Diploma Thesis

SYSTEM VERIFICATION OF UMTS BASEBAND FUNCTIONS WITH HIGH LEVEL MODELS

realized at the
UNIVERSITY OF TECHNOLOGY IN VIENNA
Faculty of Electrical Engineering
Institute of Communications and Radio Frequency Engineering

Martin Allram
Matrikelnummer 9525968
Blumengasse 11, 2751 Steinabrückl

Supervisor:
Dipl.-Ing. Thomas Herndl
Infineon Technologies MDCA GmbH
Design Center Vienna, Austria

Educational supervisor:
Univ.-Prof. Dipl.-Ing. Dr.-Ing. Markus Rupp

Vienna, February 2003

Allram Martin .....

UNIVERSITY OF TECHNOLOGY IN VIENNA
Faculty of Electrical Engineering
Institute of Communications and Radio Frequency Engineering

SYSTEM VERIFICATION OF UMTS BASEBAND FUNCTIONS WITH HIGH LEVEL MODELS

Diploma Thesis

Martin Allram
Matrikelnummer 9525968
Blumengasse 11, 2751 Steinabrückl

Supervisor:
Dipl.-Ing. Thomas Herndl

Educational supervisor:
Univ.-Prof. Dipl.-Ing. Dr.-Ing. Markus Rupp

Abstract

This diploma thesis presents implementation of the UMTS synchronization functions at different levels of abstraction. The newest system level design tool from Synopsys, CoCentric System Studio Release 2002.05 supports SystemC versions 1.0 and 2.0 was used for this task. The reference design is the slot synchronization, which is a part of the UMTS cell search synchronization procedure. Parts of the reference design will be implemented in hardware while some other parts will run in software (firmware) on a dedicated processor (DSP).

Functional abstraction level and transaction level modeling of UMTS slot synchronization were designed, simulated and verified via testvectors. SystemC 2.0 was used to describe the UMTS slot synchronization at both levels of abstraction.

The tool CoCentric System Studio, release 2002.05 was used to implement and simulate the reference design. At the abstraction level, transaction level modeling, the hardware part of the reference design is written in SystemC 1.0 (RTL code). The software parts run on an instruction set simulator for an ARM926 DSP core.

TECHNISCHE UNIVERSITÄT WIEN
Fakultät für Elektrotechnik
Institut für Nachrichtentechnik und Hochfrequenztechnik

SYSTEM VERIFIKATION VON UMTS BASISBAND FUNKTIONEN MIT "HIGH LEVEL MODELS"

Diplomarbeit

Martin Allram
Matrikelnummer 9525968
Blumengasse 11, 2751 Steinabrückl

Betreuer:
Dipl.-Ing. Thomas Herndl

Betreuer an der TU-Wien:
Univ.-Prof. Dipl.-Ing. Dr.-Ing. Markus Rupp

Zusammenfassung

In dieser Diplomarbeit wurde die Implementierung von UMTS Synchronisationsfunkionen auf verschiedene Abstraktionesbenen durchgeführt. Das neue Systemdesign Entwicklungswerkzeug von Sysnopsys, CoCentric System Studio Version 2002.05, machte es möglich SystemC 1.0 und System 2.0 zu verwenden und war so bestens für die Aufgabenstellung geeignet. Die Slot Synchronisation, ein Teil der UMTS Zellensuche und Synchronisation Prozedur, ist als Referenzimplementierung gewählt worden. Teile dieses Referenzdesigns sollten in Hardware realisiert werden, während andere Teile als Software auf einem spezifisierten Prozessor (DSP) implementiert werden.

Die UMTS Slot Synchronisation wurde auf den Abstraktionsebenen funktionale Beschreibung und die transaktionsbasierende Modellierung realisiert, simuliert und anhand von Simulationsergebnissen auf ihre Korrektheit überprüft. Als gewählte Beschreibungssprache für beide Abstraktionebenen ist SystemC 2.0 verwendet worden.

Das Entwicklungswerkzeug CoCentric System Studio, Version 2002.05, wurde als graphisches Designwerkzeug benutzt um das Referenzdesign zu implementieren und zusammen mit Testvektoren zu simulieren. Auf der Abstraktionsebene, transaktionsbasierende Modellierung, sind Teile der UMTS slot synchronisation in Hardware (SystemC 1.0 RTL) und in Teile in Software, welche auf dem ARM926 Prozessorkern-Simulator, ablaufen.

I hereby certify that the work reported in this diploma thesis is my own, and the work done by other authors is appropriately cited.

Martin Allram
Vienna, February 28, 2003

Acknowledgments

I would like to thank Dipl.-Ing. Thomas Herndl, my supervisor at Infineon Technologies MDCA DC A VIE, for helping and guiding me through my work on this diploma thesis.
I would also like to thank Dipl.-Ing. Wolfgang Haas at Infineon Technologies MDCA DC A VIE for his valuable advise and help.
I would like to thank Univ.-Prof. Dipl.-Ing. Dr.-Ing. Markus Rupp, my educational supervisor at Institute of Communications and Radio Frequency Engineering at Univeristy of Technology in Vienna, for help and encouragement.
I want to thank my girlfriend Kathrin Teubl for pushing me to work hard, supporting me and for helping me a lot.

Martin Allram
Vienna, February 28, 2003

Table of Contents

List of Abbreviations

3GPP	3rd generation partnership project
ADC	Analog to digital converter
AICH	Acquisition indication channel
AHB	Advanced high-performance bus
AMBA	Advanced microcontroller bus architecture
AMR	Adaptive multirate(speech codecs)
APB	Advanced peripheral bus
ASIC	Application specific integrated circuits
AWGN	Additive white gaussian noise
BCH	Broadcast channel
BCCH	Broadcast control channel
BS	Base station
BTS	Base transceiver station
CCSS	CoCentric System Studio
CCCH	Common control channel
CDMA	Code division multiple access
CPCH	Common packet channel
CPU	Central processing unit
CPICH	Common pilot channel
DCH	Dedicated channel
DCCH	Dedicated control channel
DMA	Direct memory access
DPCH	Dedicated physical channel
DPCCH	Dedicated physical control channel
DPDCH	Dedicated physical data channel
DSCH	Downlink shared channel
DTCH	Dedicated traffic channel
DS-CDMA	Direct spread code division multiple access
DSP	Digital signal processor
EDA	Electronic design automation

FACH	Forward access channel
FDD	Frequency division duplex
FSM	Finite state machine
GUI	Graphical user interface
GSM	Global system for mobile communication
HDL	Hardware description language
HW/SW	Hardware/Software
IC	Integrated circuit
IMC	Interface method call
IMT	International mobile telephony
IP	Intellectual property
ITU	International telecommunications network
kbps	Kilobits per second
Mbps	Megabits per second
MC	Micro controller
MRC	Maximum ratio combining
MMU	Memory management unit
MS	Mobile station
OVSF	Orthogonal variable spreading factor
PCH	Paging channel
PCCH	Paging control channel
P-CCPCH	Primary common control physical channel
PCPCH	Physical common packet channel
PCI	Peripheral component interconnect
PDC	Personal digital cellular
PDSCH	Physical downlink shared channel
PI	Paging indicator
PICH	Page indication channel
PRACH	Physical random access channel
PSC	Primary synchronization code
P-SCH	Primary synchronization channel
PUSCH	Physical uplink shared channel
QoS	Quality of Service
OSCI	Open SystemC initiative
RACH	Random access channel
RISC	Reduced instruction set core
RTL	Register transfer level
RTOS	Real time operating system

SCH	Synchronization channel
S-CCPCH	Secondary common control physical channel
SDS	Stream driven simulation
SDT	Standard Deviation
SMS	Short message service
SoC	System on chip
S-SCH	Secondary synchronization channel
TCH	Traffic channel
TCM	Tightly coupled memory
TDD	Time division duplex
TDMA	Time division multiple access
TLM	Transaction level modeling
TS	Time slot
UE	User equipment
UMTS	Universal mobile telecommunications system
USCH	Uplink shared channel
UTF	Untimed functional
WARC	World administrative radio conference
WCDMA	Wideband code division multiple access

List of Figures

List of Tables

Chapter 1

Introduction

Advances in silicon processing technology enable integration of ever more complex systems on a single chip. This leads to more and more complex integrated circuits. The so called systems-on-a-chip (SoC) contain dedicated hardware components, programmable processors, memories, etc. requiring not only the design of digital hardware, but also the design of embedded software. The successful deployment of these systems requires very high design productivity to deal with the immense complexity of the system with limited design resources.

In today's system level design flows it has become commonplace to describe the pure functional system level model of hardware building blocks in a programming language such as C/C++. Once hardware related concepts need to be expressed though, the design is usually transferred into a hardware description language (HDL) based design environment. The reason for this is that C/C++ by itself does not provide the necessary mechanism to describe concepts like concurrency, signals, reactivity, and hardware data-types being inherent to hardware. Usually, following problems exist with this approach[Sysb]:

- Error prone manual conversion from C/C++ to HDL.
  With the current methodology, a designer creates the C/C++ model, verifies that the C/C++ model works as expected and then translates the design manually into an HDL (VHDL or Verilog) description. This process is very tedious and error prone.

- Disconnection between system model and HDL model.
  After the model is converted to HDL, the HDL model becomes the focus of development. The C/C++ system level model quickly becomes out of date as changes are made. Typically, changes are made only to the HDL model and the updates are not implemented in the C/C++ system level model.

- Multiple system test-benches.
  Tests being created to validate the C/C++ system level model functions typically cannot be used for the HDL model without conversion. Not only does the designer have to convert the C/C++ system level model to HDL, but also the test-bench has to be converted to the HDL environment to have consistent test sequences.

The software parts of the original system level model have to be rewritten with calls to a Real Time Operating System (RTOS). This software model is simulated and verified with an RTOS emulator. Though parts of the original code can be reused, the change in abstraction from the original model to the RTOS based model requires significant manual recoding and verification. Such changes become a significant problem.

To deal with the increasing complexity of SoC there are some ideas to improve the current design methodology[Fit][DV]. The most promising idea is the common language approach for hardware design at all abstraction levels and software design. The main reason for such design methodology is the possibility to improve the path between different parts of the design process. In a common language approach the design engineers only have to learn one language instead of multiple languages. This will promote communication between different disciplines and abstraction levels. Another approach is the extensions to existing languages. There can be syntax extension or additional class libraries. In case of the additional libraries they can only be used with extensible languages such as C++ or JAVA.

The most known and used common language approach, which is an extension of C++, is SystemC. SystemC has been developed and is supported by an industry consortium know as the Open SystemC Initiative (OSCI). It consists of a number of major Electronic Design Automation (EDA) companies making SystemC very attractive for designers.

The aim of this diploma thesis is the implementation of the synchronization function for a UMTS baseband chip used in a UMTS handset. For this task the C-based system level design tool CoCentric System Studio is applied, supporting SystemC version 1.0 and SystemC version 2.0. An available slot synchronization part of the UMTS cell search synchronization procedure is used as a reference design. Some parts of UMTS slot synchronization will be implemented in hardware while others will be implemented in software. The reference design was designed in SystemC version 2.0, verified and simulated at the functional abstraction level and at the abstraction level of transaction level modeling within CoCentric System Studio. The functional abstraction level describes a

data driven approach of the reference design, while at the abstraction level of transaction level modeling, interfaces and channels of the reference design are described accurately. The different levels of abstraction can be seen in Figure 1.1. An example of the functional abstraction level of graphical SystemC description is shown in Figure 4.7.

Chapter 2 gives an overview on UMTS. SystemC is explained in more de-

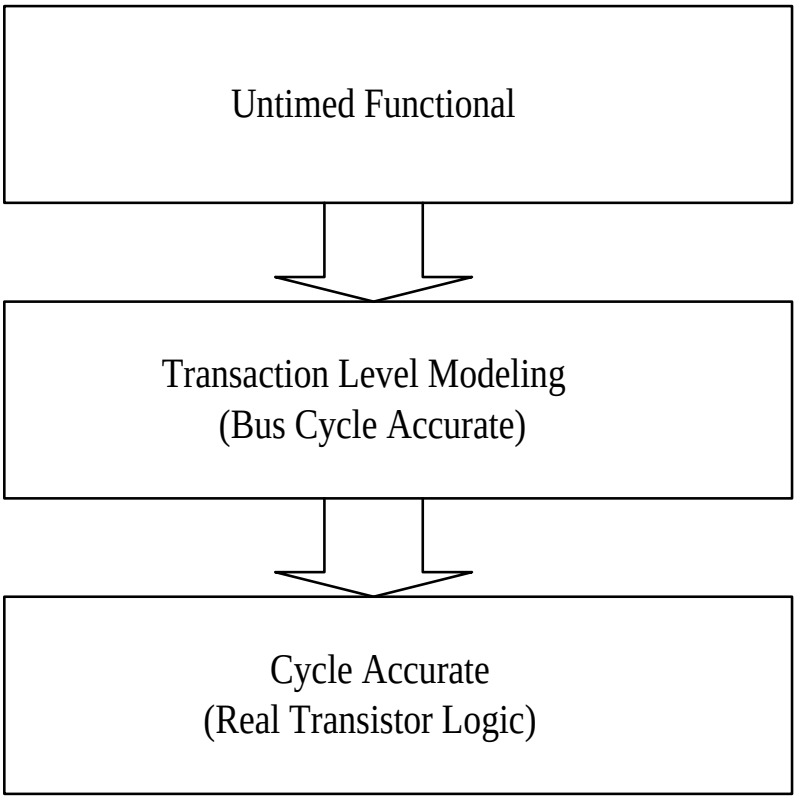

Figure 1.1: Different levels of abstraction

tail in Chapter 3. In Chapter 4, the algorithmic description of the UMTS slot synchronization procedure with preselection from primitive models in CoCentric System Studio was translated to SystemC 2.0. All hierachical blocks of the testbench were also ported from the primitive models to SystemC 2.0. The verification of each ported block was done via output files. The results of algorithmic description of the reference design in the primitive language are compared with the results of the functional description of the reference design in SystemC 2.0 and the designs were found to be identical.

Finally in Chapter 5, the functional description of the reference design in SystemC 2.0 was mapped to existing dedicated hardware (ARM 926 EJ-S and AMBA). The hardware parts of the reference design available in SystemC ver-

sion 1.0 were imported to CoCentric System Studio. The software part of the reference were ported from the primitive model to SystemC 2.0 and then translated to the specific processor, an ARM 926 EJ-S.

Chapter 2

UMTS

2.1 Common aspects of UMTS

Global System for Mobile communication (GSM), a second generation telecommunication system, enabled voice traffic to go wireless [UMT]. The number of mobile phones exceeds the number of landline phones and mobile penetration exceeds 70 percent in countries with most advanced wireless markets. In Austria, mobile penetration exceeds 80 percent. The data handling capabilities of the second generation systems are limited. Third generation systems need to provide the high bit rate services that will enable high quality images and video to be transmitted and received. Also access to the web with high data rates will be available. This third generation mobile communication system is called UMTS (Universal Mobile Telecommunications Systems). UMTS is the main third generation air interface and will be deployed at a frequency band around 2GHz.

2.2 Introduction to UMTS

2.2.1 Third generation systems

The first generation systems were analog cellular systems. The digital systems which are currently in use are GSM, PDC, CdmaOne (IS-95) and US-TDMA (IS-136). They are known as second generation systems. These systems have enabled voice communications to go wireless in many leading markets and customers are increasingly also finding value in other services, such as text messaging and access to data networks, which are starting to grow rapidly.

Multimedia communication is expected to become the main application of third generation systems. Person to person communications can be enhanced

with high quality images and video, and access to information and services on public or private networks will be supported by the higher data rates. These are new flexible communication capabilities of third generation systems.

