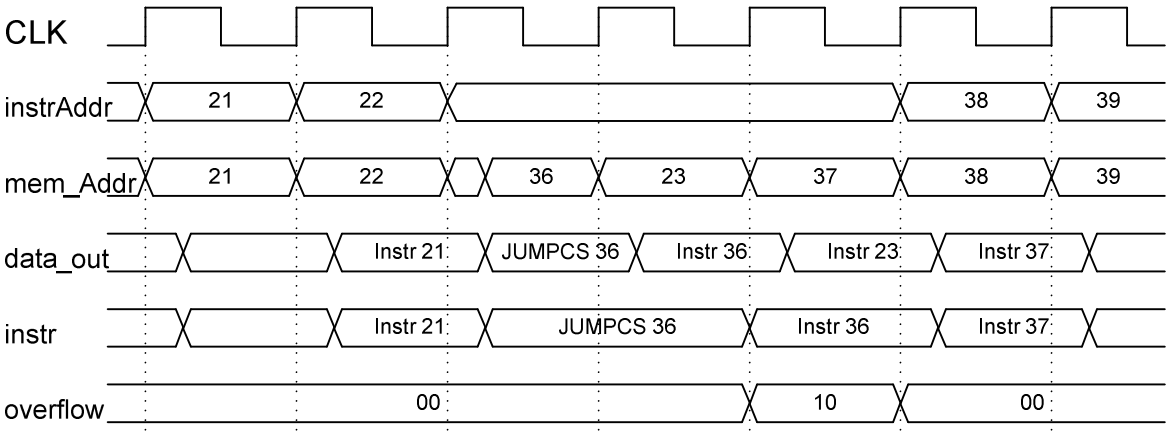

Figure 4.3.4 JUMPCS branch execution signals evolution

be executed; the *overflow* signal is the result of JUMPCS operation; *addr_vs_pc* and *instr_vs_pc* are control signals to update Program Counter configuration.

It is supposed that JUMPCS cannot be repeated two times sequentially (JUMPCS is always associated to the previous instruction). The PSL assertion verification has been realized by means of three different assertions associated to three different properties:

- *p_jumpcs* verifies that when a JUMPCS instruction is in execution in the following clock cycle the instruction in execution is again a JUMPCS, in the next

clock cycle the overflow signal must be at ‘01’(branch not executed) or ‘10’ (branch executed). There is an exception to this behavior if the instruction before a JUMPCS is a WAIT SEQR_SG 0.

```
-- psl Property p_jumpcs is
 -- always {not(instr = NOP0);
 -- rose(instr(li-1 downto li-lc)="0110")} |=>
  -- {stable(instr); (overflow = “01”) or (overflow = “10”)};

-- psl Assert p_jumpcs Severity ERROR;
```

The previous property must be verified independently on the JUMPCS result. The following two PSL sequences are introduced to support the other two properties. They describe JUMPCS behavior in two different cases:

- branch not execution

```
-- psl Sequence s_jumpcs_no is
 -- {instr(li-1 downto li-lc)="0110"[*2]; overflow = “01”};
```

- branch execution

```
-- psl Sequence s_jumpcs_ok is
 -- {instr(li-1 downto li-lc)="0110"[*2]; overflow = “10”};
```

The following PSL property controls that the branch is executed correctly in respect to the *overflow* value.

```
-- psl Property p_jumpcs_no is
 -- always s_jumpcs_no |->
  -- (instr= data_out
  --  or (data_out = WAIT_126_0
  --   and instr = NOP_0))

  -- and ((next(mem_addr, prev(mem_addr,1), nInstr))
   -- or (((instr(li-1 downto li-lc) = "1010")
    -- and (mem_addr = instr(li-lc-4 downto li-lc-limm_j)))))

  -- and instr_vs_pc = 4

  -- and next(prev(mem_addr,1), prev(mem_addr,3), nInstr)

  -- and ((addr_vs_pc = mem_addr+1)
   -- or (addr_vs_pc = mem_addr
    -- and instr(li-1 downto li-lc) = "1011"
    -- and not(instr(li-lc-4 downto li-lc-limm_j)=2)
    -- and not (instr(li-lc-4 downto li-lc-limm_j)=1)));

-- psl Assert p_jumpcs_no Severity ERROR;
```

So the *p_jumpcs_no* verifies that when there is a JUMPCS and the related upcoming overflow is '01':

- the next instruction to be executed is *data_out*, the output of Instruction Memory [blue];
- the next instruction to be read is subsequent to the present instruction (if the present instruction is JUMP the next instruction must be collected at *immediate* address from Memory) [red];
- the present instruction address, *mem_addr*, is subsequent to JUMPCS address [green];
- *instr_vs_pc* must assume the value '4' for the correct configuration of Program Counter [violet];
- *addr_vs_pc* must assume *mem_addr* value incremented of 1 (except in case of NOP *k* instruction, when *k* is different from 1 and 2) [yellow].

The firmware used for JUMPCS branch not execution verification is shown in Figure 4.3.5 and the related PSL assertions verification is shown in Figure 4.3.6.

```
...   ...
6     Instr 6
7     JUMPCS 29
8     Instr 8
9     Instr 9
...   ...
29    Instr 29
...   ...
```

Figure 4.3.5 Firmware used for JUMPCS branch not execution verification

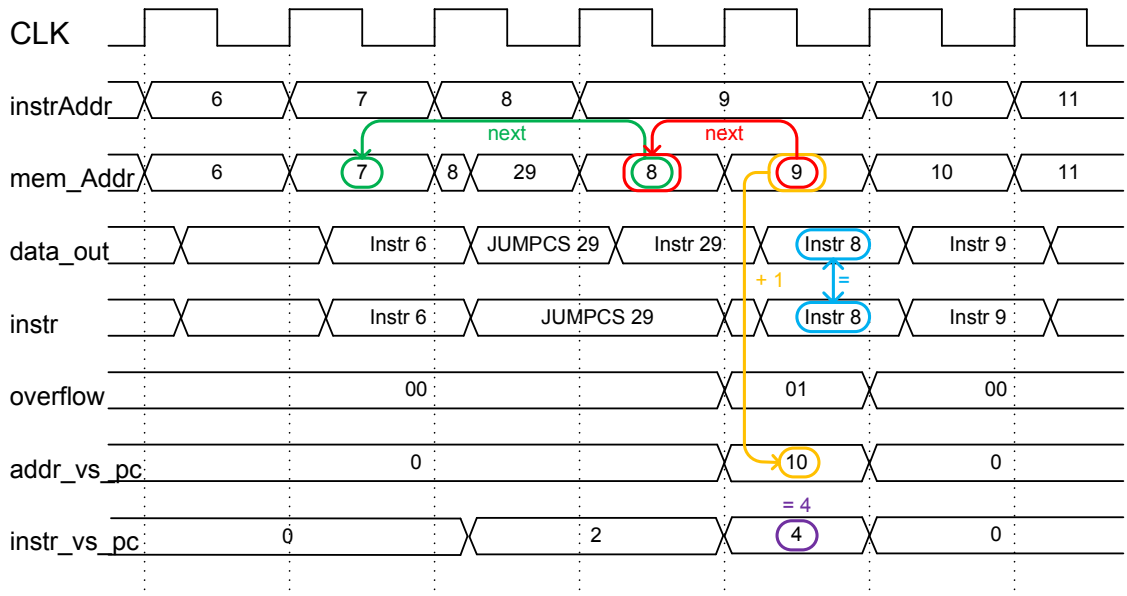

Figure 4.3.6 PSL assertion verification in case of JUMPCS branch not execution

In case of JUMPCS with branch not execution with JUMP as subsequent instruction PSL assertion verifies some different features: the firmware used for this verification case is shown in Figure 4.3.7 and the related PSL assertions verification is shown in Figure 4.3.8.

```
...   ...
6     Instr 6
7     JUMPCS 29
8     JUMP 40
...   ...
29    Instr 29
...   ...
40    Instr 40
...   ...
```

Figure 4.3.7 Firmware used for JUMPCS branch not execution with a subsequent JUMP verification

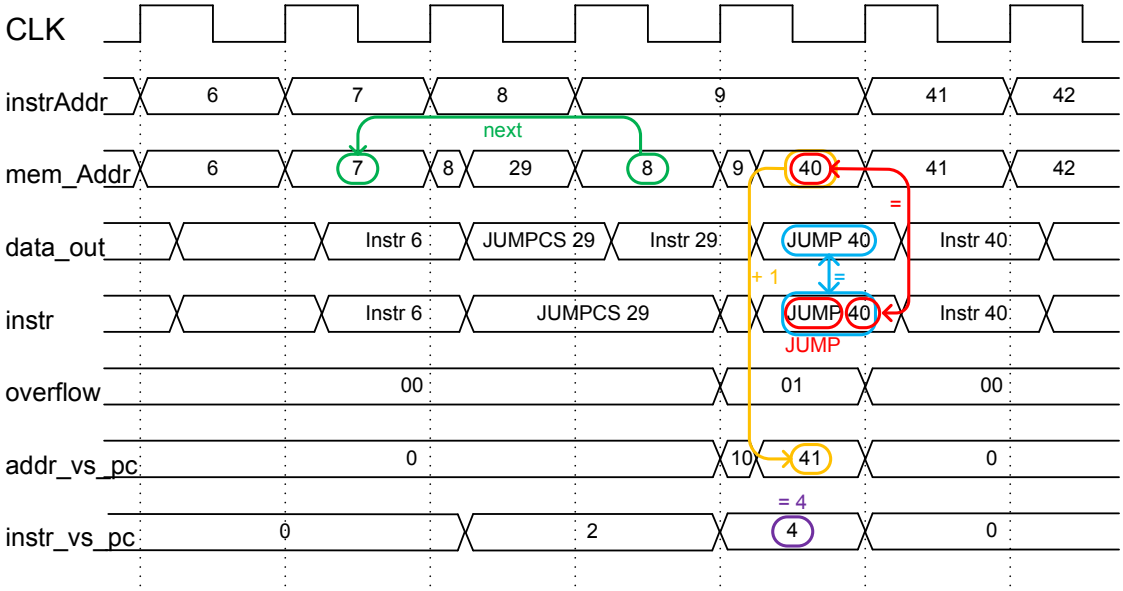

Figure 4.3.8 PSL assertion verification in case of JUMPCS branch not execution with a subsequent JUMP

In case of JUMPCS with branch not execution with NOP as subsequent instruction PSL assertion verifies some different features (Figure 4.3.9 and Figure 4.3.10).

```
...   ...
6     Instr 6
7     JUMPCS 29
8     NOP 3
9     Instr 9
...   ...
29    Instr 29
...   ...
```

Figure 4.3.9 Firmware used for JUMPCS branch not execution with a subsequent NOP verification