UMTS technology has emerged as the most widely adopted third generation air interface. The specification of UMTS has been created in 3GPP (the 3rd Generation Partnership Project) which is the joint standardization project of the standardization bodies from Europe, Japan, Korea, the USA and China. UMTS is called UTRA (Universal Terrestrial Radio Access), FDD (Frequency Division Duplex) and TDD (Time Division Duplex), the name UMTS being used to cover both FDD and TDD operations. Some important documents used throughout this thesis of the 3GPP standard are shown in Table 2.1

standard	description
3GPP TS 25.201	Physical layer(Channelisation codes, Scrambling codes, etc.) - general description
3GPP TS 25.211	Physical channels and mapping of transport channels onto physical channels (FDD)
3GPP TS 25.212	Multiplexing and channel coding (FDD)
3GPP TS 25.213	Spreading and modulation (FDD)
3GPP TS 25.214	Physical layer procedures (FDD)
3GPP TS 25.215	Physical layer - Measurements (FDD)
3GPP TS 25.221	Transport channels and physical channels (TDD)
3GPP TS 25.222	Multiplexing and channel coding (TDD)
3GPP TS 25.223	Spreading and modulation (TDD)
3GPP TS 25.224	Physical layer procedures (TDD)
3GPP TS 25.225	Physical layer - Measurements (TDD)
3GPP TR 25.944	Channel coding und multiplexing examples

Table 2.1: UMTS Standard Documents

2.2.2 Spectrum allocation for third generation systems

The development of third generation mobile systems started at the World Administrative Radio Conference (WARC) of the ITU (International Telecommunications Union) (1992 meeting) when frequencies around 2 GHz were identified to be used by future third generation systems, terrestrial and satellite. Within the ITU these third generation systems are called International Mobile Telephony 2000(IMT-2000) [HT00]. Within the IMT-2000 framework, several different air interfaces are defined for third generation systems, based on either CDMA or TDMA technology. Additional to the UMTS, other air interfaces can be used to provide third generation services: EDGE and multicarrier CDMA (cdma2000). EDGE (Enhanced Data Rates for GSM Evolution) can provide

third generation services with bit rates up to 500 kbps within a GSM carrier spacing of 200 kHz.

The allocation of the radio spectrum in Europe, Japan, Korea, an the USA is shown in Figure 2.1. In most of Asia and in Europe the IMT-2000 bands of 2x60MHz (1920-1980 MHz plus 2110-2170 MHz) are allocated for UMTS FDD. The availability of the TDD spectrum varies from region to region. In Europe 25 MHz is allocated for licensed TDD use in the 1900-1920 MHz and 2020-2025 MHz bands. The rest of the spectrum will be used for unlicensed TDD applications (SPA: Self Provided Applications) in the 2010-2020 MHz band. Different frequency bands are used for the FDD uplink and FDD downlink, separated by the duplex distance, while TDD systems utilize the same frequency for both the uplink and the downlink. The WARC-2000 was looking for additional 160 MHz spectrum on top of the currently available second generation and third generation frequency ranges. New frequencies were allocated, which makes global roaming possible and this is very beneficial for a real third generation mass market. Also an extension of the UMTS to a lower frequency band is done to deploy third generation system services in rural areas. The additional bands identified for WCDMA terrestrial components are: 806-960 MHz, 1710-1885 MHz, and 2500-2690 MHz. [1]

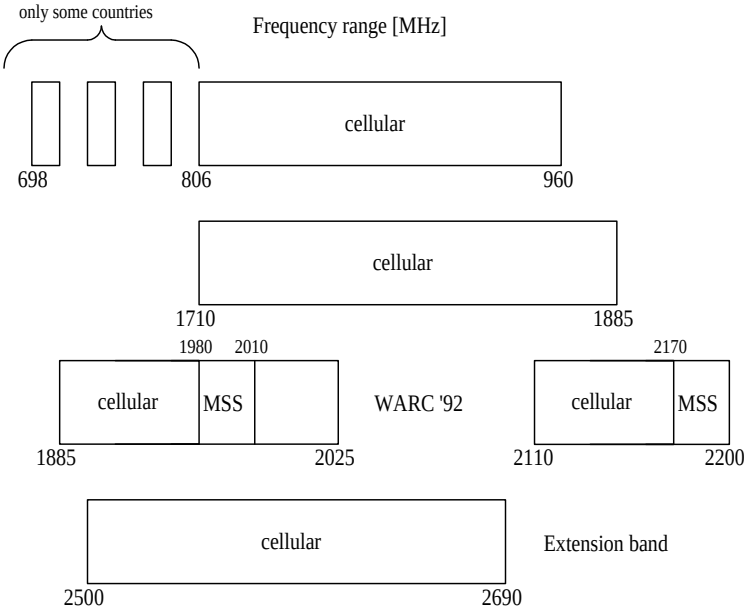

Figure 2.1: Spectrum allocation for UMTS

[1] MSS (Mobile Satellite Systems).

2.2.3 Differences between (second and third generation) air interfaces

Main differences between UMTS and the second generation air interface are described in this section. GSM and IS-95 (the standard for CdmaOne systems) have second generation air interfaces. Other second generation air interfaces are PDC in Japan and US-TDMA mainly in America; these are based on TDMA (Time Division Multiple Access) and have more similarities with GSM than with IS-95. Speech services in macro cells are the most used application for the second generation systems[HT00]. To understand the background to the differences between second and third generation systems, the new requirements of the third generation systems needed to be looked at as shown in Table 2.2.

- Bit rates up to 2Mbps
- Variable bit rate to offer bandwidth on demand
- Multiplexing of services with different quality requirements on a single connection, e.g. speech, video and packet data
- Delay requirements from delay-sensitive real-time flexible best-effort packet data
- Quality requirements from 10 percent frame error rate to 1×10^{-6} bit error rate
- Coexistence of second and third generation systems and the inter-system handovers for coverage enhancements and load balancing
- Support for asymmetric uplink and downlink traffic e.g. web browsing causes more loading to downlink than to uplink
- High spectrum efficiency
- Coexistence of FDD and TDD modes

Table 2.2: New requirements of third generation systems

Table 2.3 lists the main differences between UMTS and GSM. Only the air interface of UMTS and GSM are considered[HT00].

The new requirements are reflected in the air interface differences of both systems. Larger bandwidth, for example, is needed to support higher bit rates. The improvement of downlink capacity to support asymmetric capacity requirements needs an inclusion of transmit diversity in third generation systems. Transmit diversity is not supported by GSM. Different bit rates, services and quality requirements in UMTS requires advanced radio resource management algorithms to guarantee quality of service and to maximize system throughput. Efficient support of non real-time packet data is important for new services.

	UMTS	GSM
Carrier spacing	5 MHz	200 kHz
Frequency reuse factor	1	1-18
Power control frequency	1500 Hz	2 Hz or lower
Quality control	Radio resource management algorithms	Network planning (frequency planning)
Frequency diversity	5 MHz bandwidth allows multipath diversity with Rake receiver	Frequency hoping
Packet data	Load based packet scheduling	Time slot based scheduling with GPRS
Downlink transmit diversity	Supported for improving downlink capacity	Not supported by the standard

Table 2.3: Main differences between UMTS and GSM air interfaces

2.3 UMTS services and applications

The best known new feature of UMTS is higher user bit rates : circuit switched connections have 384 kbps, and packet switched connections supports bit rates up to 2 Mbps. Higher bit rates facilitate new services, such as video telephony and quick download of data.

At the start of the third generation system era, almost all traffic is voice but data traffic is expected to increase with time.

UMTS provides a new and important feature compared to GSM and other existing mobile networks, namely it allows negotiations of the properties of a radio bearer. Attributes defining the characteristics of the transfer may include throughput, transfer delay and data error rate. UMTS has to support a wide range of applications posing different quality of services (QoS) requirements.

2.3.1 UMTS bearer service

The negotiating process for bearer characteristics in UMTS allows a user to choose the most appropriate application for carrying information. It is also possible to change bearer properties via a bearer renegotiation procedure. Bearer negotiation is initiated by an application, while renegotiation may be initiated either by the application or by the network.

2.3.2 UMTS quality of service (QoS) classes

Applications and services can be divided into different groups, depending on how they are considered. Like new packet-switched protocols UMTS attempts

to fulfill QoS requests from applications or users. In UMTS four traffic classes have been identified[UMT]:

- Conversational class
  The best known service, the conversational class, is the speech service over circuit-switched bearers. With the usage of Internet and multimedia, a number of new applications will require this type, for example, voice over IP and video telephony. Real-time conversation is characterized by the fact that the end-to-end delay is low and the traffic is symmetric or nearly symmetric. The maximum end-to-end delay is given by the human perception of video and audio conversation(Adaptive Multi-rate(AMR), video telephony).

- Streaming class
  Multimedia streaming considers data transfers in steady and continuous streams. Streaming technologies are becoming increasingly important with the growth of the Internet, because most users do not have sufficiently fast access to download large multimedia files quickly. With streaming, the client browser or plug-in can start displaying data before the entire file has been transmitted.

- Interactive class
  If the end-user (machine or human) is on-line, requesting data from remote equipment (e.g. server), the interactive class applies. Examples of human interaction with the remote equipment are web browsing, server access, and other applications. Interactive traffic is the other classical data communication scheme broadly characterized by the request-response pattern of the end-user.

- Background class
  Applications with data traffic such as email delivery, SMS and download of databases can be delivered in the background, since such applications do not require immediate actions. The delay may be seconds or even minutes. Electronic postcards are one typical example of such class.

2.4 UMTS

In this chapter, the air interface of UMTS is described. The main parameters, the two modes of UMTS, the UMTS timing structure, the concept of spreading and despreading, the multipath radio channel, the rake receiver, and the power control is characterized.

2.4.1 Main parameters of UMTS

The main system design parameters of UMTS are summarized in Table 2.4
Here are some aspects that characterize UMTS[UMT]:

- UMTS is a wideband Direct-Sequence Code Division Multiple Access (DS-CDMA) system.
  This means user information is spread over a wide bandwidth by multiplying the data with quasi random bits (called chips) derived from CDMA spreading codes. Variable spreading factors and multi-code connections are supported.

- The chip rate of 3.84 Mbps together with a rolloff of 22% leads to a bandwidth of 5 MHz.
  The wide carrier bandwidth of UMTS supports high user data rates and offers performance benefits like increased multipath diversity. To increase capacity, multiple carriers can be deployed.

- Variable user data rates are supported by UMTS.
  Each user is allocated frames of 10 ms duration, during which the user data rate is constant, allowing user data rates to vary from frame to frame.

- Two basic modes of UMTS are supported.
  The two transmission modes of UMTS are Frequency Division Duplex (FDD) and Time Division Duplex (TDD). In the FDD mode carrier frequencies multiple of 5 MHz are used for the uplink and for the downlink. In the TDD mode a single 5 MHz bandwidth is time shared between the uplink and the downlink.

- UMTS supports the operation of asynchronous base stations.

- Coherent detection for uplink and downlink is employed in UMTS.
  For this task pilot symbols or a common pilot is used. This will result in increased coverage and capacity for the uplink and the downlink.

- Multiuser detection and smart adaptive antennas can be deployed
  by the network operator as a system option to increase capacity and coverage.

- UMTS is to be deployed in conjunction with GSM.
  Handovers between both systems are supported.

Multiple Access Method	DS-CDMA
Duplexing method	frequency divison duplex/time divison duplex
Base station synchronization	Asynchronous operation
Chip rate	3,84 Mbps
Frame length	10 ms
Service multiplexing	Multiple services with different quality of service requirements multiplexed over one connection
Multirate concept	Variable spreading factor and multicode
Detection	Coherent using pilot symbols or a common pilot
Multiuser detection, smart antennas	Supported by the standard

Table 2.4: UMTS parameters

2.4.2 Two modes of UMTS

UMTS offers flexible and dynamic variation of data rate between several kbps to 2 Mbps. The variation of data rates are different in the FDD and TDD mode.

- FDD Mode
  The variation of data rate in the FDD mode can be achieved by variation of the spreading factor (described further ahead) and also with allocation of more than one code. The spreading factor for the uplink can vary between 4 and 256 and for the downlink between 4 and 512. Therefore, the corresponding bit rates range from 15 kbps (uplink and downlink) to 960 kbps (uplink) or 1920 kbps (downlink). These rates include inband signaling and redundancy.

- TDD Mode
  The data rate variation in the TDD mode can be achieved by selecting the spreading factor, channel combining (combining of several time slots to one user) and by allocating more than one code. In the TDD mode, the spreading factor can vary from 1 to 16. Thus, the corresponding bit rates range from 32 kpbs to 512 kbps in one time slot (TS) in uplink or downlink. Higher data rates are possible by combining serveral TS for one user. The TDD mode is optimized for asymmetric data volume between uplink and downlink. There is no fixed allocation of time slots for uplink and downlink. At least two time slots must be allocated to uplink or downlink so that the network operator can optimize the usage of radio

resources flexibly for different asymmetric data volumes in different areas.

In FDD and TDD mode higher data rates can be achieved by the allocation of more than one code to one user (as long as the user equipment supports this feature). Allocation of several codes to one user can have advantages due to finer granularity of the data rates. In this case different simultaneous applications of one user are supported by different codes.

2.4.3 UMTS timing structure

UMTS has five different time units: chip, symbol, time slot, frame, superframe. The figure 2.2 shows how the time structure of UMTS looks like[UMT].

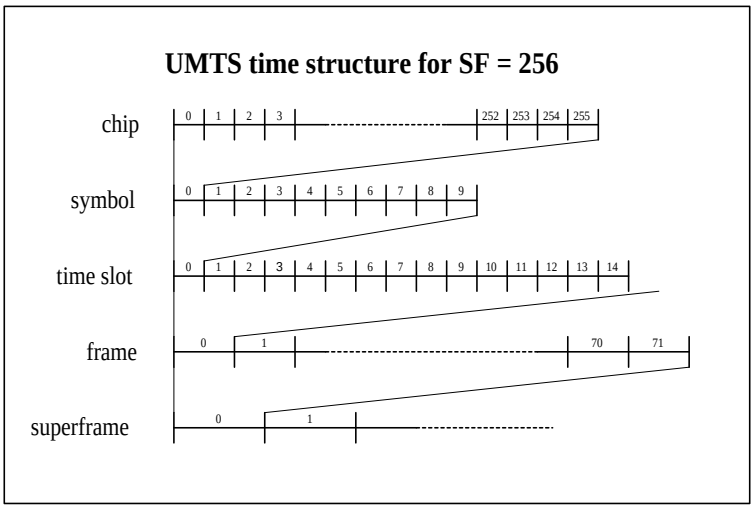

Figure 2.2: UMTS timing structure

- Chip
  The shortest time frame in UMTS is the duration of one chip. In UMTS a chip rate of 3.84 Mchip/s is used. This means the duration of one chip is 260.4 ns.

- Symbol
  The shortest duration of the data bit multiplied with the spreading sequence is called symbol.

- Time slot
  A UMTS time slot is the shortest repetition cyle in UMTS and is defined by the duration of exactly 2560 chips. Thus, the duration of one time slot is 2/3 ms. For TDD, the time slot and its duration defines one HF

burst. For FDD, the time slot is a repetition cycle for inband control information between the user equipment and the network. One example for this control information is the transmit power control.

- Frame
  A UMTS frame is defined by a duration of 10 ms consisting of 15 time slots. In TDD, the frame is the duration for one TDMA frame (repetition cycle for 15 time slots). In FDD, the frame is the shortest possible transmission duration. Short data packets (SMS, Random Access, etc.) have a minimum length of 10 ms. UMTS offers not only flexible, but dynamic data rates. One frame defines the shortest time for an adaption of the data rate.

- Superframe
  One UMTS superframe is defined by the repetition of 72 UMTS frames (720 ms). It is exactly six times longer than the GSM TCH (Traffic Channel) multiframe. Therefore, an adaptation of numbering schemes between UMTS and GSM is possible. This is a prerequisite for handover between GSM and UMTS.

2.4.4 Concept of spreading and despreading

The spreading operation is the multiplication of each two user data bits with a spreading code.

The spreading code of UMTS is the multiplication of the user data with two codes, of different types and properties: the channelisation code and the scrambling code. An overview of channelisation and scrambling codes is offered in 3G standard documents [25.01a]; details are presented in [25.01c] and [25.01f].

- Channelisation code
  Channelisation codes are used to separate transmission channels from signals coming from the same source. For the downlink this is a separation of different users or furthermore, also different applications of these different users by the Base Transceiver Station (BTS). The downlink channelisation codes are allocated to different users/applications. For the uplink this is a separation of different applications simultaneously handled by one user equipment. Up to six different applications can be handled simultaneously by one FDD user equipment. Furthermore, in the FDD uplink the in-band control information from user equipment to the network is handled with different channelisation codes than in the user applications. The channelisation codes for FDD and TDD are based on the Orthogonal

Variable Spreading Factor (OVSF) codes. These codes have orthogonal properties.

- Scrambling codes
  Scrambling codes are used to separate different sources. For the downlink this results in the separation of different BTS. A fixed scrambling code is allocated to every cell. A user equipment can separate between different BTS by different scrambling codes of the BTS. For the uplink, the scrambling codes are used to separate between the different user equipments of one cell. These special codes are allocated to the users by the network provider. FDD and TDD uses different scrambling codes. In the FDD mode so-called "Gold Codes" are used as basis for scrambling codes. Further details on this can be found in the 3G standard document [25.01c]. These codes are reduced to a length of 10 ms (1 frame). Thus, the FDD scrambling code has a length of 38400 chips. In the TDD mode 16 chip long sequences are used as scrambling codes. Further details can be found in the 3G standard document [25.01f].

The channelisation codes of the UMTS FDD and TDD mode are based on the OVSF codes. Different data rates are achieved at constant chip rate with a different spreading factor. The channelisation codes are generated similar to Walsh-matrices. The (1×1) start matrix has the value "1" and is used as channelisation code with spreading factor one (H_1). All further matrices are successively generated by combining three lower range matrices (H_i) with the same value (top right and bottom left) and one lower range matrix with the negative value (bottom right)$(-H_i)$. The channelisation codes of length n (spreading factor n) are generated by columns of an $(n \times n)$ matrix.

$$H_1 = \begin{bmatrix} 1 \end{bmatrix}$$

$$H_i + 1 = \begin{bmatrix} H_i & H_i \\ H_i & -H_i \end{bmatrix}$$

A code tree is generated if all channelisation codes of a certain length (in example spreading factors 1, 2, 4, 8, 16, ..., 256, 512) are orthogonal to each other. For a spreading factor of 256 in UMTS there exits 256 different, orthogonal codes. In the FDD mode this is equivalent to 256 different physical channels with 15 ksymb/s. For a spreading factor of four in UMTS there exists four different, orthogonal codes. In the FDD and TDD mode theoretically four different physical channels exist with 960 ksmyb/s (FDD mode) and 64

ksymb/s per time slot (TDD mode), respectively.

2.4.5 UMTS multipath advantage and RAKE receiver

Multiple reflections, diffractions and attenuations of the signal energy characterize the radio propagation in the land mobile channel[Goi99a][Goi99b][HT00]. Natural obstacles such as buildings, hills, mountains, houses, and so on are the cause for these effects, typically called multipath propagation. A UMTS signal is well matched to the problems occuring under multipath environments. The user equipment receiver receives several copies of the signal with different delays. If the signal arrives more than one chip apart from each other, the receiver can resolve them. The higher the chip rate, the more resolveable is the path. From the multipath signal's point of view, the other multipath signals can be regarded as (undesired) interference and they are suppressed by the processing gain of the spreading factor. A multipath scenario with three possible paths is shown in Figure 2.3

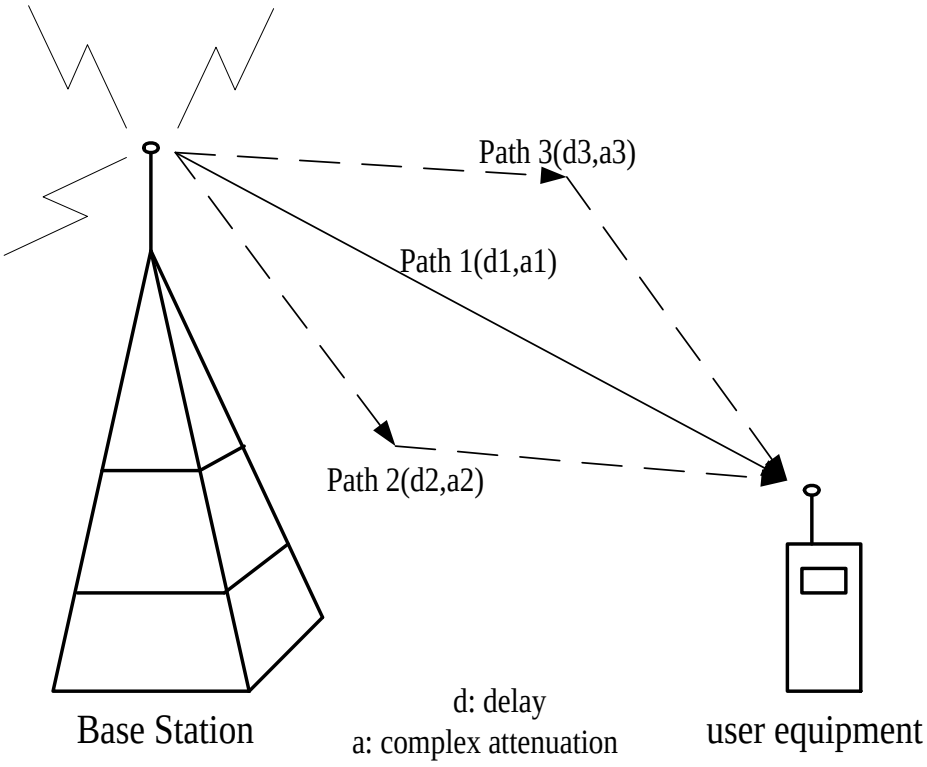

Figure 2.3: Multipath propagation

A benefit of the UMTS transmission is obtained if the resolved multipath signals are combined using a RAKE receiver. The dynamics of the radio propa-

gation suggest the following operation principle for the CDMA signal reception:

1. Identify the time delay position at which significant energy arrives and allocate RAKE fingers (correlation receivers at fixed delay position) to those peaks. The measurement grid for acquiring the multipath delay profiles is in the order of one chip duration (often smaller grids of 1/4 or 1/2 chip duration are utilized). Update rate of various RAKE fingers are in the order of tens of milliseconds.

2. Within each correlation receiver, track the fast changing phase and the amplitude values originating from a fast fading process and remove them. This tracking procedure has to be very fast, with an update rate in the order of 1 ms.

3. Linearly combine the despreaded symbols across all active fingers and present them to the decoder for further processing. This principle is known as Maximum Ratio Combining (MRC).

Figure 2.4 shows the RAKE receiver with three RAKE fingers utilizing MRC.

Multiple receive antennas can be accommodated in the same way as multiple paths received from a single antenna, by just adding additional RAKE fingers to the antennas.

2.4.6 Power control

One of the most important aspects in UMTS is the tight and fast power control[UMT]. The power control reduces the "near-far" problem[2] and enhances the system capacity. The CDMA system has a frequency reuse factor of one. User equipments and base station are transmitting simultaneously. Thus, everyone is a particular source of interference for everyone else.

CDMA systems like UMTS are limited by the overall (intra-cell and the inter-cell) interference level. The capacity is restricted by interference, so it can be increased by an efficient and fast power control. Thus, power control is a prerequisite for a resource efficient operation in UMTS. A fast power control is necessary because of the mobility of the user equipments causing a fast variation of the attenuation of the received power in the user equipment. For example, the power of a user equipment signal moving from the shadow of a building direct to the line of sight connection to the base station changes over several orders of magnitude within some milliseconds. There is power control for the

[2] A single over-powered mobile close to the BTS could block the whole cell.

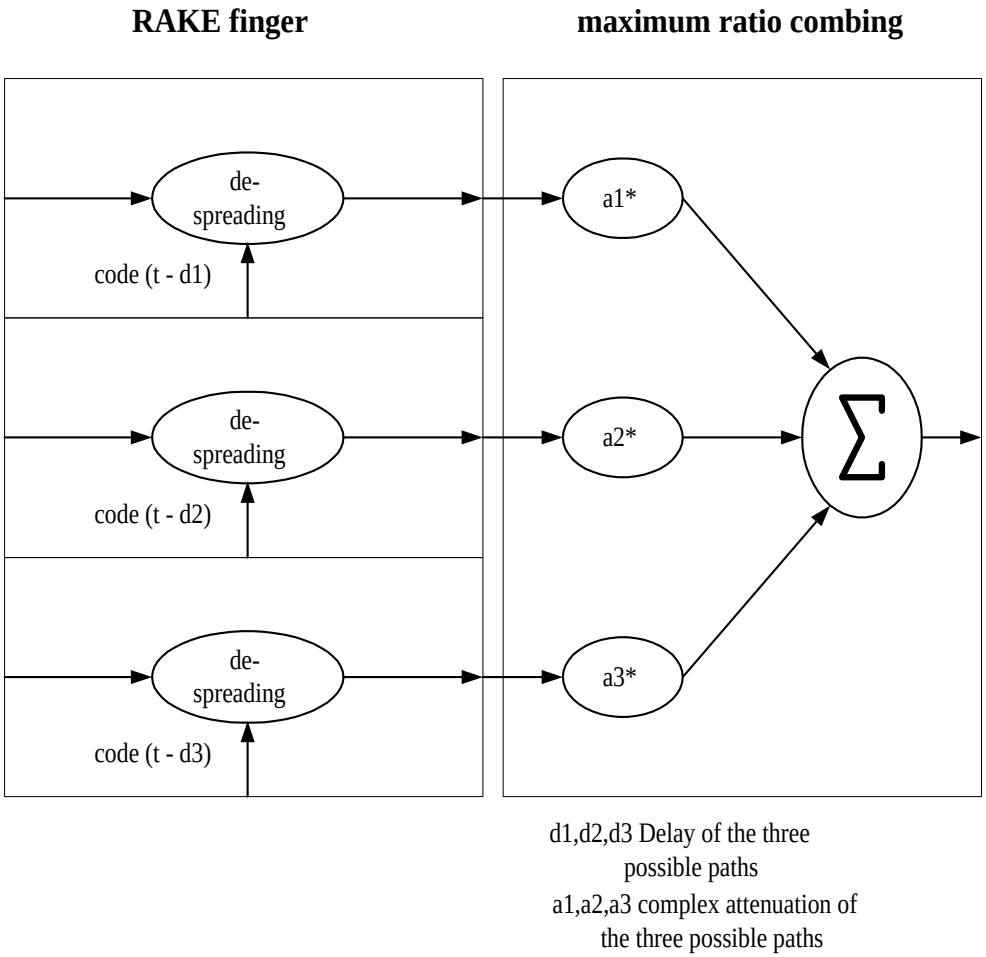

Figure 2.4: RAKE receiver

uplink and for the downlink. The uplink power control reduces the interference level between different user equipments, easing the "near-far" problem. The downlink power control reduces the interference level between neighbouring base stations.

In the UMTS FDD and TDD mode, three different types of power control are used. These types are described in the [25.01h] document.

- Open loop power control
  The open loop power control adjusts the initial access channel transmission power of the user equipment. The user equipment estimates the attenuation between the base station and the user equipment by measuring the received base station signal strength. The smaller the received power, the larger the propagation loss, and vice versa. The originating power of the base station is sent together with other system parameters

as broadcast information. Open loop power control is of special importance in UMTS, because UMTS is optimized for packet switched data transmission.

- Inner loop power control
  For inner loop power control both sides, the user equipment and the base station, compare the received signal of the other side with a predefined signal to interference (SIR) ratio. The signal to interference ratio describes the relation between the received power level of the signal and the undesired interference of other sources. For continuous transmission inner loop power control is the central tuning mechanism in UMTS.

- Outer loop power control
  The signal to interference-ratio from inner loop power control is predefined in the UMTS by the serving radio network controller. Based on the base station and the user equipment measurement reports, the serving radio network controller has information about the quality of transmission (bit error rate, frame error rate). The quality of transmission may vary with changes in environment of transmission. To guarantee the quality of transmission, the radio network controller must be able to vary the predefined signal to interference ratio individually for every connection.

2.5 UMTS logical, transport and physical channels

The concept of logical channels, which is used for UMTS as well as for GSM, is classified by the actual content of the information[UMT]. Different types of contents are described by different logical channels. A detailed description of UMTS logical channels can be found in the 3GPP document[25.01g].

In UMTS a new concept of transport channels, appears. Transport channels are described by how and with what characteristics data are transferred over the radio interface. Different types of content, such as, different logical channels for example, can be mapped together onto one transport channel.

A general classification of transport channels is split into two groups:

- Common transport channel
  There is no need for inband identification of the user equipment when particular user equipments are adressed.

- Dedicated transport channel
  The user equipment is identified by the physical channel. For example,

code and frequency of the UMTS FDD mode and code, timeslot and frequency for the UMTS TDD mode.

Physical channels describe the physical transmission of the information over the radio interface. In GSM physical channel are characterized by frequency and timeslot. UMTS physical channels are characterized either by code and frequency (UMTS FDD mode) or by code, frequency and time slot (UMTS TDD mode).

The UMTS transport channels, physical channels and the mapping of the transport channels onto the physical channels can be found in 3GPP document [25.01e] (FDD mode) and 3GPP document [25.01e] (TDD mode). The UMTS physical layer is described in the general document 3GPP [25.01a].

2.5.1 Mapping of the logical channels onto transport channels UMTS FDD and TDD mode direction downlink

The transport channels are divided into dedicated and common channels. Figure 2.5 shows the mapping of the logical channels onto transport channels for the UMTS TDD and FDD mode in the direction downlink.

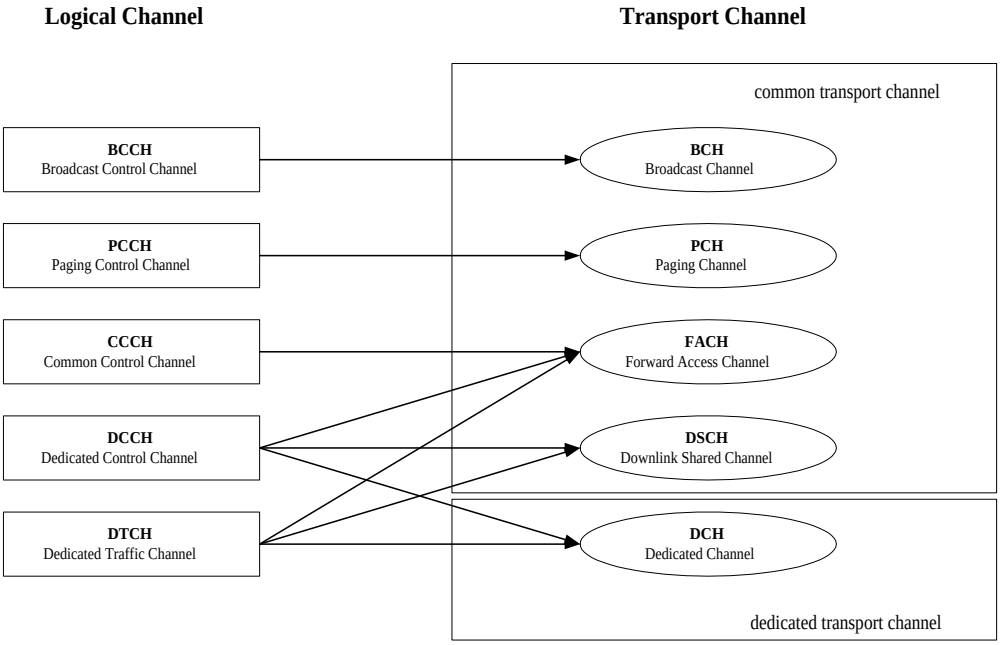

Figure 2.5: Mapping of the logical channels onto transport channels for the UMTS FDD and TDD mode in direction downlink

There exists only one type of dedicated transport channel in the direction downlink:

- Dedicated Channel (DCH)
  The transport channel DCH carries the logical Dedicated Control and Traffic Channel (DCCH and DTCH) information. DTCH information is the user data. DCCH information is necessary for the maintenance of the data transmission (power control, pilot, ...)

Four common transport channels are defined in the direction downlink:

- Broadcast Channel (BCH)
  The transport channel BCH carries only the logical channel BCCH (Broadcast Control Channel) information. It is used to broadcast continuously system- and cell-specific information over the entire area of the cell.