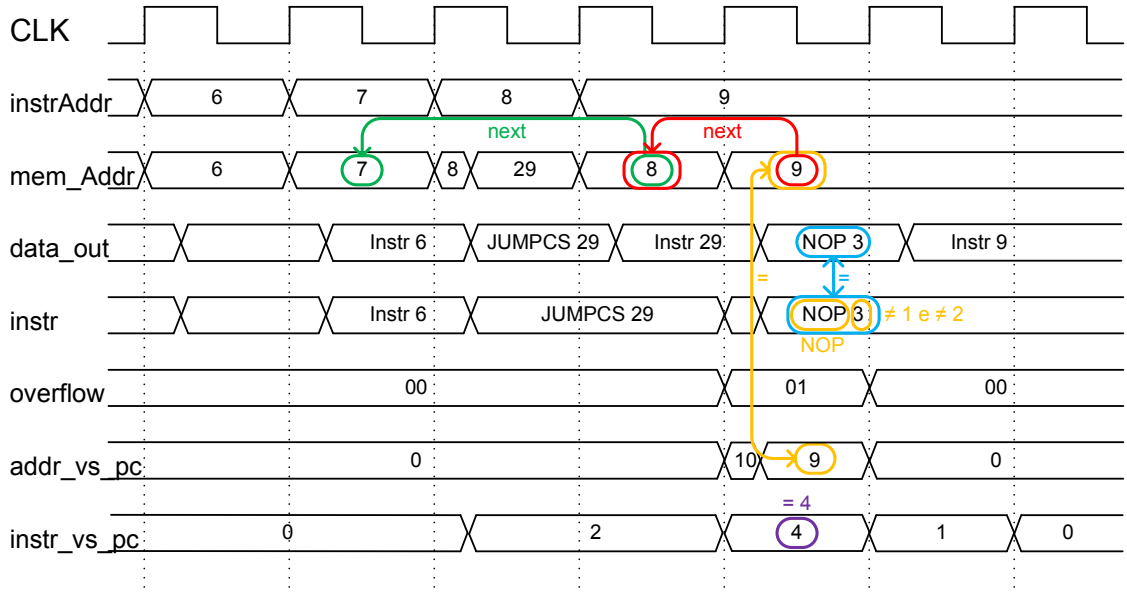

Figure 4.3.10 PSL assertion verification in case of JUMPCS branch not execution with a subsequent NOP

The following PSL assertion checks the correct behavior of JUMPCS in case of branch execution.

```
-- psl Property p_jumpcs_ok is
 -- always s_jumpcs_ok |->
  -- (instr=  prev(data_out)
  --   or (prev(data_out) = WAIT_126_0
  --     and instr = NOP_0))

  -- and (next(mem_addr, prev(mem_addr,2), nInstr)
   -- or (((instr(li-1 downto li-lc) = "1010")
    -- and (mem_addr = instr(li-lc-4 downto li-lc-limm_j)))))

  -- and (prev(mem_addr,2) = prev(data_out(li-lc-4 downto li-lc-limm_j), 2))

  -- and instr_vs_pc = 4

  -- and ((addr_vs_pc = mem_addr+1)
   -- or ((instr(li-1 downto li-lc)="1011")
    -- and (addr_vs_pc = mem_addr)
    -- and (not(instr(li-lc-4 downto li-lc-limm_j)=1))
    -- and (not(instr(li-lc-4 downto li-lc-limm_j)=2))));

-- psl Assert p_jumpcs_ok Severity ERROR;
```

The characteristics to be verified are:

- the instruction to execute is *data_out*, the output of Memory, at the previous clock cycles [blue];

- the next instruction to be read is subsequent to the present instruction (except in case of JUMP as present instruction) [red];
- the present instruction address, *mem_addr*, is subsequent to JUMPCS immediate field [green];
- *instr_vs_pc* must assume the value '4' for the correct configuration of Program Counter [violet];
- *addr_vs_pc* must assume *mem_addr* value incremented of 1 (except in case of NOP *k* instruction, when *k* is different from 1 and 2) [yellow].

The firmware used for JUMPCS branch execution verification is shown in Figure 4.3.11and the related PSL assertions verification is shown in Figure 4.3.12.

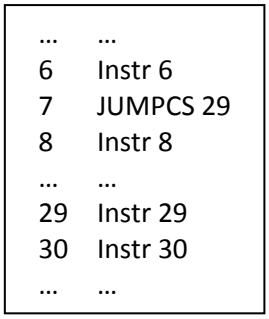

Figure 4.3.11 Firmware used for JUMPCS branch execution verification

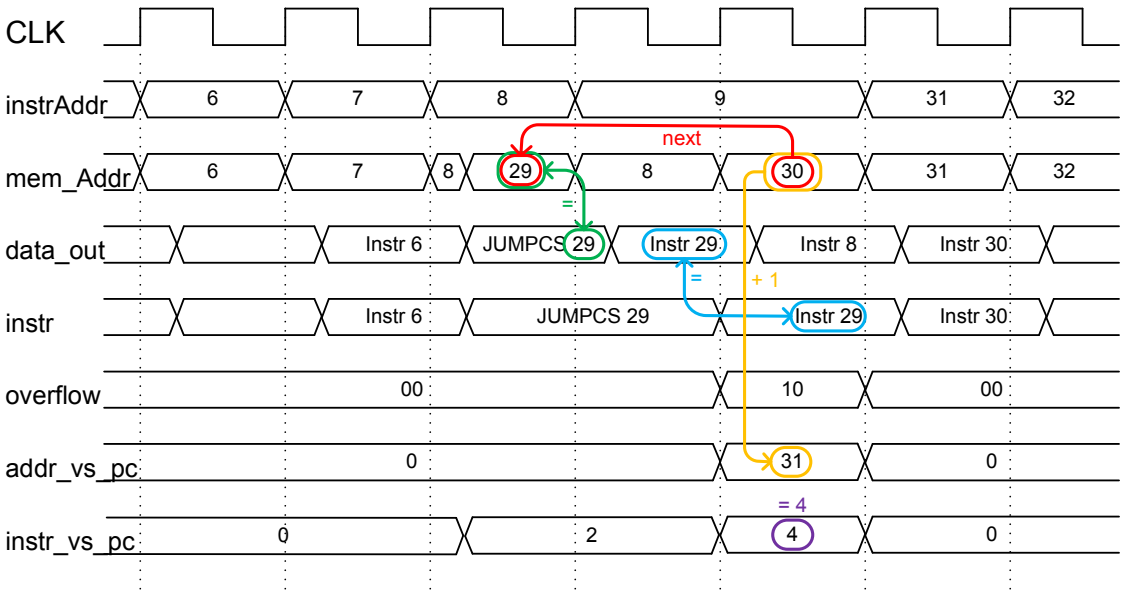

Figure 4.3.12 PSL assertion verification in case of JUMPCS branch execution

In case of JUMPCS with branch execution with JUMP as subsequent instruction PSL assertion verifies some different features (Figure 4.3.13 and Figure 4.3.14).

```
...   ...
6     Instr 6
7     JUMPCS 29
8     Instr 8
...   ...
29    JUMP 45
...   ...
45    Instr 45
```

Figure 4.3.13 Firmware used for JUMPCS branch execution with a subsequent JUMP verification

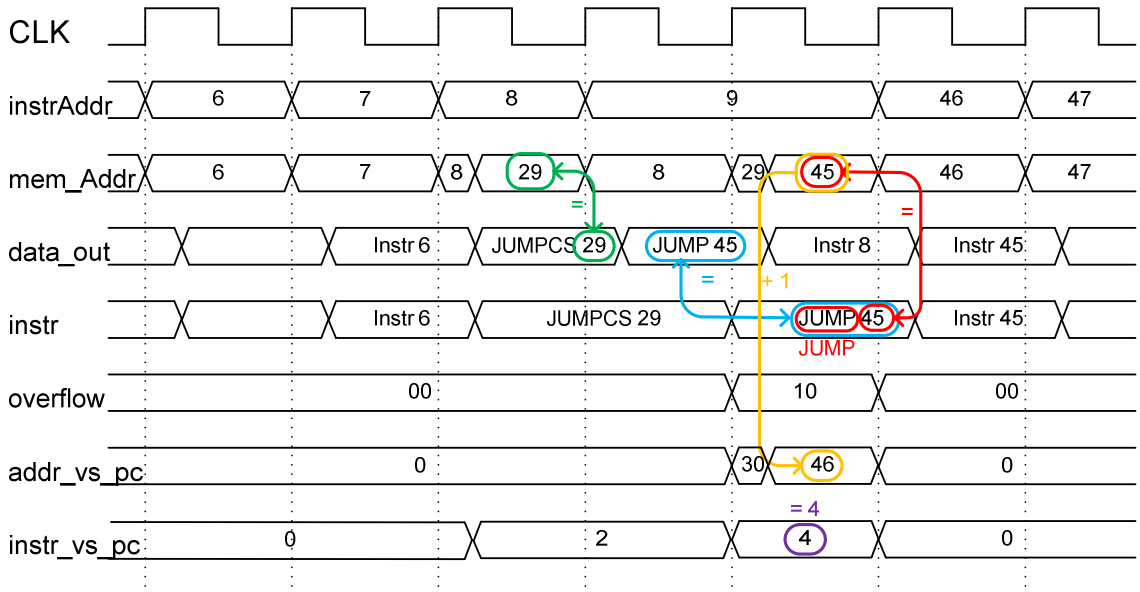

Figure 4.3.14 PSL assertion verification in case of JUMPCS branch execution with a subsequent JUMP

In case of JUMPCS with branch execution with NOP as subsequent instruction PSL assertion verifies some different features (Figure 4.3.15 and Figure 4.3.16).

```
...   ...
6     Instr 6
7     JUMPCS 29
8     Instr 8
...   ...
29    NOP 3
30    Instr 30
```

Figure 4.3.15 Firmware used for JUMPCS branch execution with a subsequent NOP verification