- Paging Control Channel (PCH)
  The transport channel PCH carries only the logical channel PCCH (Paging Control Channel) information. It is transmitted always over the entire cell for paging or notification of user equipments. The PCH transmission is associated with the paging indicator, a physical layer signal, to enable sleep-mode procedures.

- Forward Access Channel (FACH)
  The FACH enables the transmission of logical channel CCCH (Common Control Channel) information as well as DCCH (Dedicated Control Channel) and DTCH (Dedicated Traffic Channel) information.

- Downlink Shared Channel (DSCH)
  The downlink shared channel carries DCCH and DTCH information. It is shared by several user equipments. The downlink shared channel is associated with a dedicated physical channel.

2.5.2 Mapping of the logical channels onto transport channels UMTS FDD mode direction uplink

Figure 2.6 shows the mapping of the logical channels onto transport channels for the UMTS FDD mode in the direction uplink.

There exists only one type of dedicated transport channel in the direction uplink of the UMTS FDD mode:

- Dedicated Channel (DCH)
  It is identical to the dedicated channel of the UMTS FDD mode in the direction downlink.

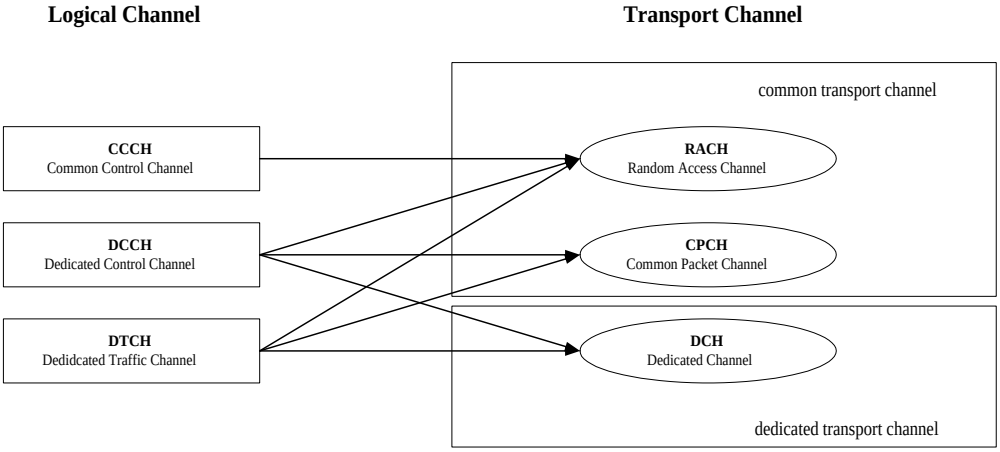

Figure 2.6: Mapping of the logical channels onto transport channels for the UMTS FDD mode in direction uplink

Only two types of common transport channels are defined for the direction uplink of the UMTS FDD mode:

- Random Access Channel (RACH)
  The transport channel RACH is a contention based uplink channel. It enables the transmission of the logical channel CCCH (Common Control Channel) information as well as DCCH (Dedicated Control Channel) and DTCH (Dedicated Traffic Channel) information. The CCCH information in the direction uplink is used by the user equipment for accessing a new cell.

- Common Packet Channel (CPCH)
  The transport channel CPCH is a contention based channel for uplink transmission of bursty traffic (packet data). It can be regarded as an extension of the RACH. It exists only in the UMTS FDD mode enabled by the continuous UMTS transmission.

2.5.3 Mapping of the logical channels onto transport channels UMTS TDD mode direction uplink

Figure 2.7 shows the mapping of the logical channels onto transport channels for the UMTS TDD mode in the direction uplink.

There exists only one type of dedicated transport channel in the direction uplink of the UMTS TDD mode:

- Dedicated Channel (DCH)

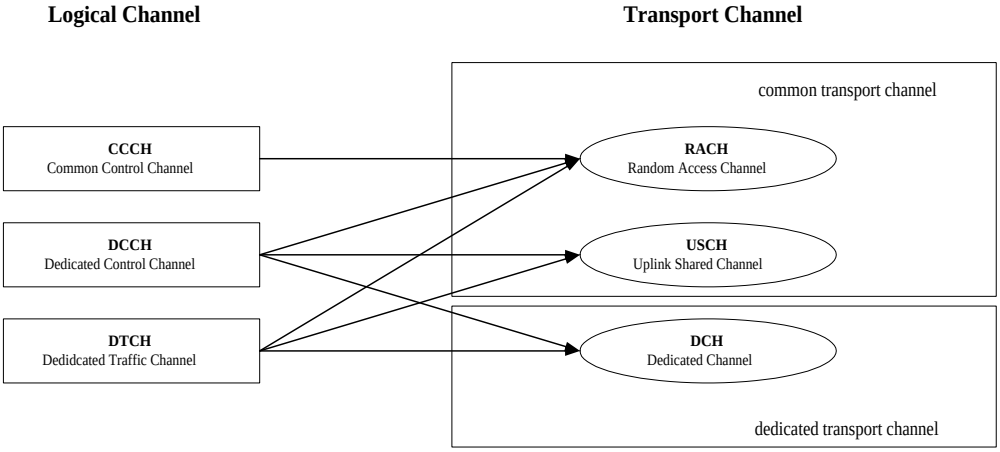

Figure 2.7: Mapping of the logical channels onto transport channels for the UMTS TDD mode in direction uplink

It is Identical to the dedicated channel of the UMTS FDD mode for the direction downlink.

Only two types of common transport channels are defined for the direction downlink of the UMTS FDD mode:

- Random Access Channel (RACH)
  It is identical to the random access channel of the UMTS FDD mode for the direction uplink.

- Uplink Shared Channel (USCH)
  The transport channel USCH is shared by several user equipments carrying DCCH (Dedicated Control Channel) or DTCH (Dedicated Traffic Channel) data. The uplink is equivalent to the downlink shared channel DSCH of the UMTS TDD mode.

2.5.4 Mapping of the transport channels onto physical channels UMTS TDD mode direction downlink

The transport channels are mapped onto the physical channel. Not every physical channel carries individual information over the radio interface. Some physical channels have fixed contents. Other physical channels of the UMTS TDD mode carry several types of information. More than one transport channel can be mapped onto them. Figure 2.8 shows the mapping of the transport channels onto physical channels for the UMTS TDD mode in the direction downlink.

The physical channels of the UMTS TDD mode in direction downlink are:

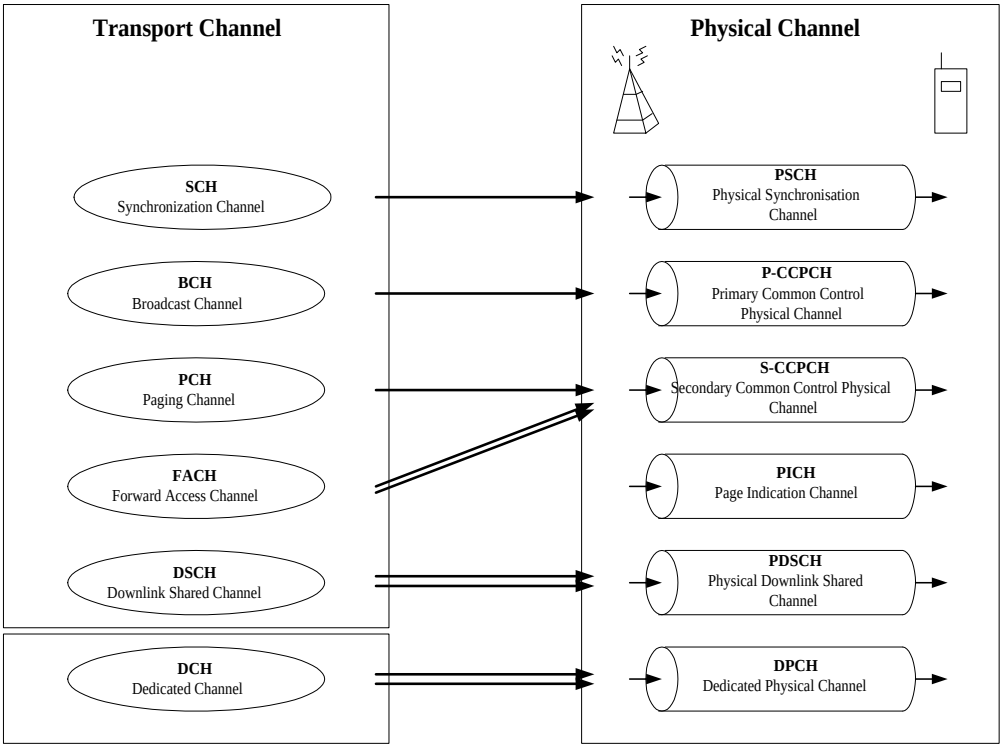

Figure 2.8: Mapping of the transport channels onto physical channels for the UMTS TDD mode in direction downlink

- Physical Synchronization Channel (PSCH)
  The PSCH is used for cell search and time synchronization. Furthermore, it indicates the allocation of the P-CCPCH (Primary Common Control Physical Channel). Therefore, the SCH (Synchronization Channel) transports channel information which is mapped on the PSCH.

- Primary Common Control Physical Channel (P-CCPCH)
  The P-CCPCH provides the beacon function for the UMTS TDD cell. The data stream of the BCH (Broadcast Channel) transport channel is mapped onto the P-CCPCH.

- Secondary Common Control Physical Channel (S-CCPCH)
  The paging/notification data of the PCH (Paging Channel) transport channel and the FACH (Forward Access Channel) common and dedicated data are mapped onto one or more S-CCPCH channels.

- Page Indication Channel (PICH)
  The PICH is a physical layer signal. This signal is always associated with an S-CCPCH (Secondary Common Control Physical Channel) to which

the PCH (Paging Channel) transport channel is mapped. It carries the Paging Indicator (PI) to enable sleep procedures.

- Physical Downlink Shared Channel (PDSCH)
  The data stream of the DSCH (Downlink Shared Channel) transport channel is mapped onto the PDSCH.

- Dedicated Physical Channel (DPCH)
  The DCH transport channel carries dedicated control and user data which are encoded and interleaved. The resulting data streams are mapped onto the physical channel DPCHs. It is identical to the random access channel of the UMTS FDD mode.

2.5.5 Mapping of the transport channels onto physical channels UMTS TDD mode direction uplink

Figure 2.9 shows the mapping of the transport channels onto physical channels for the UMTS TDD mode in the direction uplink.

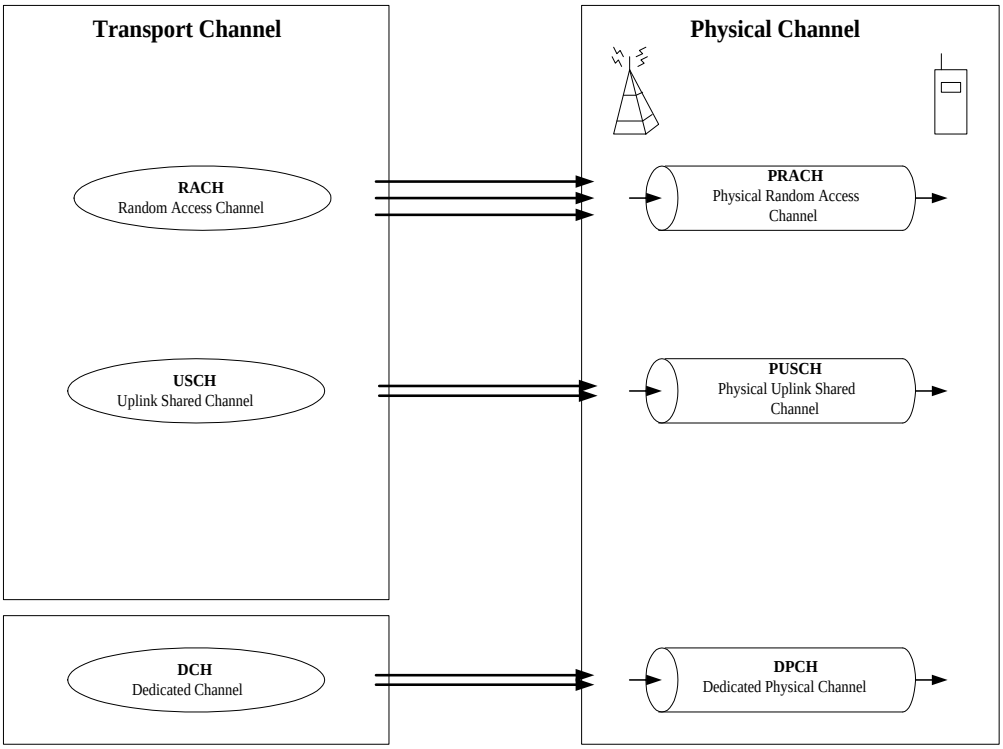

Figure 2.9: Mapping of the transport channels onto physical channels for the UMTS TDD mode in direction uplink

The physical channels of the UMTS TDD mode in direction uplink are:

- Physical Random Access Channel (PRACH)
  The data stream of the RACH (Random Access Channel), consisting of common control and dedicated data, is mapped after coding and interleaving onto the PRACH.

- Physical Uplink Shared Channel (PUSCH)
  The data stream of the USCH (Uplink Shared Channel), consisting of dedicated control and user data, is mapped onto the PUSCH. The PUSCH is shared by several user equipments. It is the uplink equivalent of the PDSCH (Physical Downlink Shared Channel) of the UMTS TDD mode, but different to the PDSCH as defined in the UMTS TDD mode only.

- Dedicated Physical Channel (DPCH)
  In the same way as in the downlink of the UMTS TDD mode the DCH (Dedicated Physical Channel) transport channel dedicated control and user data are encoded and interleaved. The resulting data streams are mapped onto the physical channels DPCHs. The difference to the handling of dedicated data in the uplink of the UMTS FDD mode is caused by the bursty nature of the UMTS TDD mode.

2.5.6 Mapping of the transport channels onto physical channels UMTS FDD mode direction downlink

The transport channels are mapped onto the physical channels. Not every physical channel carries individual information over the radio interface. Therefore, some physical channels are defined which need no corresponding transport channel. Other physical channels carry several different types of information. More than one transport channel can be mapped onto them. Figure 2.10 shows the mapping of the transport channels onto physical channels for the UMTS FDD mode in the direction downlink.

The physical channels of the UMTS FDD mode in direction downlink are:

- Common Pilot Channel (CPICH)
  The CPICH is an unmodulated code channel, which is scrambled with a cell specific scrambling code. It is the default phase reference for all downlink shared physical channels.

- Synchronization Channel (SCH)
  The SCH is fixed within a cell. It is used for cell search and time synchronization.