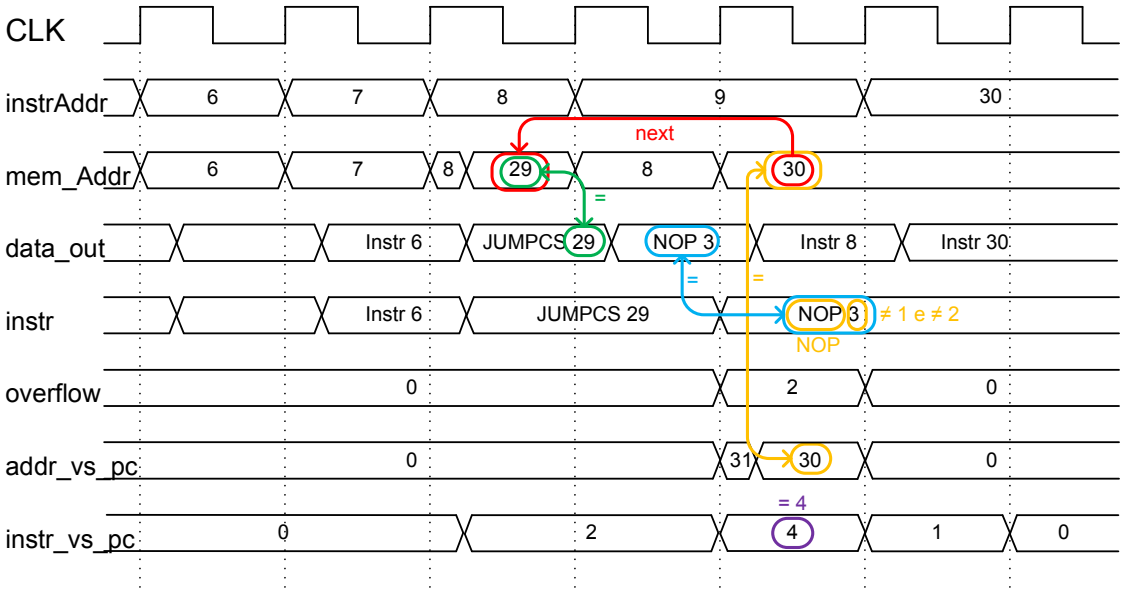

Figure 4.3.16 PSL assertion verification in case of JUMPCS branch execution with a subsequent NOP

4.4 Simulation based verification

After the PSL assertions insertion in the RTL source code the behavior of Microprogrammed Servo Sequencer has been verified by means of simulations. At first JUMP, JUMPCS, NOP and their interactions has been verified. Then WAIT and WAIT SEQR_SG, that depends on input signals values, has been verified. For these instructions not only firmware, but also input signals values have been changed. At last Signal Mapping has been verified.

So there are three different verification phases:

- JUMP, JUMPCS and NOP verification: different versions of firmware have been written to check all the possible situations that may cause uncorrected behaviors. In Table 4.4.1 the firmwares used contain some registers labels: reg0, reg1 and reg2. The verification process consists of a simulation with these firmwares to point out uncorrected behaviors. The previous firmwares test each

Table 4.4.1 Firmwares for simulation based verifications of JUMP, NOP and JUMPCS instructions

JUMP		JUMPCS		NOP	
A:		SET_V	reg0, 0	NOP	1
NOT	reg2	A:		NOT	reg0
JUMP	C	COMPI	reg0, 0	NOP	2
B:		JUMPCS	C	NOT	reg1
NOT	reg1	B:		NOP	3
JUMP	A	NOT	reg1	NOT	reg2
C:		C:		NOP	27
NOT	reg0	COMPNI	reg0, 0	NOT	reg0
JUMP	B	JUMPCS	B	NOP	1023
				NOT	reg1

instruction alone, but it is necessary to check also interactions between them; so in Table 4.4.1 an example of some firmwares to test the complex situations of NOP after JUMPCS instruction are shown.

In Table 4.4.2 the firmwares represent the following situations:

a) NOP 1 after JUMPCS branch taken;
b) NOP 2 after JUMPCS branch taken;
c) NOP 1023 after JUMPCS branch taken;
d) NOP 1 after JUMPCS branch not taken;
e) NOP 2 after JUMPCS branch not taken;
f) NOP 1023 after JUMPCS branch not taken;

Table 4.4.2 Firmwares for simulation based verifications of NOP and JUMPCS instructions possible interactions

a) STORE reg0, 1 COMPI reg0, 1 JUMPCS A NOT reg1 A: NOP 1 AND reg0, reg1	b) STORE reg0, 1 COMPI reg0, 1 JUMPCS B NOT reg1 B: NOP 2 AND reg0, reg1	c) STORE reg0, 1 COMPI reg0, 1 JUMPCS C NOT reg1 C: NOP 1023 AND reg0, reg1
d) STORE reg0, 1 COMPI reg0, 0 JUMPCS C NOP 1 D: AND reg0, reg1	e) STORE reg0, 1 COMPI reg0, 0 JUMPCS C NOP 2 E: AND reg0, reg1	f) STORE reg0, 1 COMPI reg0, 0 JUMPCS C NOP 1023 F: AND reg0, reg1

- For WAIT and WAIT SEQR_SG the test bench has been modified to change input signals values because these instructions depends on these ones. Some firmwares are written to analyze both single instruction behavior both all possible interactions with other out-of-order instructions.
- For Signal Mapping verification two different firmwares are realized: in the first all instructions that may use signal mapping mechanism have been introduced; in the second all the possible memory locations for Signal Mapping are allocated, then verified with all the instructions. Not only the correct behavior of Signal Mapping mechanism has been verified, but also memory reconfigurations.

With this approach all the wrong behaviors have been corrected. For every kind of Microprogrammed Servo Sequencer needed to fit different Servo application versions the Assertion Based Verification allows to verify MSS behavior. This kind of approach can also be applied in the Test phase achieving prototyping verification.

4.5 Bibliography

[4.1] Matteo Miotti, “Studio ed implementazione di una metodologia di verifica funzionale di un’architettura a microcontrollore dedicate”, Tesi di Laurea Specialistica in Ingegneria Elettronica, Università degli Studi di Pavia, a.a. 2007 – 2008.

[4.2] Janick Bergeon, Writing Testbench using systemverilog, Springer-Verlag, New York, 2006.

[4.3] Perry D. L., Foster H. D., *Formal verification: for digital circuit design*, McGraw-Hill Professional, 2004.

[4.4] Harry Foster, slides “Introduction to verification academy”, *Mentor Graphics' Verification Academy.*

[4.5] IEEE Standard 1850 for Property Specification Language (PSL).

[4.6] IEEE Standard 1800 for SystemVerilog— Unified Hardware Design, Specification, and Verification Language.

[4.7] System Verilog assertions handbook, Di Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari

5 Case study and synthesis results

The Microprogrammed Servo Sequencer is able to emulate different Regular Servo Sequencer state diagrams. A Regular Servo Sequencer industrial product is used for testing the ability of MSS. Microprogrammed Servo Sequencer behavior is compared to RSS one. Then the MSS is synthesized in 1.2 V, 65 nm CMOS technology. MSS synthesis results are compared with RSS results.

5.1 Regular Servo Sequencer Finite State Machine

The Regular Servo Sequencer is a part of an industrial product. It is implemented with a hardwired synchronous Finite State Machine. At system Start up the RSS FSM is in idle waiting for a Servo pattern reading or writing. When a Servo pattern is going to be read the *Servo Gate* signal is activated: the Regular Servo Sequencer has to orchestrate Servo operations.

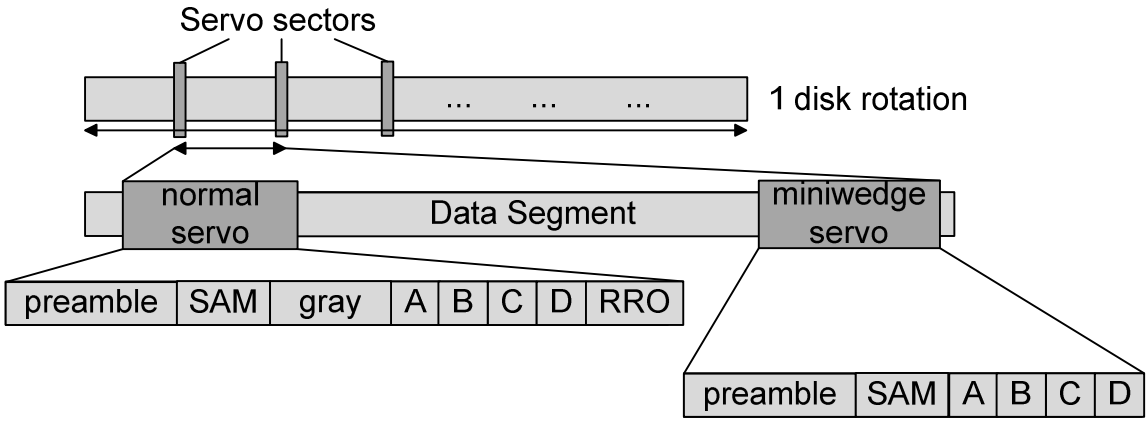

Figure 5.1.1 User, normal servo and miniwedge servo data

In Figure 5.1.2 the Regular Servo Sequencer Finite State Machine is shown:

- IDLE is the start state;
- SVO_SG1 and SVO_SG2 states, activated by the *Servo Gate*, prepare the RSS to the Servo pattern decoding process;
- WAIT_INPUT process waits for the first data of normal Servo pattern (it might be the *preamble* field);
- PGR_DATA_PRMBL decodes the normal Servo *preamble* field; if this field is not found the *Servo Gate* is closed and all the Servo operations are interrupted by means of CLOSE_SG state;
- if the *preamble* is decoded correctly the WAIT_SAM_SRCH state waits for the *Servo Address Mark* (SAM) field;
- SAM_SRCH decodes *preamble* field;
- GC_DET decodes the *gray code* which arrives after the *preamble* field;
- BRST_DEM decodes the bursts A, B, C and D fields.

The normal Servo pattern might have zero, one or two *Repetable Run Out* (RRO) fields:

- WAIT_RRO1, PGR_RRO1, DATA_RRO1, WAIT_RRO2, PGR_RRO2, DATA_RRO2 decode RRO1 and RRO2 fields, if they are present in the Servo pattern;
- WAIT_INPUT_MWG, PGR_MWEGDE, SAM_MWEGDE and BRST_ MWEGDE states act the Miniwedge Servo pattern (Figure 5.1.1) decoding process;

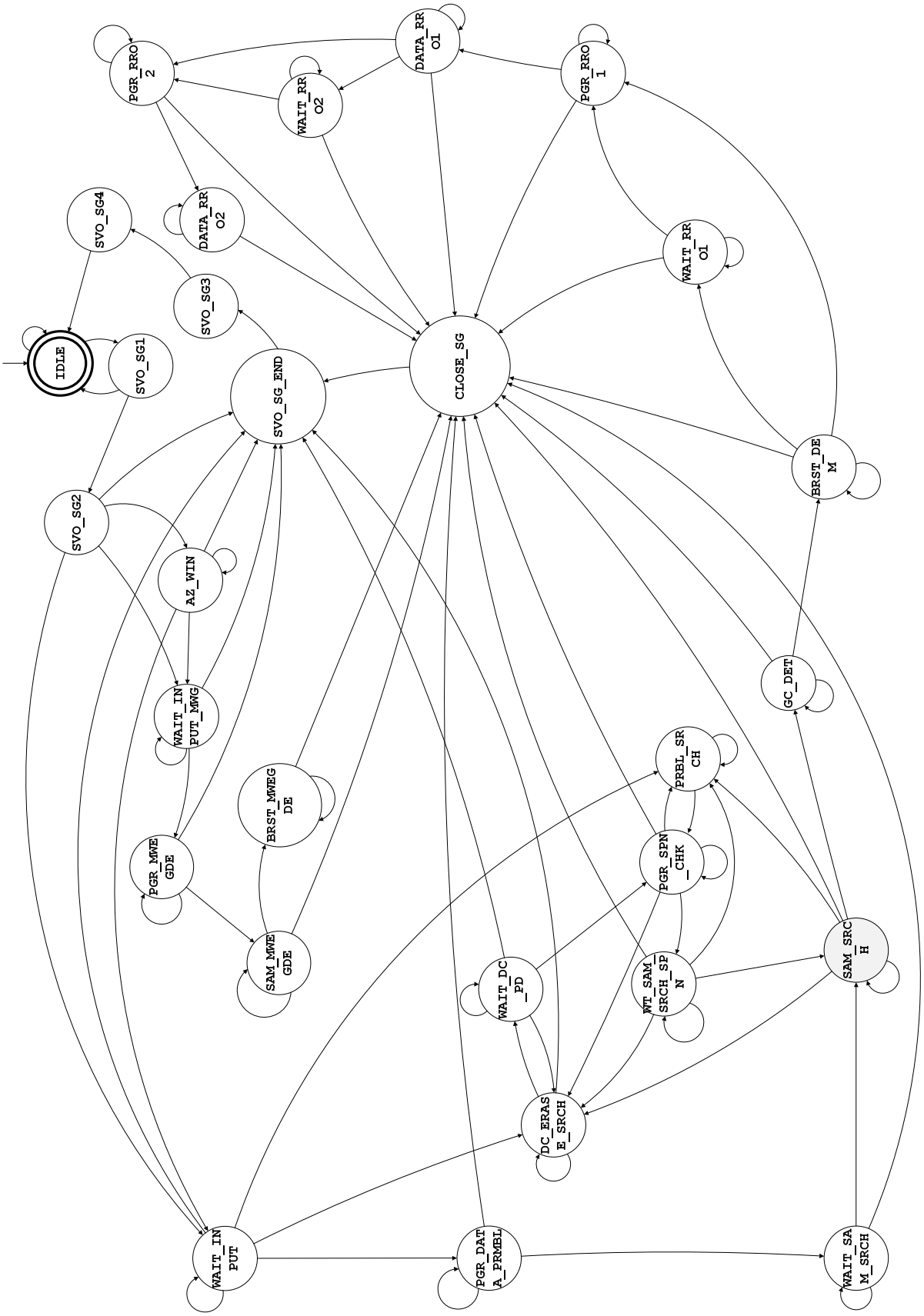

Figure 5.1.2 Regular Servo Sequencer Finite State Machine Diagram

- DC_ERASE_SRCH, WAIT_DC_PD, WT_SAM_SRCH_SPN, PGR_ SPN_CHK, PRBL_SR_CH manage Servo operations in case of *SAM* not found and *preamble* not found.

Every minimal modifications of this Finite State Machine cause the need of redesign and verify all the Regular Servo Sequencer FSM. The use of ASIP architecture assures flexibility to the system due to the ability of emulating more Finite State Machines. Thanks to this feature the redesign and verification of a complex Finite State Machine can be substituted by simple firmware rewriting [5.1].

5.2 Microprogrammed Servo Sequencer approach

In Figure 5.2.2 the Finite State Machine diagram emulated by Microprogrammed Servo Sequencer is shown: it's behavior is the same of Regular Servo Sequencer Finite State Machine (Figure 5.1.2). This diagram is simply the firmware scheme: each state corresponds to a label, which stands for an instruction address (Figure 5.2.1). In the diagram some states are added (the grey ones) due to the MSS ASIP nature: an hardwired Finite State Machine is a parallel hardware (it can take a decision choosing between more than two future operations at a time, in the diagram more than two future states); a processor, an Application Specific Processor in this case, is a sequential hardware (it can take a decision choosing between only two future operations at a time). States insertion is necessary to maintain the same behavior of the hardwired RSS FSM. The operation of states addition does not cause any complexity increasing of the ASIP architecture (it is a zero cost operation) because states are simply labels that are translated by the assembler in instruction addresses (Figure 5.2.1); instead the same operation for an hardwired FSM causes an increasing of complexity and area occupation.

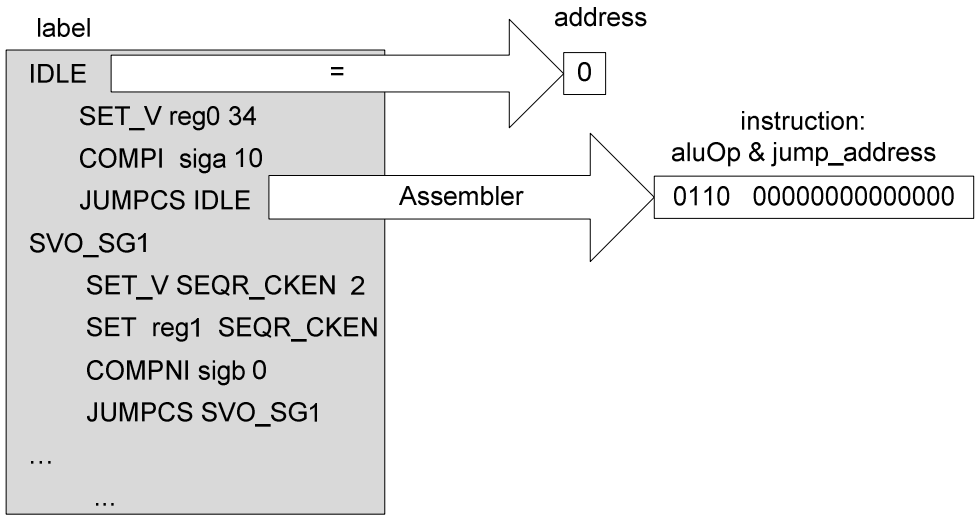

Figure 5.2.1 State label correspondence (firmware example)

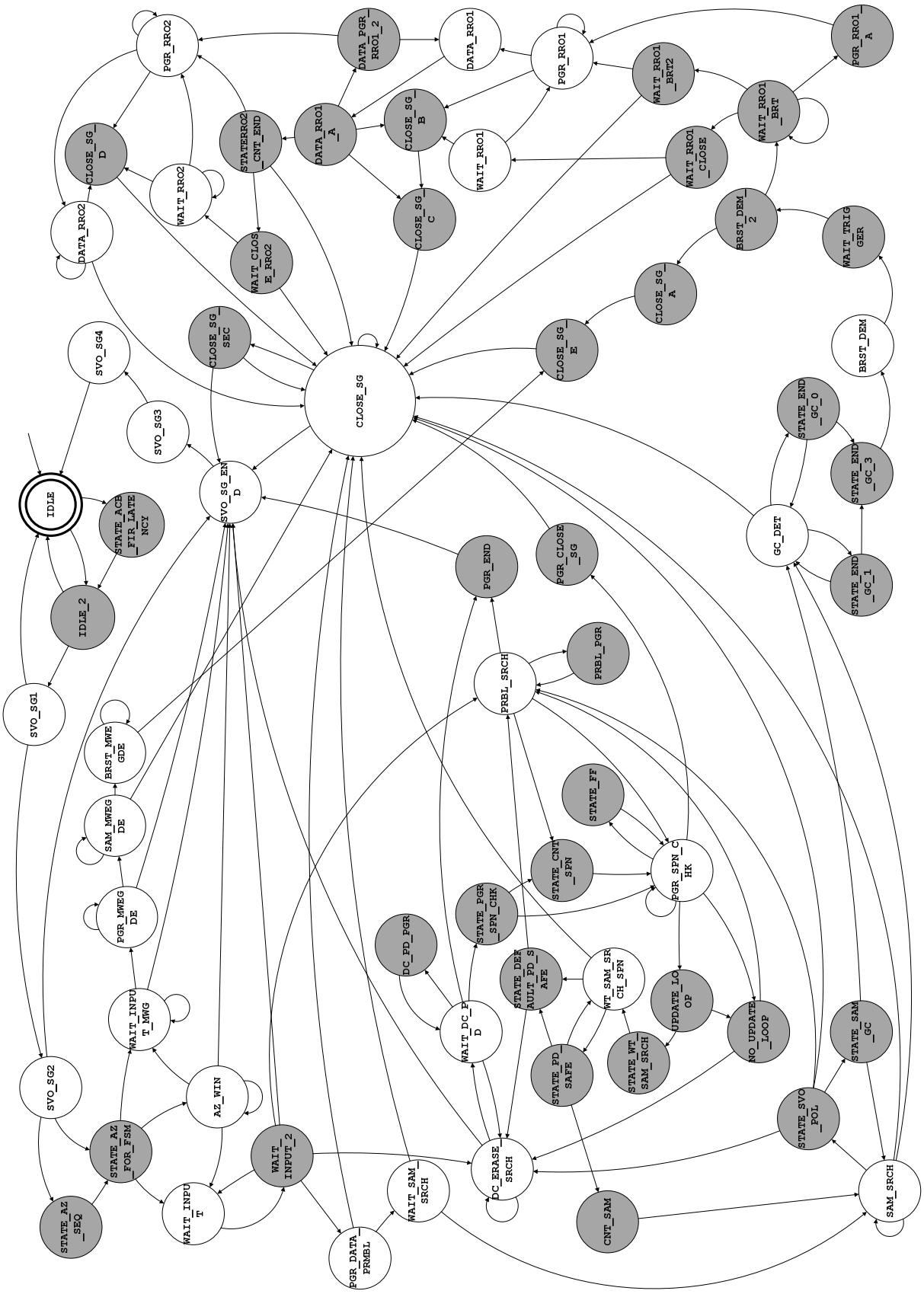

Figure 5.2.2 Finite State Machine diagram emulated by Microprogrammed Servo Sequencer

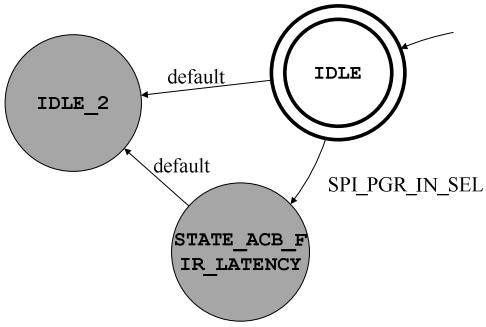

Figure 5.2.3 Part of MSS FSM

```
IDLE
      NMI SG 1
      COMPI SPI_PGR_IN_SEL 1
      JUMPCS STATE_ACB_FIR_LATENCY
      ADD SPI_SVO_LATENCY ACB_LATENCY
      ADD Acc SPI_SVO_AZ_PGR
      COUNTER INT_CNT_GEN Acc
      JUMP IDLE_2
STATE_ACB_FIR_LATENCY
      ADD SPI_SVO_LATENCY ACB_FIR_LATENCY
      ADD Acc SPI_SVO_AZ_PGR
      COUNTER INT_CNT_GEN Acc
IDLE_2
      …
```

Figure 5.2.4 Firmware example

In Figure 5.2.3 a part of MSS FSM diagram is illustrated [5.2]. The *Servo Gate* (SG) that informs about the Servo pattern read process beginning from the media is the Servo operation enable signal: when it is high the MSS starts Servo operations, when it is low the MSS is IDLE. In (Figure 5.2.4) there is a part of Microprogrammed Servo Sequencer firmware which emulates this part of RSS state diagram. In states IDLE and STATE_ACB_FIR_LATENCY the generic counter that considers the latency of some Servo system blocks is initialized: it is possible to have different system latencies depending on servo pattern characteristics.