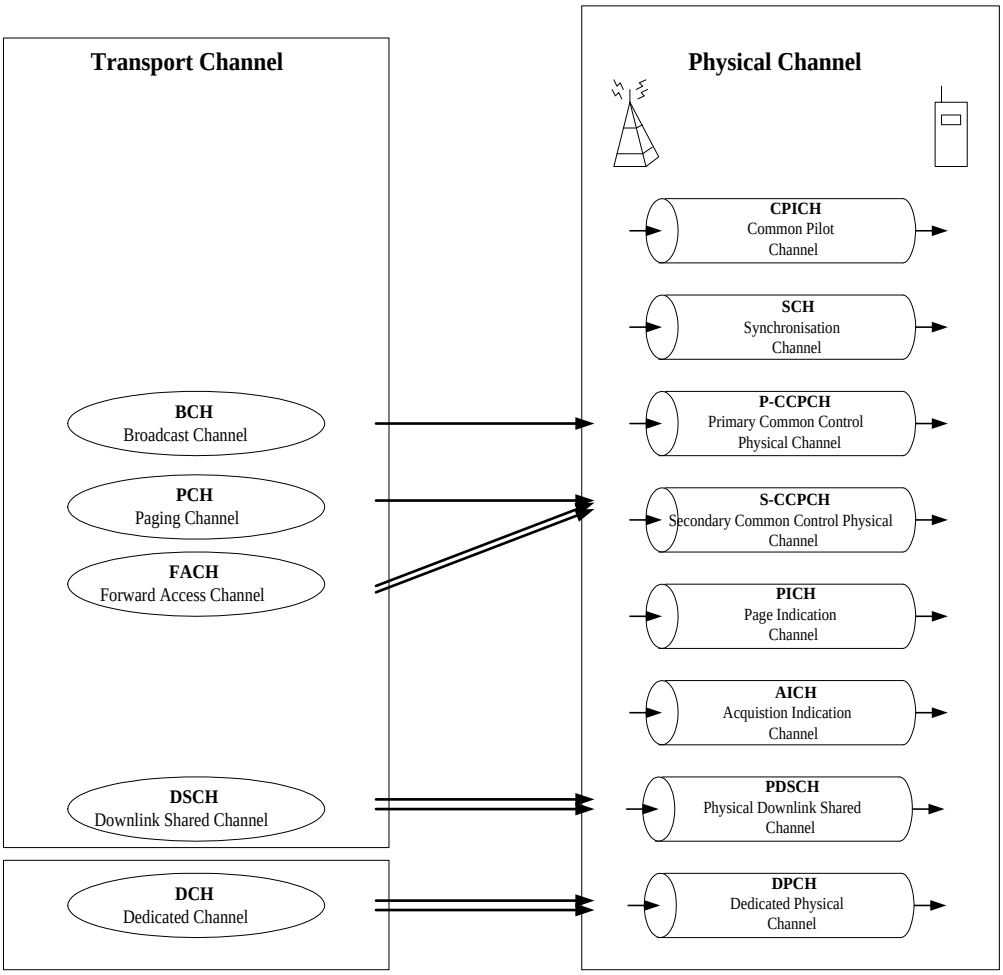

Figure 2.10: Mapping of the transport channels onto physicals channels for the UMTS FDD mode in direction downlink

- Primary Common Control Physical Channel (P-CCPCH)
  The data stream of the BCH (Broadcast Channel) transport channel is mapped sequentially onto the P-CCPCH channel.

- Secondary Common Control Physical Channel P-CCPCH
  The paging/notification data of the PCH (Paging Channel) transport channel and the FACH (Forward Access Channel) control and dedicated date are mapped sequentially onto the one or more S-CCPCHs after coding and interleaving.

- Page Indication Channel (PICH)
  The PICH is a physical layer signal. It is always associated with the S-CCPCH (Secondary Common Control Channel) to which a PCH (Paging

Channel) transport channel is mapped. It carries the paging indicator to enable sleep procedures.

- Acquisition Indication Channel (AICH)
  The AICH carries short sequences, the acquisition indicators. These indicators are used to acknowledge the PRACH (Physical Random Access Channel)/PCPCH (Physical Common Packet Channel) preamble reception.

- Physical Downlink Shared Channel (PDSCH)
  The data stream of the DSCH (Downlink Shared Channel) transport channel is mapped onto the PDSCH.

- Dedicated Physical Channel (DPCH)
  The DCH transport channel data are coded and interleaved. The resulting data stream is mapped sequentially directly onto the physical channels DPCHs.

2.5.7 Mapping of the transport channels onto physical channels UMTS FDD mode direction uplink

Figure 2.11 shows the mapping of the transport channels onto physical channels for the UMTS FDD mode in the direction uplink.

The physical channels of the UMTS FDD mode in direction uplink are:

- Physical Random Access Channel (PRACH)
  The data stream of the RACH (Random Access Channel), consisting of common control and dedicated data, are sequentially mapped after coding and interleaving onto the message part of the PRACH.

- Physical Common Packet Channel (PCPCH)
  The data of the CPCH which can be regarded as an extended RACH are used for optimized packet data transfer, are sequentially mapped after coding and interleaving onto the message part of the PCPCH.

- Dedicated Physical Control Channel (DPCCH) and Dedicated Physical Data Channel (DPDCH)
  Different to the UMTS FDD mode download direction, the dedicated control data and user data of the DCH (Dedicated Channel) are not multiplexed together in the UMTS FDD mode uplink direction. There are two types of uplink dedicated physical channels, the DPDCH for user data and the DPCCH for dedicated control data. The DPDCH and the DPCCH are I/Q code multiplexed within each frame.

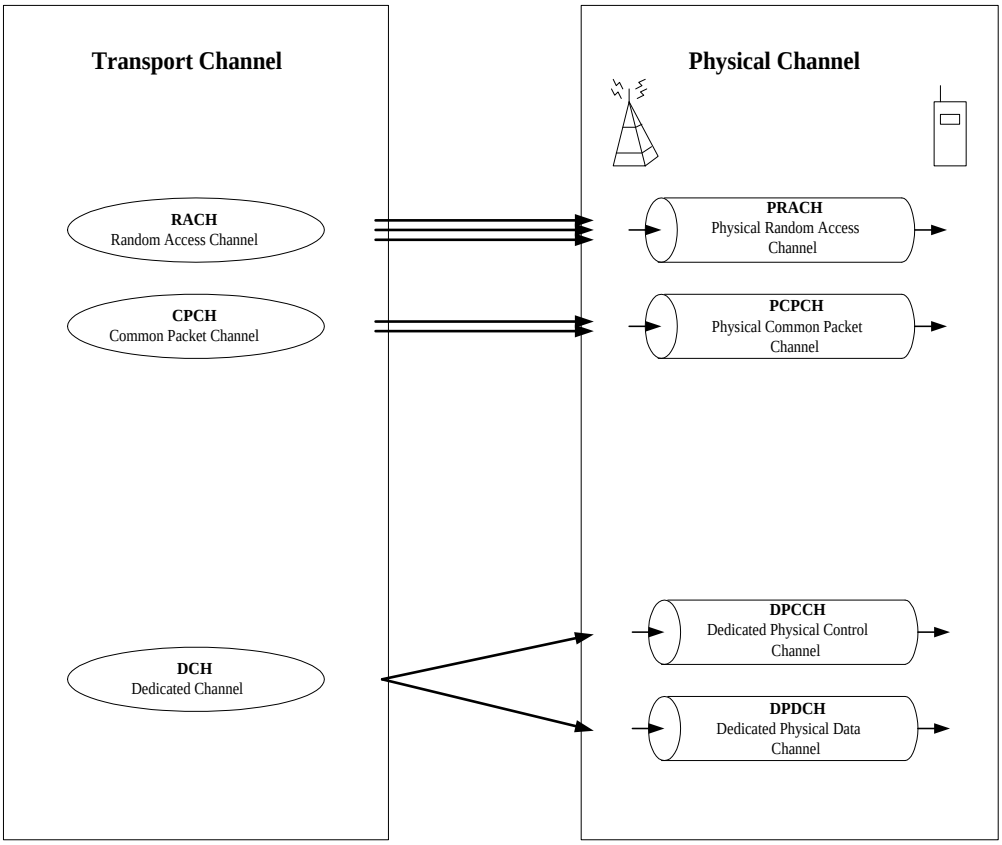

Figure 2.11: Mapping of the transport channels onto physicals channels for the UMTS FDD mode in direction uplink

2.6 UMTS slot synchronization

The SCH (Synchronization Channel) is needed for the UMTS slot synchronization, as will be explained in detail in the following sections.

2.6.1 Synchronization Channel

Physical UMTS downlink channels are defined by a specific carrier frequency, scrambling code, channelization code, time start and time stop. The shortest time duration in the UMTS is the chip. It is coded as $\{+1, j, -1, -j\}$. The UMTS downlink physical channel SCH is used for time synchronization (chip, frame and time slot) and for scrambling code group recognition as well as BTS identification. Figure 2.12 illustrates the structure of the SCH frame.

It consists of two sub channels, the primary synchronization channel and the secondary synchronization channel.

- Primary Synchronization Channel (P-SCH)

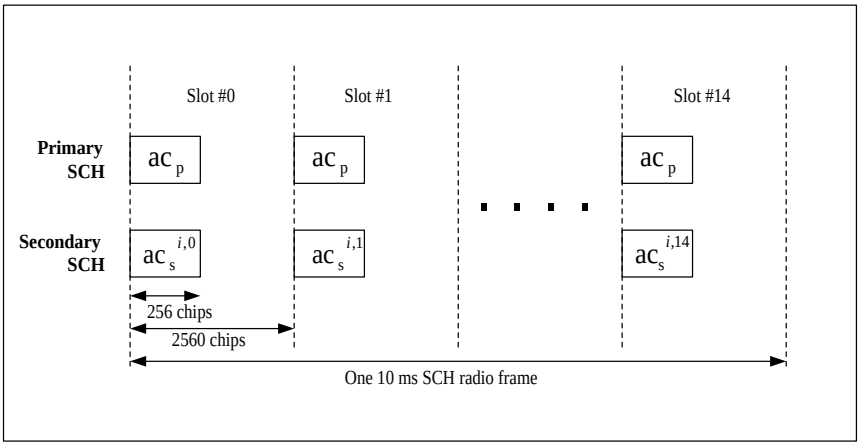

Figure 2.12: Structure of the synchronization channel

The P-SCH consists of a predefined code sequence (Primary Synchronization Code (PSC) c_p) with 256 chip length, located at the start of every time slot. The primary synchronization channel is identical in every cell in the system. The primary synchronization code is furthermore chosen to have good aperiodic auto correlation properties. Define:

$$b =< x_1, x_2, x_3, \ldots, x_{16} >=$$

$$< 1, 1, 1, 1, 1, 1, -1, -1, 1, -1, 1, -1, 1, -1, -1, 1 > .$$

The primary synchronization code is generated by repeating the sequence b (variable name chosen) modulated by a specific Golay complementary sequence. The real part and imaginary part are identical. The PSC c_p is defined as:

$$c_p = (1 + j) \times < b, b, b, -b, -b, b, -b, -b, b, b, b, -b, b, -b, b, b >$$

The aperiodic auto correlation function of a time limited-signal is defined:

$$r(n) = \sum_{k=0}^{255-n} x(k) \cdot x(k + n) \tag{2.1}$$

Good aperiodic auto correlation properties requires the primary synchronization code (constructed out of a Golay code) to have small side-correlation peaks. The synchronization will be performed only on the main correlation peak. Figure 2.13 shows the aperiodic auto correlation function of the PSC.

- Secondary Synchronization Channel (S-SCH)
  The S-SCH is transmitted in parallel to the P-SCH. It consists of 15 different code sequences (secondary synchronization code $c^{i,1..14}$;i= 1 .. 64), each 256 chips in length. These code sequences are repeated every frame. One of the 64 different options (i = 1 .. 64) is equivalent to the number of the scrambling code group. Therefore the sequence on the S-SCH indicates the scrambling code group the cell belongs to, thus identifying the BTS. In this way, the search of the scrambling code group of the cell can be limited to a subset of all the codes.

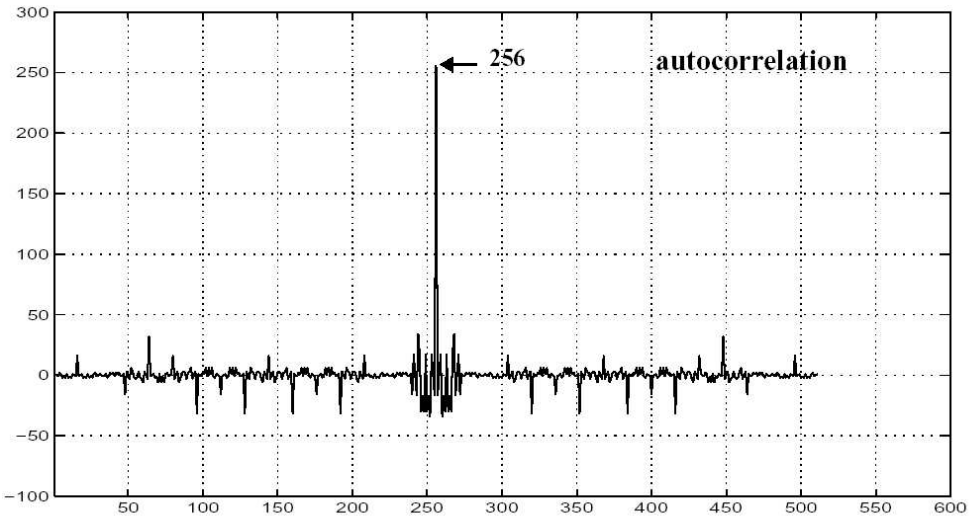

Figure 2.13: Aperiodic auto correlation function of the PSC

The user equipment searches for a cell and determines the downlink scrambling code and common channel frame synchronization of that cell. This is done during the cell search procedure. The cell search procedure is carried out in three steps:

- first step: slot synchronization
  The user equipment uses the Primary Synchronization Code (PSC) being transmitted in the primary synchronization channel P-SCH, to acquire slot synchronization to a cell. This task is done typically by a matched filter (or any similar device). The slot timing of the cell can be obtained by detecting peaks at the matched filter output.

- second step: frame synchronization
  The second step of the cell search procedure is the frame synchronization.

The user equipment uses the secondary synchronization code being transmitted in the secondary synchronization channel S-SCH, to find frame synchronization and to identify the group of the cell found in the first step.

- third step: scrambling code identification
  During the last step, the user equipment determines the exact primary scrambling code used by the found cell.

The UMTS cell search procedure is described in more detail in the 3GPP documents [25.01b], [25.01c], [25.01d].

Chapter 3

Chip Design with SystemC

3.1 Motivation to a new modeling language

Introducing a new modeling language such as SystemC was motivated by the changing nature of systems under design[Sysb]. When systems were composed primarily of discrete parts such as microprocessors, memory chips, analog devices and Application Specific Integrated Circuits (ASIC), the design process usually started with one or two system design experts who would partition the functionality into hardware and software, and further partition the hardware parts into parts of the ASIC. It was possible to write a specification for an ASIC of a few thousand to few hundred thousand gates in natural languages and hand off to an ASIC designer or team who would start the translation process by capturing the design at the Register Transfer Level (RTL) for which Hardware Description Languages (HDLs) are the perfect match. In contrast to this stands a modern design called System on a Chip (SoC). This new type of an integrated circuit may contain one or more processors including both 32-bit Micro Controllers (MC) and Digital Signal Processors (DSPs) or specialized media processors. On-chip memory, accelerating hardware units for dedicated functions, and peripheral control devices will be linked together with processors by a complex on-chip communication network incorporating on-chip busses. Specifying, designing and implementing such complex systems in hardware and software, the designers are compelled to move on from the old hardware description languages. The designers also have to move beyond the RTL level of abstraction used with this HDLs. The designers need to move to what has been termed the “system-level“ of design. So what are the requirements for this new system-level modeling language? A new language should cover all abstraction levels, so that there will be no need to translate the code between different chip design languages if the level of abstraction changes. The best way to solve this

problem is to base this new system level design language on well-established programming language, in order to capitalize on the extensive infrastucture of capture, compilation, and debugging tools already available. This new design language is based on an object-oriented programming language, this allows modeling flexibility and facilitate reuse, through capabilities such as templates and inheritance. C++ has proven to be a reasonable choice as the basis for such a system design language. The absence of a standardized and well accepted system level design language has inhibited the development, exchange and reuse of Intellectual Property (IP) models at the system level. Where reuse occurs in hardware design, it starts at the real transistor level using standard HDLs. When IP models are used in HDLs, this is usually done with the implementation in mind, thus inhibiting design space exploration and providing relatively poor IP protection, which in turn prevents easy evaluation. Real transistor level models also simulate slowly compared to more abstract models. A solution for this problem is the new design language SystemC. The fundamental motivation for SystemC is to provide a modeling framework for systems in which high-level functional models can be refined down to implementation in a single language.

3.2 Introduction to SystemC

Modeling of systems above the RTL level of abstraction, including systems which can be implemented in software, hardware or the combination of the two is the ultimate goal of SystemC.

3.3 SystemC

SystemC is a C++ class library and provides a methodology allowing effective creation of cycle-accurate models of software algorithms, hardware architectures and interfaces of a system on a chip [Sysb][TGS02]. SystemC uses standard C++ development tools to create a system level model allowing the designer to simulate quickly, validate and optimize the design and explore various algorithms. C or C++ are the language of choice for software algorithm and interface specification because they provide the control and data abstraction necessary to develop compact and efficient system descriptions. Most designers are familiar with these languages and the large number of development tools associated with them.

SystemC uses a layered approach, allowing for the flexibility of introducing new, higher-level constructs sharing an efficient simulation engine. Figure 3.1 shows the various layers of SystemC.