The complete firmware which emulates the Regular Servo Sequencer Finite State Machine is shown in the Appendix 9.2.

5.3 Microprogrammed Servo Sequencer dimensioning

The Regular Servo Sequencer characteristics has been studied to identify the correct dimensioning of Microprogrammed Servo Sequencer. The programmable ports are 20, 7 bit wide to match the 100 in/out signals of RSS (in case of signals extension greater than 7 bit CONCAT instruction provides a concatenation mechanism to recover these signals). Port addressing is 5 bit wide, but port number is 20 to match Servo operations needs reducing so power consumption and area. It is possible to extend port number till 32 in case of signals number and/or dimension increasing. The MSS Register Memory is

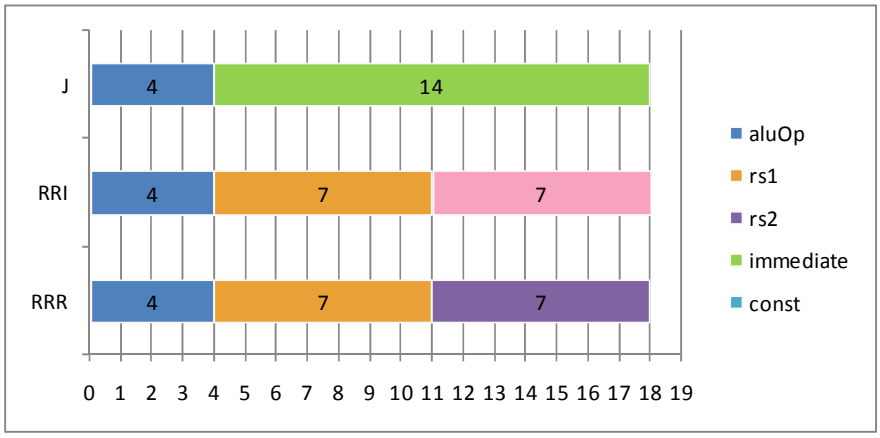

Figure 5.3.1 MSS Instruction-set

dimensioned with 128 word of 12 bit: the RSS signals are about 100 and the remaining memory locations are used for internal registers values; each register is 12 bit wide to contain also 7 bit signals and to have the possibility to execute some ADD instructions between two 7 bit signals without any addition overflow. So there is a unique 7 bit addressing space for signal mapping and registers. With this bit information each instruction type can be dimensioned (Figure 5.3.1). The R instruction type contains *rs1* and *rs2* signals/registers fields; I instruction type contains *rs1* signal/register field and immediate *const* field; J instruction type contains 14 bit *immediate* field. For this particular Servo application 480 firmware instructions are needed so a 9 bit addressing is needed and the Instruction Memory is dimensioned with 512 word of 18 bit. It is possible to expand this memory till 16384 words maintaining the same instruction-set.

All the RTL source code that describes the Microprogrammed Servo Sequencer is parametric: it is possible to change MSS ISA characteristic to fit different Servo applications.

5.4 Behavioral matching verification

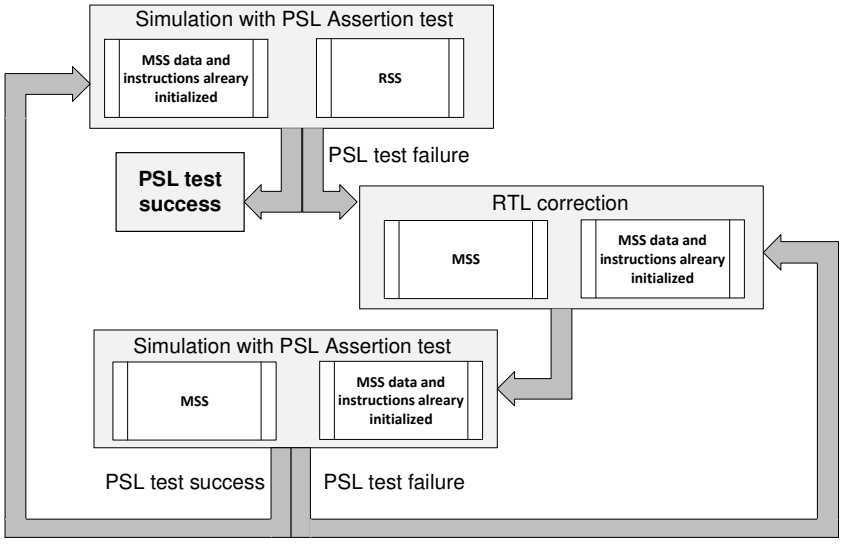

Figure 5.4.1 Behavioral matching verification

The MSS functional simulations have been compared to the RSS on behavioral matching verification (Figure 5.4.1): at the beginning a MSS with the data and register memories already initialized has been described; it has been then compared with the MSS with the Reprogramming FSM through the functional verification simulation-based technique; at the end the MSS already initialized has been compared with the Regular Servo Sequencer for the behavioral matching verification. This kind of behavioral verification assures also the correct loading of data memory and instruction memory: the loading operations are verified by means of PSL assertions.

5.5 Synthesis results

The MSS has been synthesized in 1.1 V, 65 nm CMOS technology with three types of transistors: High Voltage Threshold (HVT), Standard Voltage Threshold (SVT) and Low Voltage Threshold (LVT). The Instruction Memory makes use of an Intellectual Property (IP) single port RAM with 512 words and 18 word bits in 1.2 V, 65 nm CMOS technology with SVT transistor: in Figure 5.5.1 IP memory symbol is shown; in Table 5.5.1, Table 5.5.2, Table 5.5.3, Table 5.5.4 the main characteristics of this memory are indicated. Single port memory with process worst, 1.1 V and temperature 125°C is considered to obtain Microprogrammed Servo Sequencer area and slack synthesis results, whereas Single port memory with process best, 1.3 V and temperature 125°C to obtain dynamic and leakage power synthesis results.

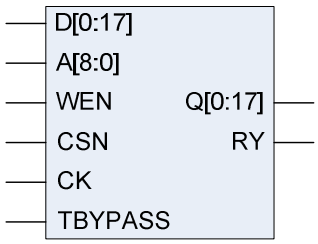

Figure 5.5.1 RAM single port symbol

Table 5.5.1 RAM single port primary parameters

Primary Parameters	
Parameter	**Value**
Number of words	512
Number of bits	18
Number of multiplexer inputs	8
Driving capability	35
Transistor	Standard
Redundancy	No
Bit Mask	no
Debug Mode	Not Available
Pipeline	no
Zero Hold Time	Yes

Table 5.5.2 RAM single port pin description

Pin Description	
Pin Name	**Pin Function**
CK	External clock input for the memory.
CSN	Chip Select pin. When this input is logic low, memory is enabled and read/write operations can be performed.
WEN	Write Enable pin. When this input is logic low, memory is in the write mode.
A[8:0]	Address Input. The Address input is used to address the location to be read during the read cycle and written during the write cycle.
D[0:17]	Data Input bus. This is used to write data to the memory location specified by the Address Input port during the write cycle.
TBYPASS	Memory Bypass in Test Mode. It is used for data path checking. This signal is not dependent on clock. Therefore, no setup or hold time is required. Whenever this signal is active, the output bus (Q) gets the value of the input bus (D) in a specified time delay. This pin is managed by BIST.
Q[0:17]	Data output bus. Generates the contents of the memory location addressed by the Address Input signals. Q is always buffered.
RY	Memory Handshake signal.

Table 5.5.3 RAM single port derived parameter

Derived Parameters	
Parameter	**Value**
Number of rows (rows)	64 words/mux
Number of columns (cols)	144 bits*mux
Aspect ratio	0.444 rows/cols
Capacity	9216 words*bits

Table 5.5.4 RAM single port physical parameter

Physical Parameters	
Parameter	**Value**
Width	187.2 um
Height	64.0 um
Area	11980.8 um^2

The Register Unit has been implemented by means of an IP dual port RAM with 128 words and 12 word bits in 1.2 V, 65 nm CMOS process technology with SVT transistor:

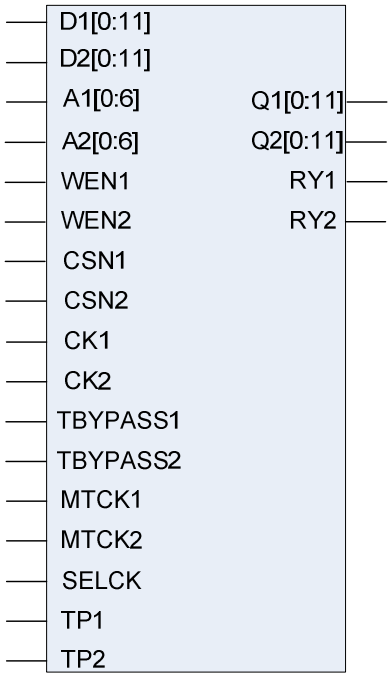

Figure 5.5.2 RAM dual port symbol

in Figure 5.5.2 IP memory symbol is shown; in Table 5.5.5, Table 5.5.6, Table 5.5.7, Table 5.5.8 the main characteristics of this memory are indicated. Dual port memory with process worst, 1.1 V and temperature 125°C is considered to obtain MSS area and slack synthesis results, whereas dual port memory with process best, 1.3V and temperature 125°C to obtain dynamic and leakage power synthesis results.

Table 5.5.5 RAM dual port primary parameters

Primary Parameters	
Parameter	**Value**
Number of words	128
Number of bits	12
Number of multiplexer inputs	8
Driving capability	35
Transistor	Standard
Redundancy	No
Bit Mask	no
Write Test Mode	Available
Pipeline	no
Zero Hold Time	Yes

Table 5.5.6 RAM dual port pin description

Pin Description	
Pin Name	**Pin Function**
CK1,2	External clock input for the memory.
CSN1,2	Chip Select pin. When this input is logic low, memory is enabled and read/write operations can be performed.
WEN1,2	Write Enable pin. When this input is logic low, memory is in the write mode.
A1,2[0:6]	Address Input. The Address input is used to address the location to be read during the read cycle and written during the write cycle.
D1,2[0:11]	Data Input bus. This is used to write data to the memory location specified by the Address Input port during the write cycle.
SELCK	Clock select Mux. This will select either of the two clocks i.e. When this input is logic low, Functional clock 'CK' is selected and when this input is logic high, BIST clock 'MTCK' is selected.
MTCK1,2	BIST clock. This will be active in test mode of memory.
TP1,2	Write Test Mode to enable special BIST test mode. This mode emulates the worst write clock skew condition and it is mandatory for Low Leakage option.
TBYPASS1,2	Memory Bypass in Test Mode. It is used for data path checking. This signal is not dependent on clock. Therefore, no setup or hold time is required. Whenever this signal is active, the output bus (Q) gets the value of the input bus (D) in a specified time delay. This pin is managed by BIST.
Q1,2[0:11]	Data output bus. Generates the contents of the memory location addressed by the Address Input signals. Q is always buffered.
RY1,2	Memory Handshake signal.

Table 5.5.7 RAM dual port physical parameter

Physical Parameters	
Parameter	**Value**
Width	247.2 um
Height	44.8 um
Area	11074.56 um^2

Table 5.5.8 RAM dual port derived parameter

Derived Parameters	
Parameter	**Value**
Number of rows (rows)	16 words/mux
Number of columns (cols)	96 bits*mux
Aspect ratio	0.167 rows/cols
Capacity	1536 words*bits

The synthesis process has been realized through Synopsys Design Compiler® at 300 MHz, the elaborating frequency of the actual Regular Servo Sequencer, with 1.1 V, 65 nm, process worst CMOS technology. The dynamic power dissipation and leakage have been simulated onto the Standard Cell implementation of Microprogrammed Servo Sequencer in a 65 nm CMOS technology at 1.3V process best, getting in this way the power dissipation worst case.

Microprogrammed Servo Sequencer area and timing outputs have been compared to RSS synthesis ones. The main results have been summarized in Table 5.5.9. The MSS is nine times bigger than RSS, the dynamic power dissipation is about two times higher whereas the leakage is approximately six times higher but, being MSS programmable, it is possible to change its behavior.

Table 5.5.9 Synthesis results

Device	**Library**	**SYNTHESIS RESULTS**		
		Area [µm^2]	**Dynamic power [mW]**	**Leakage Power [µW]**
RSS	HVT, SVT, LVT 65nm	4,2 x 10^3	1,4	232
MSS (5.3)	HVT, SVT, LVT 65 nm	36,7 x 10^3	3.24	1234

In Table 5.5.10 Synthesis results details are shown: the combinational area is about 26 % of the MSS total area, the non combinational area is about the 12 % of MSS area and black boxes area (Intellectual Property single port RAM and dual port RAM) is about the 63 % of the MSS total area. Microprogrammed Servo Sequencer area and timing outputs have been compared to RSS synthesis ones.

Table 5.5.10 Synthesis results details

Global Cell Area		**Local Cell Area**					
Absolute Total [µm^2]	**%**	**Combinational [µm^2]**	**%**	**Non combinational [µm^2]**	**%**	**Black boxes [µm^2]**	**%**
36770	100	9390	~25	4326	~12	23055	~63

In case of longer firmware it is possible to extend the RAM single port from 512 to 16384 words with few changing in the RTL code. For this particular Servo application 380 firmware instructions are needed. However the Microprogrammed Servo Sequencer is parametric, so it is possible to reduce or to extend every instruction or register fields with simple RTL code changes.

5.6 Bibliography

[5.1] Zhenyu Liu, Tughrul Arslan, Sami Khawam, Iain-Lindsay, “A High Performance Synthesisable UnsymmetricaI Reconfigurable Fabric For Heterogeneous Finite State Machines”, *Proc. of the 2005 Design Automation Conference Asia and South Pacific*, IEEE, pp. 639 – 644, Vol. 1, Jan. 2005.

[5.2] P. Baldrighi, M.M. Maggi, M. Castellano, C. Vacchi, D. Crespi, P. Bonifacino, **“**Implementation of Microprogrammed Hard Disk Drive Servo Sequencer”, *Proc. 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools,* IEEE, pp. 442 – 446, Sep. 2008.

6 Conclusion

The Application Specific Instruction-set Processors has a great diffusion in different applications thanks to continuous increasing of chip density. This Ph. D. thesis presents an innovative approach for the realization of the Servo Subsystem of an Hard Disk Drive R/W channel. The Microprogrammed Servo Sequencer (MSS) designed implements a generic Servo Subsystem usually realized with an hardwired approach (a Finite State Machine). MSS is an Application Specific Instruction-set Processors: it is composed by an elaborating core, embedded counters, programmable I/O ports and a Reprogramming Finite State Machine; the MSS instruction set contains both generic both custom instructions dedicated to Servo operations. All the RTL source code that describes Microprogrammed Servo Sequencer is parametric so that it is possible to dimension part of Instruction Set fields to fit different Servo system features. Moreover the reprogramming process provides Microprogrammed Servo Sequencer behavior changing: after Integrated circuit fabrication it is possible to change its behavior by means of dedicated firmwares for different Servo System fitting. The Assertion Based Verification process supplies the MSS behavior improvement reducing functional bugs number before manufacturing process. This kind of approach can be used not only in Verification process, but also in Test phase. Moreover for every MSS versions needed to fit different Servo applications Assertion Based Verification can be reused to verify MSS behavior.

The MSS is then reprogrammed with a dedicated firmware for the emulation of the Hard Disk Drive Servo read/write channel servo system; results on this real application prove that the Microprogrammed Servo Sequencer can operate like a Servo System. Synthesis results show that the MSS area is bigger than Servo System one, both dynamic and leakage power dissipation is higher but, being MSS RTL source code parametric it is possible to change ISA dimensioning to fit new Servo system features; after that being MSS programmable it is possible to change its behavior after IC fabrication.

7 Figures index

8 Tables index

9 Appendixes

9.1 Property Specification Language assertion

PSL language comes from IMB Sugar language. It has been improved from Accelera and in 2005 it became an IEEE standard. It is compatible with VHDL, *Verilog*, *SystemC* and *SystemVerilog*. A methodology based on *property* definition has been defined. It has great verification potentiality that allow to increment productivity and quality of electronics devices reducing so time-to-market.

An important feature of PSL language is the possibility of using temporal relation by means of SERE (*Sequential Extended Regular Expressions*): it is possible to control both the value of a signal in an instant both a signal temporal sequence evolution by means of automatic function defined in PSL.

Properties can describe desirable device behavior and eventually (an assertion must be activated with an assert) check the device.

PSL language has three different level:

- *Boolean:* a functionality that has to be checked is defined by means of the control of signal and variable values present in the source code of the device. These functionalities is described as properties evaluated during simulation.
- *Temporal:* it is possible to evaluate device feature in different moment and to consider also signal temporal sequences. If these temporal checks were described in an HDL language they would lead to a waste of registers and complex logic.
- *Verification:* this level informs simulation tools about directives such as properties that have to be checked.

PSL assertions can be directly introduced in RTL source code as special comments: the word "*psl*" at the beginning of comment content permits to recognize assertions. For simulation tools these special comments are considered as assertions and elaborated to execute verification process; instead synthesis tools don't consider any of these comments to avoid the insertion of non functionality features in the device netlist.

The following examples explain PSL assertion and SERE use.

The first example defines the property P1: it checks that READ and WRITE signals are never contemporary activated.

```
-- psl property P1 is never WRITE and READ;
-- psl assert P1;
```

In Figure 9.1.1 the assertion behavior is shown during simulation phase (considering positive logic).

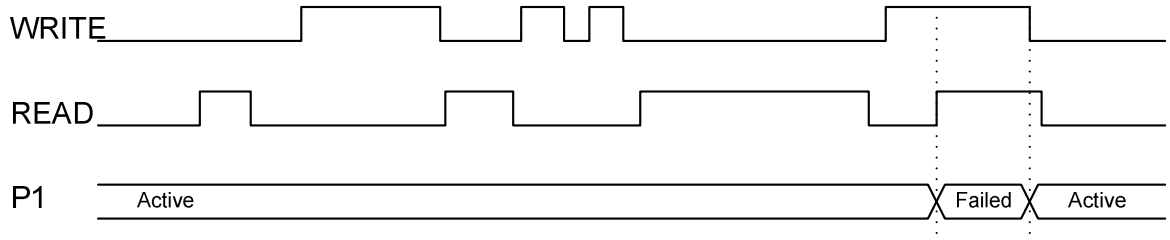

Figure 9.1.1 PSL P1 assertion example

In Figure 9.1.2 a SERE PSL assertion activated at positive clock event is shown: when A and B signals is at high logic level in temporal sequence then C must becomes high in the next clock cycle and in the following clock cycle A, B and C are low level.

```
-- psl Default Clock is rising_edge(CLK);
-- psl property P2 is always { A ; B } |=> { C ; not (A or B or C) };
-- psl assert P2;
```

In Figure 9.1.2 during simulation the assertion is activated three times: two successes and one failure.