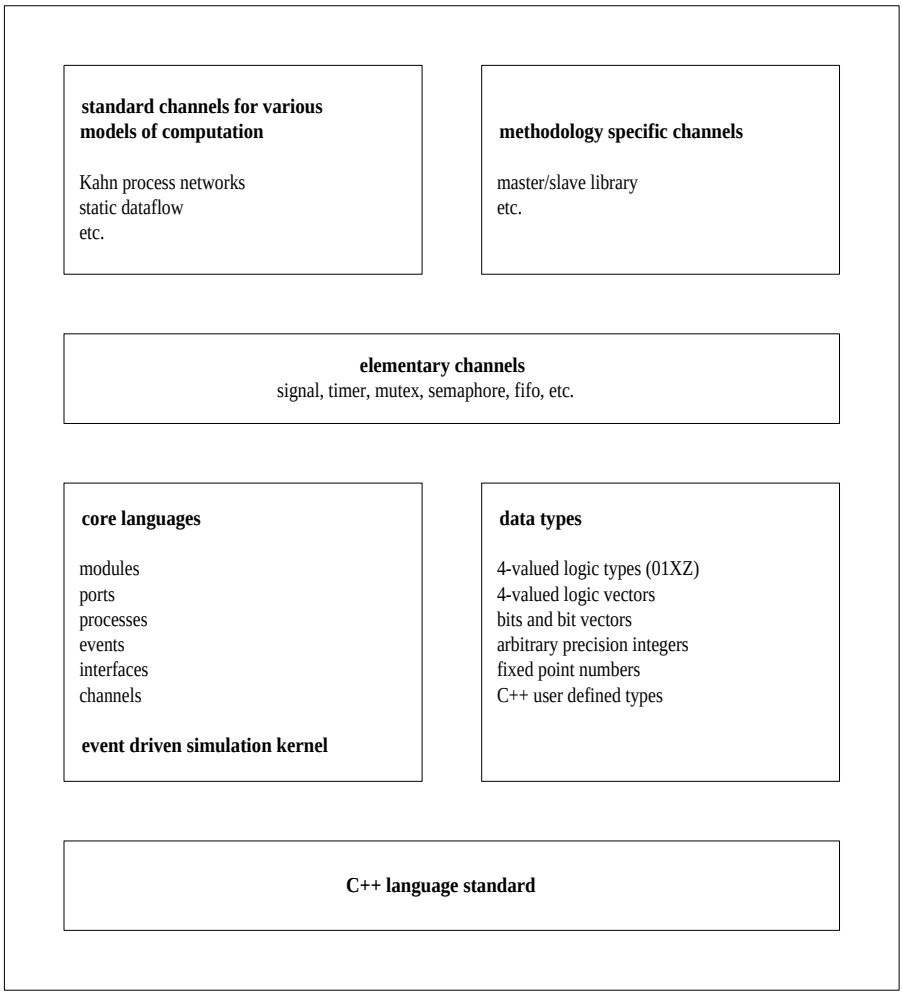

Figure 3.1: SystemC language architecture

The base layer of SystemC provides an event driven simulation kernel. This kernel works with events and processes in an abstract manner. It knows only how to operate on events. It switches between processes without knowing what the events actually represent or what the processing does. Other elements of SystemC include modules and ports for representing structural information, and interfaces and channels as an abstraction for communication. The simulation kernel and these abstract arguments together form the core language.

3.3.1 SystemC design methodology

The SystemC design methodology is based on refinements leading from one level of abstraction to another. Using this refinement methodology, the designer can implement design changes more easily and detect bugs during refinement.

Figure 3.2 shows the SystemC methodology.

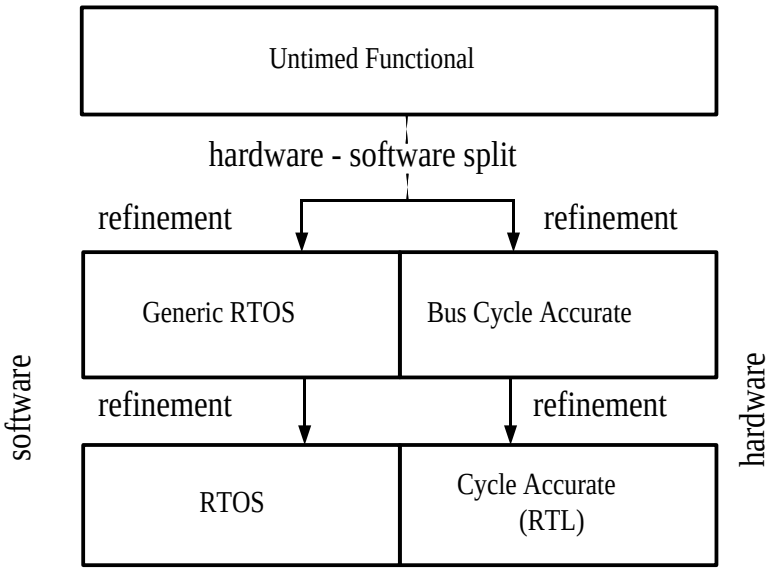

Figure 3.2: SystemC methodology

The SystemC methodology uses different levels of abstraction[TGS02].

- Untimed functional
  At the untimed functional level, the device is modeled in a data driven (stream) approach. Connections are done via channels. Processes react whenever data appears between different modules. No timing information is required at this level of abstraction. The processes which are running algorithms are modeled with zero system time to run. SystemC 2.0 covers this level of abstraction.

- Bus cycle accurate
  The bus cycle accurate level is also called Transaction Level Modeling (TLM). In a transaction level model, communication is modeled using function calls. Only the interfaces and channels are described accurately. At this level of abstraction timing information may be, but does not have to be added. Transaction level models can be used to accurately model effects such as bus loading and contention and overall system performance. They also provide a high performance, accurate and effective way to model hardware and software interactions at a very early stage in the design process. As an example of transaction level modeling, a PCI bus and the interfaces to it may be described. This is also supported by SystemC 2.0.

- Cycle accurate
  A cycle accurate model is pin accurate, cycle accurate and functionally accurate at its boundaries. It is at the same level of abstraction as real transistor level. SystemC 1.0 covers this level of abstraction.

Future versions of SystemC, namely SystemC 3.0, will also include the possibility to describe Real Time Operation Systems (RTOS). This allows the designer to model firmware and hardware in one design language and simulate hardware and firmware together at an early design stage.

3.3.2 SystemC highlights

SystemC supports hardware software co-design and the description of the architecture of complex systems, consisting of both hardware and software components. It supports the description of hardware, software and interfaces in a C++ environment. The following features of SystemC will be explained in detail [Sysb][Sysa][Hor00]:

- Modules
  Modules are the basic blocks for partitioning a design. They allow designers to break complex systems into smaller, more manageable pieces, and to hide internal data representation and algorithm from other modules. A module consists of ports, processes, internal data, channels and may have a hierarchical structure. A small code example for `SC_MODULE` declaration reads:

```
SC_MODULE(Adder)
{
  //ports, processes, internal data, etc.
  SC_CTOR(Adder)
  {
    //body of the constructor
    //process declaration, sensitivities, etc.
  }
};
```

  The `SC_MODULE` macro is simply a shorthand for deriving class *Adder* from the library `SC_MODULE`. The `SC_CTOR` constructor provides a convenient way to manage the module's hierarchical names automatically.

- Processes
  Processes are used to describe functionality. They are contained inside modules. The processes are defined as member function of the module and declared to be a SystemC process in the module's constructor. A

declaration in the module's constructor is required to register the member function with the simulation kernel. Processes access external channels through ports of their containing module. SystemC has two kinds of processes: *method process* and *thread process*. A *method process*, when triggered, always executes its body from the beginning to the end. It does not keep an implicit execution state. A *thread process* may have its execution suspended by calling the library function `wait()` or any of its variants. The *thread process* remembers the point of suspension along with all local variables, so that when execution is resumed, it will continue from that point rather than from the beginning of the process. Continuing with the `Adder` example, we have:

```
SC_MODULE(Adder)
{
  //ports, processes, internal data, etc.
  void compute()
  {
    c = a + b;
  }

  SC_CTOR(Adder)
  {
    SC_METHOD(compute);
    sensitive << a << b;
  }
};
```

The member function `compute()`, when invoked, computes the sum of the inputs `a` and `b`, and writes the result to the output `c`. The statement `sensitive << a << b` specifies that this process is sensitive to changes in the values of the hardware signals, which are connected to the input ports `a` and `b`.

- Interfaces, channels and ports
  SystemC uses interfaces, ports and channels to provide for flexibility and high level of abstraction. Channels are responsible for holding and transmitting data. The interface is a "window" into a channel describing the set of operations, or a subset thereof, that the channel provides. Ports are facilitating access to channels through interfaces. Continuing with the `Adder` example, we have:

```
SC_MODULE(Adder)
{
  sc_in<int> a;
  sc_in<int> b;
  sc_out<int> c;

  void compute()
```

```
  {
    c = a + b;
  }

  SC_CTOR(Adder)
  {
    SC_METHOD(compute);
    sensitive << a << b;
  }
};
```

- Rich set of port and signal types
  To support modeling at different level of abstraction, from the functional level to the RTL, SystemC supports a rich set of port and signal types. This is different from languages like Verilog that only support bits and bit-vectors as port and signal types.

- Rich set of data types
  SystemC has a rich set of data types to support multiple design domains and abstraction levels. The fixed precision data types allow for fast simulation, the arbitrary precision types can be used for computation with large numbers and the fixed-point data types can be used for DSP applications. There is no size limitation for arbitrary precision SystemC types.

- Clocks
  SystemC captures the notion of clocks in special signals. Clocks are the timekeepers of the system during simulation. Multiple clocks, with arbitrary phase relationships, are supported.

- Cycle bases simulation
  SystemC includes a small, but powerful cycle-based simulation kernel allowing high speed simulation.

- Multiple abstraction levels
  SystemC supports untimed models at different levels of abstraction, ranging from high level functional models to detailed clock cycle accurate RTL models. It supports iterative refinement of high level models into lower levels of abstraction.

- Communication protocols
  SystemC provides multi level communication semantics that enable the designer to describe systems on a chip and system I/O protocols on different levels of abstraction.

- Debugging support
  SystemC classes have run-time error checking that can be turned on with a compilation flag.

- Waveform tracing
  Waveform formats like VCD, WIF and ISDB are supported by SystemC.

3.3.3 SystemC at all abstraction levels

Using SystemC has the benefit that the design engineers do not have to be experts in multiple languages. SystemC allows modeling from the abstraction level untimed functional to RTL, if necessary. The SystemC approach provides higher productivity, because the designer can model at higher abstraction levels leading to faster simulation speed. The higher level can result in smaller code, that is easier to write and simulates faster than traditional modeling environments. The biggest benefit is that test-benches can be reused from the system level model to the RTL model. This saves conversion time. Using the same test-bench also gives the designer a high confidence for the system level and the RTL model implementing the same functions.

Chapter 4

Cell searcher algorithm in untimed functional SystemC

Due to the increasing complexity of IC designs, design verification is becomeing a major bottleneck in digital hardware design. At the same time, the increasing mask costs and the relevance of time-to-market make it crucial to have a reliable, but fast verification process. The traditional techniques for functional verification are based on simulation, using a set of test data or a test bench. One of the drawbacks of this approach is that, verifying even a small part of the functions of a complex IC, requires large test sets and/or sophisticated test-benches. To reduce this complexity of testing, the designer starts with a purely functional model of a system. This functional description can be timed or untimed. The untimed functional level is a data driven (stream) approach. Interfaces in the functional description are transaction based and event oriented, rather than cycle accurate, as at lower abstraction levels. Details such as timing, fine-grain structure, or low-level communication protocols are best avoided at the functional abstraction level. At this early stage of the design process there is only interest in describing the functions of the system. At this point, there is no interest in partitioning the functions into hardware and software parts. The untimed functional model is an executable specification of the system.

The functional models of the system are implemented as dataflow models and are connected with channels. At the abstraction level of untimed functional model, data is represented as a sequence of numbers with no notion of time. The channels between the functional models are implemented as queues with no data loss. As a consequence, if the queue is full with data, the next block which is connected to this queue, needs to process data from the queue. Dataflow models typically read data from their inputs, process the data according to

their functionality and write the results to their output ports.

4.1 Algorithmic and architectural modeling with CoCentric System Studio

CoCentric System Studio offers a wide variety of modeling capabilities, providing the designer with the means to capture complex systems quickly and efficiently. The modeling paradigm can be mixed hierarchically at all levels of abstraction. CoCentric System Studio models are divided into two distinct domains, algorithmic and architectural, reflecting the primary design focus that each type of the model supports[CoC]:

- Algorithmic models
  These models describe the functionality of a system at the untimed functional level of abstraction. The design is captured using a mixture of data flow and extended hierarchical state machine models.
  - Dataflow graph models
    this is a data driven approach in which blocks of data are read and written through the model's ports. Data are read from the input ports, processed according to the model's algorithm and the results are written to the output ports.
  - Control models
    allow the designer to describe the control flow through a Finite State Machine (FSM). The basic elements of control models are states, transition, signals and parameters. A specialized type of control models are *AND models*. These *AND models* are hierarchical control models with multiple sub-instances, called pages, that execute in parallel.
  - *Prim* models
    may contain hierarchy, but may also be used as the lowest level of a hierarchical structure in both dataflow graph and control models. The functions of the *prim model* are written in generic C which is extended with C++ constructs to make it possible to use interfaces.
  - Stream Driven Simulation (SDS) models
    are a special form of primitive model. These models are usually the result of importing COSSAP[1] designs into CoCentric System Studio.

[1]Former C-based system level design tool by Synopsys.

- Architectural models
  Architectural (SystemC) models capture the architecture of the system at various levels of granularity and abstraction. These models allow the designer to describe the overall platform architecture in terms of its busses, memories and processors, as well as making it possible for the design engineer to describe the internal architecture of individual components. These models span a broad range from high-level abstractions suitable for functional descriptions of a system and early architectural analysis down to cycle-accurate and pin-accurate synthesizeable hardware models.

CoCentric System Studio in conjunction with SystemC gives the designer the possibility to describe a system at the untimed functional abstraction level. The Graphical User Interface (GUI) of Cocentric System Studio makes it easy for the designer to draw functional blocks, which are connected with FIFO channels, to simulate and to verify the design at the functional abstraction level. This tool also makes it possible to reuse blocks with different data types (floating point and fix point data types). The simple syntax of SystemC is very convenient and allows the designer to concentrate on the main functionality of the system. A graphical SystemC description is shown on Figure 4.5.

4.2 Cell searcher concept

The cell searcher has to perform UMTS slot synchronization, UMTS frame synchronization and estimate the delay on the basis of incoming chips. The incoming chips are two-times oversampled , so the obtained half chips are stored in the *Searcher-RAM*. The cell searcher concept implies two independent data paths, called the *datapath_SF* and the *datapath_DEL* as indicated in Figure 4.1.

The data paths operate in parallel with the same input half chip sequence. The first data path, the *datapath_SF*, is dedicated to processing the UMTS slot- or frame synchronization. The slot and the frame synchronization cannot be processed simultaneously. The second data path, the *datapath_DEL*, is dedicated to processing the delay estimation. An extension of the cell searcher system with a third or a fourth data path is possible within the cell searcher concept. Figure 4.1 shows the global architecture of the UMTS cell searcher.

4.2.1 UMTS slot synchronization

Five modes of UMTS slot synchronization are possible.