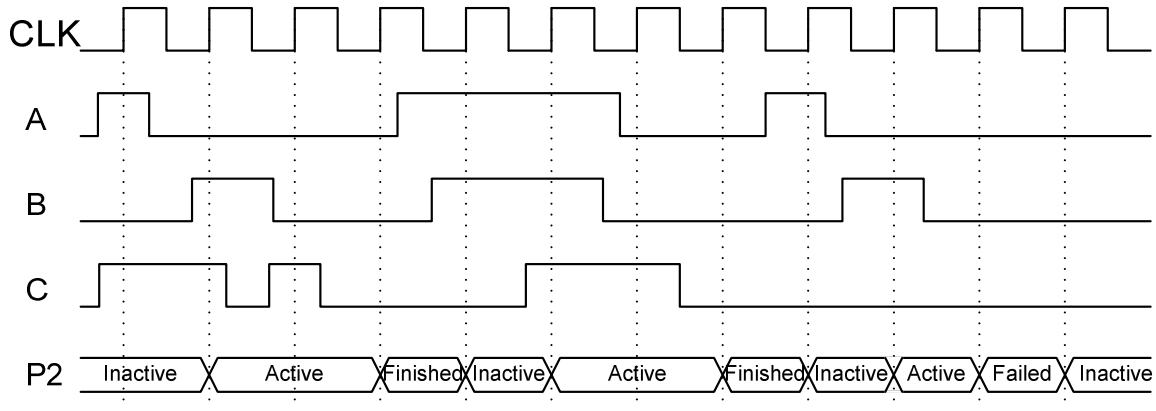

Figure 9.1.2 SERE PSL P2 assertion example

9.2 MSS Firmware

```
#
deafault_jump 009
LAT_DC_PD 008
SG_EN_1 000
AZ_FOR_FSM 000
PREAMBLE_CNT_EN 000
PBL_FND_D 000
PD_SAFE_LOST 000
SAM_SRCH_STRT_D 000
PREAMBLE_SAFE 000
SMD_EXP_FLAG 000
SMD_DET_FLAG 000
SMD_QUAL_FLAG 000
SPN_PGR_MAX 000
SEQR_RRO_EXP2 000
R_SVO_AZ_DIG 000
ACB_FIR_LATENCY 007
ACB_LATENCY 002
SPN_PGR_MIN 006
SPN_PGR_MAX 00a
t0 000
t1 000
t2 000
t3 000
#
SEQR_SG port_1 1
PGR_READY port_1 2
AZ_SEQ port_1 4
DET_SVO_SMD port_1 8
SVO_SMD_QL_OK port_1 16
SVO_SMD_EXP port_1 32
DET_GRAY_RDY port_1 64
DET_RRO_VL1 port_2 1
DET_RRO_VL2 port_2 2
BRT_END port_2 4
BRT_TRIGGER port_2 8
RRW_END_WRITE port_2 16
SPN_PBL_FOUND port_2 32
SPN_DC_FOUND port_2 64
SVO_SPI_SMD_FND port_3 1
SVO_POL_KO port_3 2
DET_SVOSMD_INV port_3 4
ONE_EARLY_GC port_3 8
ZONE_CHANGE port_3 16
SCAN_MODE port_3 32
INT_CNT_GEN port_4 126
SEQR_LD_DATA port_out_1 1
SEQ_CODE_EN port_out_1 2
SEQR_ITR_ON port_out_1 4
SEQR_BRT_START port_out_1 8
RST_INTF port_out_1 16
SEQ_OK_INTF port_out_1 32
SEQ_UPD_PARAM port_out_1 64
SEQR_CKEN_SVO port_out_2 1
SEQR_RST_LOOPS port_out_2 2
INT_SAM_SRC port_out_2 4
SG_EN port_out_2 8
SVO_AZ_DIG port_out_2 16
SEQR_LD_LOOPS port_out_2 64
SEQR_RRO_EXP1 port_out_3 1
SEQR_AGC_EN port_out_3 2
SEQ_ID_RRO port_out_3 4
SEQR_PGR_SRST port_out_3 8
SEQR_PGR_START  port_out_3 16
SEQ_UPDATE_LOOP port_out_3 32
SEQR_UPD_GLS port_out_3 64
SEQR_FL_SG port_out_4 1
INTF_ITR_CODE port_out_1 22
UPD_OK_BRT_ITR port_out_1 108
ITR_CODE port_out_1 6
ITR_LD port_out_1 5
UPD_OK_BRT port_out_1 104
START_SRST_AGC port_out_3 26
START_SRST port_out_3 24
UPDATE_SRST port_out_3 40
UPDATE_SRST_AGC port_out_3 48
START_RRO port_out_3 17
UPD_START port_out_3 80
SEQ_SEARCH_1 port_SPI_1 1
SPI_SVO_DC_EN port_SPI_1 2
SPI_PGR_IN_SEL port_SPI_1 4
SPI_PGR_WIN_SEL port_SPI_1 8
SPI_SVO_PD_SAFE port_SPI_1 112
SPI_SVO_AZ_PGR port_SPI_2 3
SPI_SVO_LATENCY port_SPI_2 30
SPI_RRO_STP_QBAD port_SPI_2 64
SPI_AZ_DIG port_SPI_3 1
SPI_SVO_AZ_EN port_SPI_3 2
SPI_SG_IGNORE port_SPI_3 4
SVO_RRO_MD port_SPI_3 24
INT_CNT_GEN_end port_in_SPECIAL_1 3
SAM_SRCH_STRT port_in_SPECIAL_1 4
END_GC port_in_SPECIAL_1 8
RD_RRO port_in_SPECIAL_1 16
RRO1_CNT_END port_in_SPECIAL_1 32
RRO2_CNT_END port_in_SPECIAL_1 64
RD_2RRO port_in_SPECIAL_2 1
WR_RRO port_in_SPECIAL_2 2
ANALOG_AZ_RSY port_in_SPECIAL_2 12
ACQ_COUNT_EN port_in_SPECIAL_2 16
SEQ_NSVO_MINIWB port_in_SPECIAL_2
32
SEQR_SG_SYNC port_in_SPECIAL_2 64
#
```

```
IDLE
    NMI SEQR_SG 1
    COMPI SPI_PGR_IN_SEL 1
    JUMPCS STATE_ACB_FIR_LATENCY
    ADD SPI_SVO_LATENCY
ACB_LATENCY
    ADD Acc SPI_SVO_AZ_PGR
    COUNTER INT_CNT_GEN Acc
    JUMP IDLE_2
STATE_ACB_FIR_LATENCY
    ADD SPI_SVO_LATENCY
ACB_FIR_LATENCY
    ADD Acc SPI_SVO_AZ_PGR
    COUNTER INT_CNT_GEN Acc
IDLE_2
    SLL SEQR_SG  1
    OR Acc SEQR_SG
    COMPNI Acc 3
    JUMPCS IDLE
    COMPI SPI_SG_IGNORE 1
    JUMPCS IDLE
    SET_V SG_EN 1
SVO_SG1
    NOT ZONE_CHANGE
    SET  SEQR_CKEN_SVO Acc
    COMPNI SEQR_SG 1
    JUMPCS IDLE
SVO_SG2
    NOT ZONE_CHANGE
    SET  SEQR_CKEN_SVO Acc
    SET_V SEQR_LD_LOOPS 1
    COMPI SEQR_SG 0
    JUMPCS SVO_SG_END
    SET SVO_AZ_DIG R_SVO_AZ_DIG
    COMPI R_SVO_AZ_DIG 0
    JUMPCS STATE_AZ_SEQ
    SET AZ_FOR_FSM AZ_SEQ
    JUMP STATE_AZ_FOR_FSM
STATE_AZ_SEQ
     AND SPI_SVO_AZ_EN ANA-
LOG_AZ_RSY
     SET AZ_FOR_FSM Acc
STATE_AZ_FOR_FSM
     COMPI AZ_FOR_FSM 1
     JUMPCS AZ_WIN
     COMPNI SEQ_NSVO_MINIWB  1
     JUMPCS WAIT_INPUT_MWG
WAIT_INPUT
    COUNTER INT_CNT_GEN 126
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
WAIT_INPUT2
    NMI SEQR_SG_SYNC 0
    JUMPCS SVO_SG_END
    NMI INT_CNT_GEN_end 2
    CONCAT SEQ_SEARCH_1
SPI_SVO_DC_EN
    COMPI Acc 3
    JUMPCS DC_ERASE_SRCH
    CONCAT SEQ_SEARCH_1
SPI_SVO_DC_EN
    COMPI Acc 2
    JUMPCS PRBL_SRCH
    SET_V SEQR_PGR_START 1
PGR_DATA_PRMBL
    NOT  ZONE_CHANGE
    SET  SEQR_CKEN_SVO Acc
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG
    NMI PGR_READY 1
    SET_V INTF_ITR_CODE 7
WAIT_SAM_SRCH
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V INT_SAM_SRC 1
    NMI SAM_SRCH_STRT 1
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG
SAM_SRCH
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    NOT SEQ_SEARCH_1
    COMPI Acc 0
    JUMPCS STATE_SVO_POL
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG
    NMI DET_SVO_SMD  1
    AND  DET_SVOSMD_INV
SVO_POL_KO
    OR Acc DET_SVO_SMD
    COMPI Acc 0
    JUMPCS SAM_SRCH
GC_DET
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG
    COMPI SVO_POL_KO 1
    JUMPCS STATE_END_GC_1
   JUMP STATE_END_GC_0
STATE_END_GC_1
    COMPI ONE_EARLY_GC 0
    JUMPCS GC_DET
    JUMP STATE_END_GC_3
STATE_END_GC_0
    NMI DET_GRAY_RDY 1
    COMPI DET_GRAY_RDY 0
    JUMPCS GC_DET
STATE_END_GC_3
    SET_V UPD_OK_BRT_ITR 14
```

```
BRST_DEM
    NOT  ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
WAIT_TRIGGER
    NMI BRT_TRIGGER 1
    SET_V SEQR_LD_DATA 0
    JUMP BRST_DEM_2
STATE_LD_DATA
    SET_V SEQR_LD_DATA 0
BRST_DEM_2
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG_A
WAIT_RRO1_BRT
    CONCAT BRT_END RRO1_CNT_END
    COMPI Acc 3
    JUMPCS WAIT_RRO1_BRT2
    CONCAT BRT_END RRO1_CNT_END
    COMPI Acc 1
    JUMPCS PGR_RRO1_A
    CONCAT BRT_END RRO1_CNT_END
    COMPI Acc 2
    JUMPCS WAIT_RRO1_CLOSE
    JUMP WAIT_RRO1_BRT
WAIT_RRO1_BRT2
    SET_V SEQR_UPD_GLS 1
    COMPI RD_RRO 0
    JUMPCS CLOSE_SG
    SET_V SEQR_LD_DATA 0
    SET_V SEQR_PGR_START 1
    JUMP PGR_RRO1
WAIT_RRO1_CLOSE
    SET_V SEQR_UPD_GLS 1
    SET_V SEQR_LD_DATA 0
    COMPI RD_RRO 0
    JUMPCS CLOSE_SG
WAIT_RRO1
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_LD_DATA 0
    NMI RRO1_CNT_END 1
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG_B
    SET_V SEQR_PGR_START 1
PGR_RRO1
    NOT  ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_LD_DATA 0
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG_B
    NMI PGR_READY 1
    SET_V ITR_CODE 3
DATA_RRO1
    NOT  ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V ITR_LD 1
DATA_RRO1_A
    CONCAT DET_RRO_VL1
SEQR_SG_SYNC
    COMPI Acc 3
    JUMPCS STATERRO2_CNT_END
    CONCAT DET_RRO_VL1
SEQR_SG_SYNC
    COMPI Acc 2
    JUMPCS CLOSE_SG_C
    CONCAT DET_RRO_VL1
SEQR_SG_SYNC
    COMPI Acc 1
    JUMPCS DATA_PGR_RRO1_2
    CONCAT DET_RRO_VL1
SEQR_SG_SYNC
    COMPI Acc 0
    JUMPCS CLOSE_SG_B
STATERRO2_CNT_END
    COMPI RRO2_CNT_END 0
    JUMPCS WAIT_CLOSE_RRO2
    COMPI RD_2RRO 0