- Preselection part, hard decision algorithm
  For the hard decision algorithm, each of the 5120 coherent correlation re-

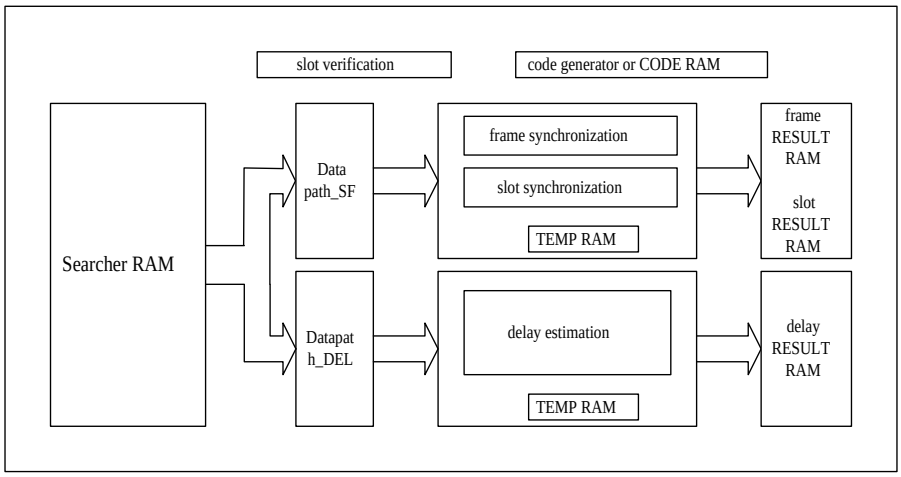

Figure 4.1: UMTS cell searcher according the cell searcher concept

sults per time slot are compared with a predefined threshold. If the correlation result is bigger than the *threshold0*, a related internal counter value (4 bit) is incremented. The counter values are stored in the *TEMP_RAM* block. This procedure is done for several slots for averaging. After the procedure, 5120 counter values within the *TEMP_RAM* are available for further hardware processing. In the first step, all counter values, stored in the *TEMP_RAM*, above a pre-defined counter threshold are evaluated. The indices of these entries are stored within the *MASK_RAM* and serve as candidates for different time slot timings. In the second step, these candidate indices are used for referencing soft output averaged correlation values in the soft decision algorithm.

- Selection part, soft decision algorithm
  The memory area of the *TEMP_RAM* is not sufficient to store all 5120 averaged correlation values (16 bits) over several time slots. The size of *TEMP_RAM* is determined by the bit width of the internal counter. For example, if the oversampling ratio is equal to two (5120 correlation results per time slot) and a bit width of the internal counter is equal to four, then storing all counter values requires 20kBits. To reduce the numbers of correlation results, a hard decision algorithm is utilized for preselection as indicated in Figure 4.2. With this algorithm up to 1280 entries of the *MASK_RAM* are selected as indices for which the averaged correlation values have to be stored with a soft output resolution of 16 bits within the *TEMP_RAM*. With this information the UMTS slot synchronization algorithm is performed a second time, but now only the indices refer-

enced by the *MASK_RAM* are of significance and only for these indices the soft output values are stored. It has to be evaluated whether the correlation process can be limited only to the addressed correlation indices or whether the correlation is performed without limitation and only the relevant indices by the *MASK_RAM* are stored. At the end of the soft decision algorithm, 1280 soft decision output values of 16 bits are available as input for the peak detector. The peak detector extracts up to 128 maximum values out of the *TEMP_RAM* and stores the peak information in combination with the peak index within the *RESULT_RAM*. The *RESULT_RAM* can be accessed by the DSP. The results of the pre-performed hard decision algorithm are the base for the soft decision algorithm. The firmware has the possibility to supervise the performance of the hard decision algorithm and to control the soft decision algorithm on the basis of these results. Figure 4.2 shows the architecture of the UMTS slot synchronization model with a preseletion part and a selection part.

- Hard decision algorithm is processed serially
  The same procedures as described above are performed by hardware in series without intervention of the DSP.

- Masked soft decision algorithm
  The algorithm is the same as the soft decision algorithm. The only difference is the ability of the DSP to edit the *MASK_RAM* for disabling defined correlation indices from calculating or disabling the storage of related soft decision output values. The results of a performed hard decision algorithm are the basis for the masked soft decision algorithm.

- Soft decision partial algorithm
  A derivation of the soft decision algorithm can be performed without the hard decision algorithm. The DSP determines a correlation interval within the slot for which correlations have to be calculated.

The *threshold0 control* shown in Figure 4.2 adjust the *threshold0* value once per slot and allows adaptation to the IQ channel. The block diagram of the *threshold0 control* part is shown in Figure 4.3.

The *threshold decision* value which indicates if the correlation result is higher then *threshold0* value, is accumulated within each slot and serves as a number of possible candidates for the selection part of the algorithm:

$$Number\_of\_Candidates0 = \sum_{i=1}^{n} threshold\_decision(i)$$

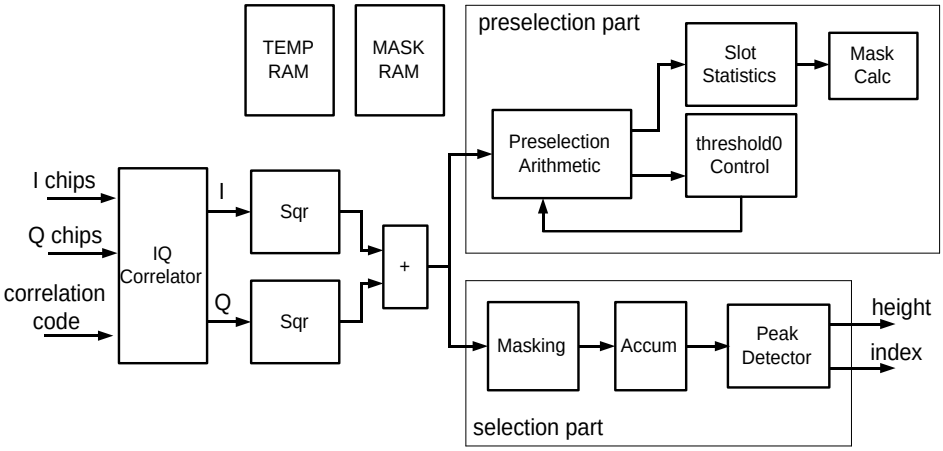

Figure 4.2: Architecture of UMTS slot synchronization model with a preselection part and a selection part

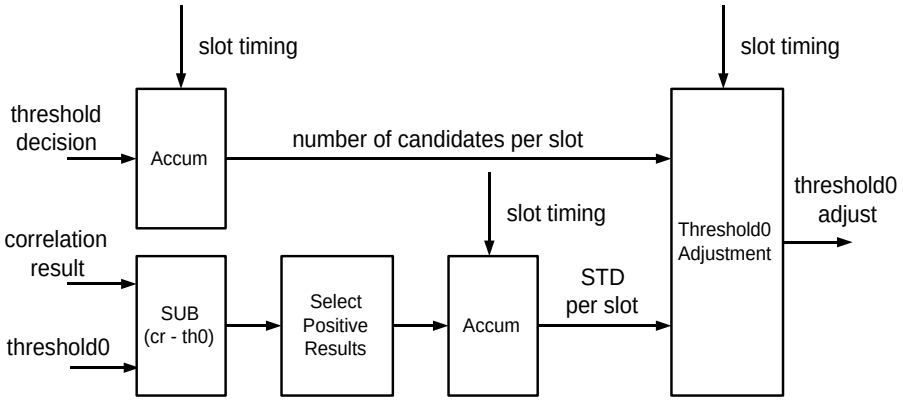

Figure 4.3: Block diagram of *threshold0 control*

At the same time, the standard deviation (STD) of each correlation value related to the *threshold0* value is evaluated:

$$STD = \sum_{i=1}^{n} (correlation\_result(i) - threshold0)$$

In both formulas n stands for the number of samples per slot.

Four situations are to distinguished, depending on the values of STD and *Number_of_Candidates0*.

1. $STD >$ a, *Number_of_Candidates0* $>$ b
   A broad distribution of correlation results causes a high number of possible

candidates. The *threshold0* value has to be incremented by c steps.

2. $STD \leq$ a, *Number_of_Candidates0* $\leq$ b
   A narrow distribution of correlation results causes a small number of possible candidates. The *threshold0* value has to be decremented by one step.

3. $STD >$ a, *Number_of_Candidates0* $\leq$ b
   Although a broad distribution of correlation results exists, only a low number of possible candidates is selected. The *threshold0* value has to be decremented by c steps.

4. $STD \leq$ a, *Number_of_Candidates0* $>$ b
   A narrow distribution of correlation results causes a high number of possible candidates. The *threshold0* value has to be incremented by one step.

The values of a,b,c are parameters to calculate *step_thres*.

4.3 Porting the functional description of the UMTS slot synchronization model from *prim* to untimed functional SystemC

The functional description of the UMTS slot synchronization model with a preseletion and selection part described in specialized CoCentric System Studio dataflow description language prim is shown in Figure 4.4.

Each block of the functional description in Figure 4.4 is described in the CoCentric System Studio data flow description language *prim*, and is ported to an equivalent untimed functional SystemC block. The channels between each block are implemented as FIFO channels with variable FIFO depth. The functional description of the UMTS slot synchronization model with a preselection and selection part described in untimed functional SystemC language (SystemC 2.0) is shown in Figure 4.5.

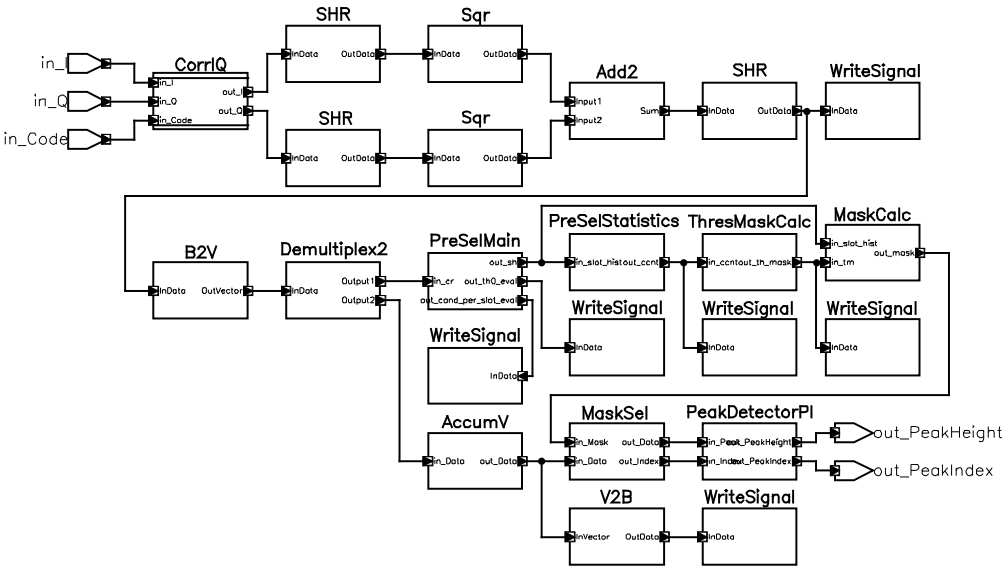

Figure 4.4: Functional model of the UMTS slot synchronization model in *prim* with preselection and selection part

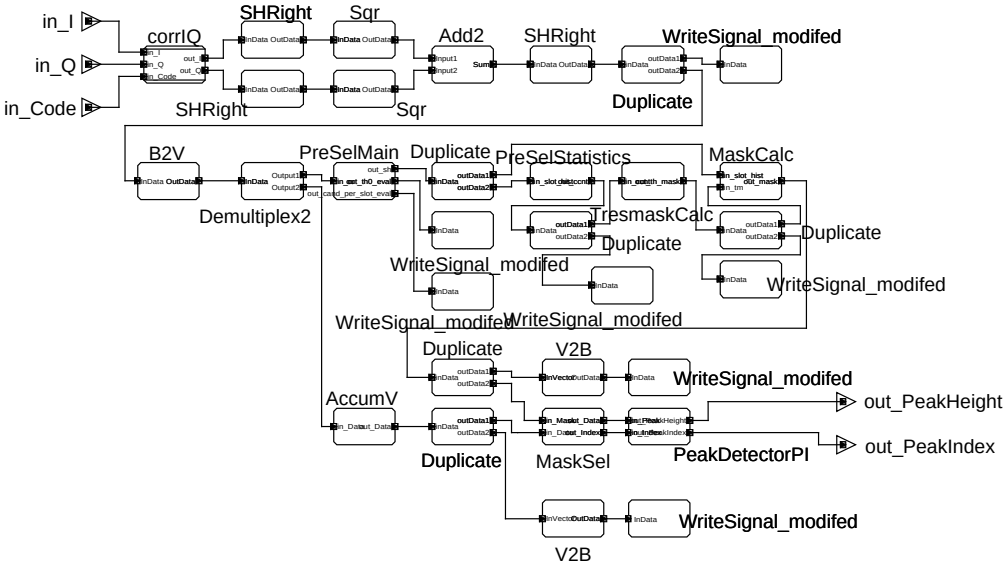

Figure 4.5: Functional model of the UMTS slot synchronization model in SystemC 2.0 with preselection and selection part

This model consists of the *corrIQ* block used instead of a matched filter. The *corrIQ* block is an IQ correlator and has no implemented memory of samples as the matched filter does. The memory block storing samples is outside of the functional description of the UMTS slot synchronization model. Using global memory makes it possible to reuse this memory block for different peripheral blocks of the UMTS base band chip. For example, the memory block can be reused for different cell search procedures. The CoCentric System Studio block schematic of the IQ correlator in the specialized dataflow language *prim* is shown in Figure 4.6. The same *corrIQ* block in SystemC 2.0 is shown in Figure 4.7.

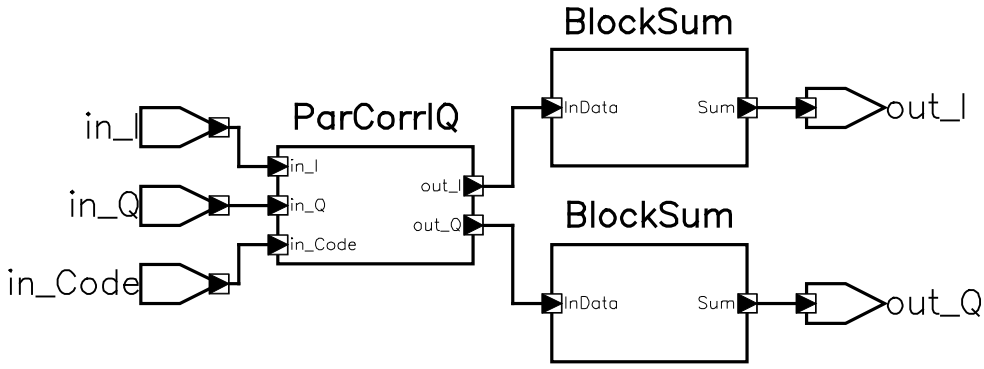

Figure 4.6: *corrIQ* block schematic in graphical *prim* description

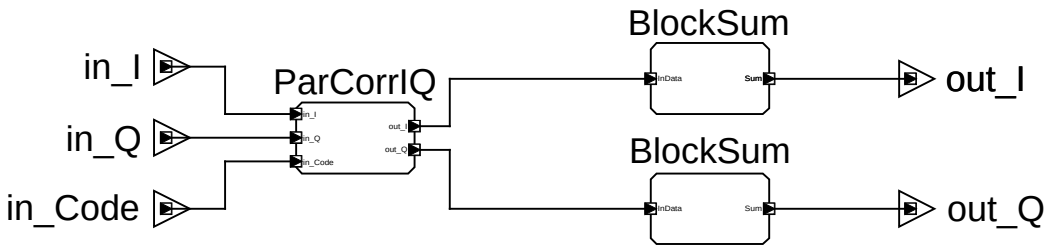

Figure 4.7: *corrIQ* block schematic in graphical SystemC description

A partial correlation is done in *ParCorrIQ* block and IQ correlation results are accumulated in *BlockSum* blocks. The corresponding *prim* model of *ParCorrIQ* block given here as an example reads as follows:

```
prim_model ParCorrIQ {
  // --- interface ---
  type_param T = long; // generic type parameter
  param structural int DataWidth = 16;
  port in T in_I[DataWidth];
  port in T in_Q[DataWidth];
  port in int in_Code[DataWidth];
```

```
  port out T out_I;
  port out T out_Q;
  // --- functional part ---
  main_action {
    T sum_I = (T)0, sum_Q = (T)0;
    read(in_I); read(in_Q); read(in_Code);
    for(int i = 0; i < DataWidth; i++) {
      sum_I += (T)in_Code[i] * (in_I[i] - in_Q[i]);
      sum_Q += (T)in_Code[i] * (in_I[i] + in_Q[i]);
    }
    out_I = sum_I; out_Q = sum_Q;
    write(out_I);   write(out_Q);
  }
}
```

The ported SystemC2.0 model of the *ParCorrIQ* block can be found in Appendix A.