    JUMPCS CLOSE_SG
    SET_V SEQR_PGR_START 1
    JUMP PGR_RRO2
WAIT_CLOSE_RRO2
    COMPNI RD_2RRO 1
    JUMPCS CLOSE_SG
WAIT_RRO2
    NOT  ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_LD_DATA 0
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG_D
    COMPI RRO2_CNT_END 0
    JUMPCS WAIT_RRO2
    SET_V SEQR_PGR_START 1
PGR_RRO2
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_LD_DATA 0
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG_D
    COMPI PGR_READY 0
    JUMPCS PGR_RRO2
    SET_V ITR_CODE 3
DATA_RRO2
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V ITR_LD 1
    SET_V SEQ_ID_RRO 1
    NMI  SEQR_SG_SYNC  0
    JUMPCS DATA_RRO2
    COMPI DET_RRO_VL2 0
    JUMPCS CLOSE_SG_D
CLOSE_SG
```

```
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_RST_LOOPS 1
    COMPI WR_RRO 0
    JUMPCS CLOSE_SG_SEC
    NOT  SVO_SMD_QL_OK
    AND Acc SPI_RRO_STP_QBAD
    STORE t0
    NOT SEQR_SG_SYNC
    AND Acc t0
    OR Acc RRW_END_WRITE
    STORE t0
    NOT  SVO_SPI_SMD_FND
    OR Acc t0
    STORE t0
    CONCAT SEQR_SG
SEQ_NSVO_MINIWB
    CONCAT Acc t0
    COMPI Acc 2
    JUMPCS CLOSE_SG
SVO_SG_END
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_FL_SG 1
SVO_SG3
    SET_V SEQR_LD_DATA 0
SVO_SG4
    SET_V SEQR_LD_DATA 0
    JUMP IDLE
CLOSE_SG_SEC
    NMI SEQR_SG_SYNC  0
    JUMPCS CLOSE_SG
    JUMP SVO_SG_END
DATA_PGR_RRO1_2
    COMPI RRO2_CNT_END  0
    JUMPCS DATA_RRO1
    SET_V SEQR_RRO_EXP1 1
    SET_V SEQR_PGR_START 1
    JUMP PGR_RRO2
PGR_RRO1_A
    SET_V SEQR_UPD_GLS 1
    SET_V SEQR_PGR_START 1
    JUMP PGR_RRO1
CLOSE_SG_A
    SET SEQR_RRO_EXP1 RD_RRO
    SET SEQR_RRO_EXP2 RD_2RRO
CLOSE_SG_E
    SET_V SEQR_UPD_GLS 1
    JUMP CLOSE_SG
CLOSE_SG_B
    SET_V SEQR_RRO_EXP1 1
CLOSE_SG_C
    SET SEQR_RRO_EXP2 RD_2RRO
    JUMP CLOSE_SG
CLOSE_SG_D
    SET_V SEQR_RRO_EXP2 1
    JUMP CLOSE_SG
AZ_WIN
    SET_V SEQR_LD_LOOPS 1
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    COMPI SEQR_SG 0
    JUMPCS SVO_SG_END
    COMPI AZ_FOR_FSM 1
    JUMPCS AZ_WIN
    NOT SEQ_NSVO_MINIWB
    COMPI Acc 0
    JUMPCS WAIT_INPUT
WAIT_INPUT_MWG
    COUNTER INT_CNT_GEN 126
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    NMI SEQR_SG_SYNC 0
    JUMPCS SVO_SG_END
    NMI INT_CNT_GEN_end 2
    COMPI INT_CNT_GEN_end 0
    JUMPCS WAIT_INPUT_MWG
    SET_V SEQR_PGR_START 1
PGR_MWEDGE
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    NMI SEQR_SG_SYNC 0
    JUMPCS SVO_SG_END
    COMPI PGR_READY 0
    JUMPCS PGR_MWEDGE
    SET_V INTF_ITR_CODE 7
SAM_MWEDGE
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_ITR_ON 1
    NMI SEQR_SG_SYNC 0
    JUMPCS CLOSE_SG
    COMPI DET_SVO_SMD 0
    JUMPCS SAM_MWEDGE
    SET_V UPD_OK_BRT_ITR 14
BRST_MWEDGE
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    CONCAT  BRT_END SEQR_SG_SYNC
    COMPI Acc 1
    JUMPCS BRST_MWEDGE
    JUMP CLOSE_SG_E
PRBL_SRCH
    NOT ZONE_CHANGE
    SET SEQR_CKEN_SVO Acc
    SET_V SEQR_AGC_EN 1
    COMPI SEQR_SG_SYNC 0
    JUMPCS PGR_END
    COMPI SPN_PBL_FOUND 0
    JUMPCS PRBL_PGR
```

```
        SET_V SEQR_PGR_START 1
        SET SEQ_UPDATE_LOOP
ACQ_COUNT_EN
        COMPI SPI_PGR_WIN_SEL 1
        JUMPCS STATE_CNT_SPN
        COUNTER INT_CNT_GEN
SPN_PGR_MIN
        JUMP PGR_SPN_CHK
DC_ERASE_SRCH
        COUNTER INT_CNT_GEN
LAT_DC_PD
        NOT ZONE_CHANGE
        SET SEQR_CKEN_SVO Acc
        SET t0 LAT_DC_PD
        SET_V START_SRST_AGC 1
        NMI SEQR_SG_SYNC 0
        JUMPCS SVO_SG_END
        COMPI SPN_DC_FOUND 0
        JUMPCS DC_ERASE_SRCH
WAIT_DC_PD
        COUNTER INT_CNT_WT 126
        NOT ZONE_CHANGE
        SET SEQR_CKEN_SVO Acc
        SET_V SEQR_AGC_EN 1
        NMI SEQR_SG_SYNC 0
        JUMPCS PGR_END
        NMI INT_CNT_GEN_end 1
        COMPI INT_CNT_GEN_end 0
        JUMPCS DC_PD_PGR
        COMPI SPN_PBL_FOUND 1
        JUMPCS STATE_PGR_SPN_CHK
        SET_V SEQR_PGR_SRST 0
        JUMP DC_ERASE_SRCH
DC_PD_PGR
        SET_V SEQR_PGR_SRST 0
        JUMP WAIT_DC_PD
STATE_PGR_SPN_CHK
        SET_V START_SRST 3
        COMPI SPI_PGR_WIN_SEL 1
        JUMPCS STATE_CNT_SPN
        COUNTER INT_CNT_GEN
SPN_PGR_MIN
PGR_SPN_CHK
        COUNTER INT_CNT_GEN 126
        NOT ZONE_CHANGE
        SET SEQR_CKEN_SVO Acc
        SET_V SEQR_AGC_EN 1
        NMI SEQR_SG_SYNC  0
        JUMPCS PGR_CLOSE_SG
        COMPI PGR_READY 1
        JUMPCS UPDATE_LOOP
        NMI INT_CNT_GEN_end 1
        CONCAT PBL_FND_D
INT_CNT_GEN_end
        COMPI Acc 0
        JUMPCS NO_UPDATE_LOOP
        SET t0 INT_CNT_GEN
        CONCAT PBL_FND_D
INT_CNT_GEN_end
        COMPI Acc 6
        JUMPCS STATE_FF
        JUMP PGR_SPN_CHK
PGR_CLOSE_SG
        SET_V SEQR_PGR_SRST 0
        JUMP CLOSE_SG
UPDATE_LOOP
        CONCAT PBL_FND_D
INT_CNT_GEN_end
        COMPI Acc 6
        JUMPCS STATE_WT_SAM_SRCH
        SET_V SEQ_UPDATE_LOOP 1
NO_UPDATE_LOOP
        SET_V SEQR_PGR_SRST 0
        COMPI SPI_SVO_DC_EN 1
        JUMPCS DC_ERASE_SRCH
        NOP 1
        JUMP PRBL_SRCH
STATE_FF
        COUNTER INT_CNT_GEN t0
        JUMP PGR_SPN_CHK
STATE_WT_SAM_SRCH
        SET_V INTF_ITR_CODE 7
        SLL SPI_SVO_PD_SAFE 3
        COUNTER INT_CNT_GEN Acc
        SET_V PREAMBLE_CNT_EN 1
WT_SAM_SRCH_SPN
        COUNTER INT_CNT_GEN 126
        NOT ZONE_CHANGE
        SET SEQR_CKEN_SVO Acc
        SET_V SEQR_ITR_ON 1
        SET_V SEQR_AGC_EN 1
        NMI SEQR_SG_SYNC  0
        JUMPCS CLOSE_SG
        COMPI PD_SAFE_LOST 0
        JUMPCS STATE_PD_SAFE
STATE_DEFAULT_PD_SAFE
        SET_V UPDATE_SRST 2
        COMPI SPI_SVO_DC_EN 1
        JUMPCS DC_ERASE_SRCH
        NOP 1
        JUMP PRBL_SRCH
STATE_PD_SAFE
        CONCAT SAM_SRCH_STRT_D
PREAMBLE_SAFE
        COMPI Acc 3
        JUMPCS CNT_SAM
        CONCAT SAM_SRCH_STRT_D
PREAMBLE_SAFE
        COMPI Acc 2
        JUMPCS STATE_DEFAULT_PD_SAFE
```

```
    SET_V PREAMBLE_CNT_EN 1
    JUMP WT_SAM_SRCH_SPN
CNT_SAM
    SET_V PREAMBLE_CNT_EN 0
    JUMP SAM_SRCH
STATE_SVO_POL
    COMPI SVO_POL_KO 1
    JUMPCS CLOSE_SG
    COMPI SMD_EXP_FLAG 0
    JUMPCS STATE_SAM_GC
    SET_V UPDATE_SRST_AGC 5
    COMPI SPI_SVO_DC_EN 1
    JUMPCS DC_ERASE_SRCH
    NOP 1
    JUMP PRBL_SRCH
STATE_SAM_GC
    CONCAT SMD_DET_FLAG
SMD_QUAL_FLAG
    COMPI Acc 3
    JUMPCS GC_DET
    SET_V SEQR_AGC_EN 1
    JUMP SAM_SRCH
PGR_END
    SET_V SEQR_PGR_SRST 0
    JUMP SVO_SG_END
PRBL_PGR
    SET_V SEQR_PGR_SRST 0
    JUMP PRBL_SRCH
STATE_CNT_SPN
    COUNTER INT_CNT_GEN
SPN_PGR_MAX
    JUMP PGR_SPN_CHK
#
```