In the complete CoCenctric System Studio schematic of the UMTS slot synchronization in SystemC 2.0, the *SHRight* block (shift right by constant) is used for data scaling. If the model is simulated with the floating point data type, block *SHRight* passes data from the input port to the output port without modification. The block *Demultiplex2* switches data flow of the correaltion peaks to the preselection or selection part. The preselection arithmetic and the *threshold0* control mechanism is implemented in block *PreSelMain*. The *prim* model of *PreSelMain* block is shown below:

```
prim_model PreSelMain {
  // --- interface ---
  type_param T = long; // generic type parameter
  param read_on_reset int data_width; // samples per slot
  param read_on_reset int num_of_acc;
  param read_on_reset T th0_par;  // threshold0
  param read_on_reset T st_par;   // step_thresh
  param read_on_reset long a_par; // a parameter
  param read_on_reset int b_par;  // b parameter
  param read_on_reset int c_par;  // c parameter
  port in T in_cr[data_width];
  port out int out_sh[data_width];
  port out T out_th0_eval;
  port out int out_cand_per_slot_eval;
  // --- functional part ---
  main_action {
    int TEMP_RAM[data_width], k = 1, sub = 0;
    T tmp_Threshold0 = th0_par;
    for(int i = 0; i < data_width; i++) TEMP_RAM[i] = 0;
    for(int j = 0; j < num_of_acc; j++) {
      int OvflDetected = 0, NumOfCandPerSlot = 0;
      T STD = 0;
      read(in_cr);
      for(int i = 0; i < data_width; i++) {
        T CorrResult = in_cr[i];
        int inc = (CorrResult > tmp_Threshold0) ? 1 : 0;
        int SlotHist = TEMP_RAM[i] + inc - sub;
        if(SlotHist == 15)
          OvflDetected = 1; // overflow
        else if(SlotHist < 0)
          SlotHist = 0;     // underflow
        TEMP_RAM[i] = SlotHist;
        NumOfCandPerSlot += inc;
        if((CorrResult - tmp_Threshold0) > 0)  STD++;
```

```
      }
      sub = OvflDetected;
      // --- threshold0 control part ---
      T arg2 = (STD > (T)a_par) ? st_par*c_par : st_par;
      if(NumOfCandPerSlot > b_par)
        tmp_Threshold0 += arg2;
      else
        tmp_Threshold0 -= arg2;
    }
    out_sh = TEMP_RAM;
    write(out_sh);
  }
}
```

The ported SystemC2.0 model of the *PreSelMain* shown in Figure 4.5 block can be found in Appendix A.

The *Slot Statistics* from Figure 4.2 is implemented in the block *PreSelStatistics* which is shown in Figure 4.5. The following source code of the ported module from the data flow description language *prim* to SystemC 2.0 can be found in Appendix A.

During the last slot of the preselection procedure, the number of entries of the *TEMP_RAM* contents which are above a predefined threshold value are evaluated by the block *PreSelStatistics*. Four threshold values, the *th_par[1..4]* (can be found in the SystemC2.0 source code of the *PreSelStatistics* block) are given. The amount of possible slot timings with four predefined values of significance are evaluated during the specified preselection interval.

The ported block *ThresMaskCalc* shown on Figure 4.5 computes, on the basis of different candidates per slot, threshold mask values which are used by the block *MaskCalc* for setting mask bits in the *MASK_RAM*.

The selection part of the UMTS slot synchronization model shown in Figure 4.2 is activated with values of *MASK_RAM*. The ported block *AccumV*, displayed in Figure 4.5, accumulates the energy of both correlation results, which are found from the imaginary and the real parts of the correlation, over several slots. The ported block *MaskSel* is responsible for getting the masked preselected indices out of the *MASK_RAM*. Finally, the ported block *PeakDetctorPI* extracts out of the accumulated peaks a certain number of the highest values. The indices of extracted peaks are candidates for slot boundary searching in the frame synchronization procedure. The ported blocks *B2V* and *V2B* are responsible the bit to vector and the vector to bit conversions. The block *Duplicate* duplicates the input signal to two output signals. The block *WriteSignal_modified* traces the data values of the incoming data and writes the data to a file which is responsible for verifying the design.

4.3.1 Test-bench for the UMTS slot synchronization model in untimed functional SystemC

The CoCentric System Studio schematic of the untimed functional test-bench for the UMTS slot synchronization model is shown in Figure 4.8.

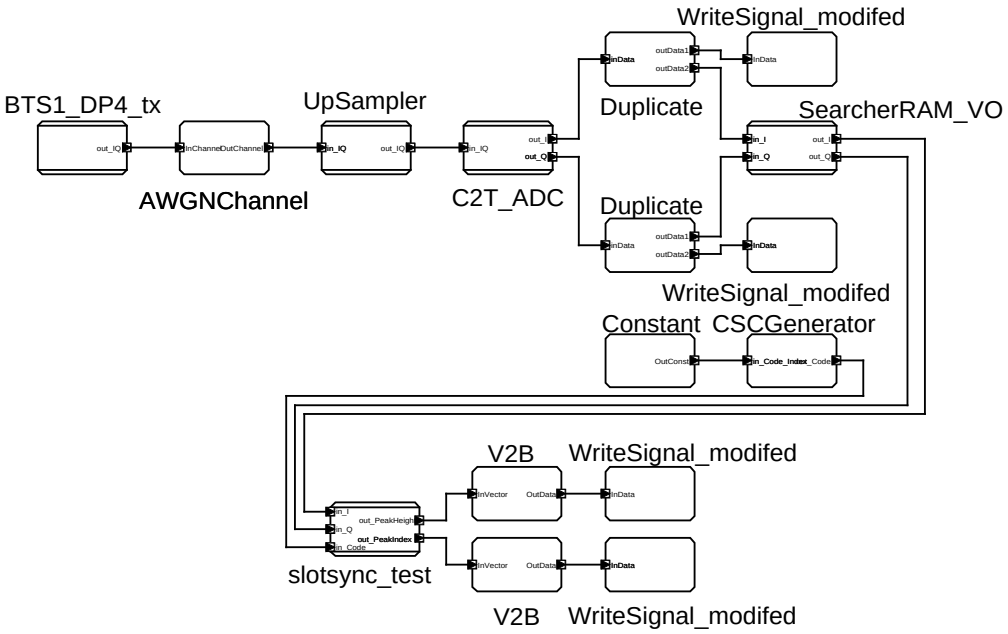

Figure 4.8: CoCentric System Studio test-bench for the UMTS slot synchronization model with preselection

The block *BTS1_DP4_tx* is a simplified Base Transceiver Station (BTS) with four different delay paths and with a root raised cosine filter at the output. The complex valued IQ signal is transmitted from the block *BTS1_DP4_tx* through an Additive White Gaussian Noise (AWGN) channel to the *UpSampler* block where the received signal is oversampled and bandlimited with a root raised cosine filter. The *C2T_ADC* splits the complex valued IQ signal to an I component and a Q component. This block also converts the analog data (floating point) to digital data (fixed point). When simulating with floating point data types, the ADC is bypassed. The *out_I* and *out_Q* outputs, shown in Figure 4.8, are used to stimulate the *SearcherRAM_VO* block which forms, as previously mentioned, a matched filter with the *Corr_IQ* block.

4.4 Verification of the functional description of the UMTS slot synchronization model in untimed functional SystemC

The verification of the functional description of the UMTS slot synchronization model in untimed functional SystemC is done via output files. After each block, data is written to an output file and manually compared to the data generated during simulation of the UMTS slot synchronization model. It is expected that the output files from the UMTS slot synchronization written in *prim* should be identical to the output files of the UMTS slot synchronization written in SystemC 2.0. After the block *C2T_ADC* in Figure 4.5, the I and Q signals are traced and compared to data from the *prim* model. The I and Q data after the block *C2T_ADC* are the same as the I and Q data generated from the test-bench of the UMTS slot synchronization written in *prim*. The I and Q test data are shown in Figure 4.9, respectively.

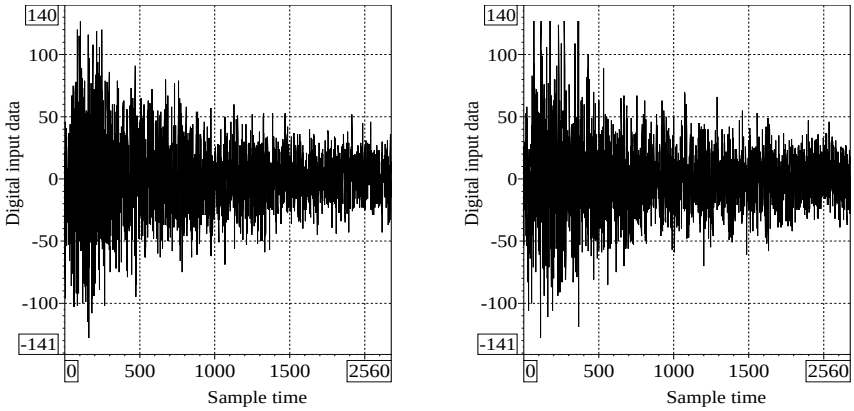

Figure 4.9: Data at SearcherRAM_VO input I_in (a) and at SearcherRAM_VO input Q_in (b) for the untimed functional SystemC (SystemC 2.0) test-bench

For the *BTS1_DP4_tx* block, delays of 0, 10, 20, 30 sample time units were defined. An oversampling ratio of two and a factor of 0.22 for root raised cosine filter were chosen in the *UpSampler* block. A signal to noise ratio of -6dB was selected for the *AWGN* channel block. Furthermore, for the *C2T_ADC* block an 8 bit ADC and a range from -3 to +3 were defined.

The data values after the *SHRight* block shown in Figure 4.5 are compared with the data values traced during simulation of the UMTS slot synchronization model written in *prim* and are the same. The data in the UMTS slot

synchronization after the *SHRight* block is shown in Figure 4.10 part (a). The evaluation for 20 accumulations of the candidates per slot of the UMTS slot synchronization model in untimed functional SystemC is shown in Figure 4.10 part (b) .

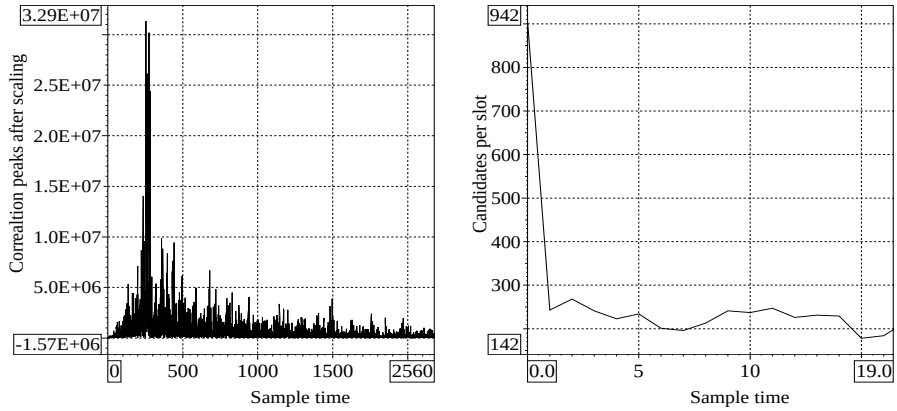

Figure 4.10: (a) Data in UMTS slot synchronization model after scaling in SystemC2.0 and (b) Evaluation of the candidates per slot for the UMTS slot synchronization model in in SystemC 2.0

Finally, the output of the UMTS slot synchronization model in SystemC 2.0 is compared with the results of the UMTS slot synchronization model in *prim*. The result of the UMTS slot synchronization in SystemC 2.0 and in *prim* are the same. Figure 4.11 shows the slot indices sorted according to their peak height.

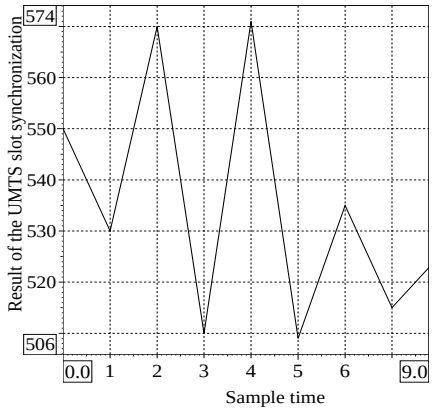

Figure 4.11: Slot indices sorted according to their peak height for the UMTS slot synchronization model in SystemC 2.0

The achieved results have shown that the implementation of the UMTS slot synchronization in *prim* is equivalent to the implementation of the UMTS slot synchronization in SystemC 2.0.

4.5 Summary

In this chapter, the porting of the UMTS slot synchronization model and test-bench from the CoCentric System Studio dataflow description language *prim* to untimed functional SystemC (SystemC 2.0) was shown. All the ported modules were verified according to their syntax and functions. Also, the output of the UMTS slot synchronization model was verified.

SystemC 2.0 together with CoCentric System Studio has proven its ability to model a complex design such as the UMTS slot synchronization at the algorithmic abstraction level. CoCentric System Studio with its graphical user interface makes it easy to draw functional blocks and connect them together with FIFO channels. The use of SystemC as a chip design language brings the benefit of refining the Design Under Test (DUT) to lower abstraction levels. The biggest advantage of using SystemC is the reusing of the test-bench for the design under test at different levels of abstraction.

Chapter 5

Transaction level modeling of the cell searcher algorithm

System architects working on System on Chip (SoC) designs have traditionally been hampered by the lack of a cohesive methodology for architecture evaluation and co-verification of hardware and software. These activities are crucial and must be addressed at an early stage to prevent a costly redesign effort later in the design cycle, which can adversely affect time to market. SystemC 2.0 as a design language facilitates the development of Transaction Level Models (TLMs), which are models of the hardware system components at a high level of abstraction. System architects can quickly develop these models and finish an executable specification of the hardware blocks as soon as the initial functional specification of the system has been decided on. The high speed of simulation of these transaction level models allows early development and verification of hardware dependent application software. Timing details can be incorporated into these models to allow performance estimation and architecture exploration. The modular nature of SystemC also promotes reuse of developed components from one system to another[Pas].

5.1 Transaction level modeling

Transaction level modeling is a high level approach to model digital systems where details of communication among modules are separated from details of the implementation of the functional units or of the communication architecture. Communication mechanisms such as busses or FIFOs are modeled as channels and are presented to modules using SystemC interface classes. Transaction requests take place by calling interface functions of these channel models encapsulating low level details of the information exchange. In other words,

at the TLM abstraction level, the emphasis is more on the functionality of the data transfers than on cycle accurate signals. This approach makes it easier for the system level designer to experiment without having to recode models. The following are some example design experiments allowing for transaction level models[TGS02]:

- Different bus architectures can be used to try and find out during simulation whether the chosen bus architecture fulfills the requirements of the design, such as bus load or data width.
- One master that represents the Central Processing Unit (CPU) can be connected to the design. A gigh level software model, respresenting the software intended for the actual system, can be executed on the CPU. The execution delays of software can be implemented. Calls to the device driver routines will be replaced by code calling the interface methods of the bus channel used. Also, bus traffic during reading and writing of the CPU can be observed and statistical traffic generation schemes or other approaches to generate the expected bus traffic may be used.
- The designer can link an Instruction Set Simulator (ISS) model into the SystemC simulation to model software running on a CPU or on a Digital Signal Processor (DSP). This allows the designer to execute actual object code, including a Real Time Operating System (RTOS) and assembly code, for the software part of the system along with the transaction level hardware models.
- As another example, a cycle accurate or cycle approximate model of hardware modules in SystemC can be created and can be attached to the chosen bus architecture. The design team can refine these hardware modules completely to Register Transfer Level (RTL) within the same SystemC environment, reusing the original test-bench from higher abstraction levels.

5.1.1 SystemC 2.0 for transaction level modeling

SystemC 2.0 can be used to model transaction level models as mentioned before. SystemC is a C++ library aimed specifically at system level modeling supporting all benefits of C++. SystemC 2.0 defines primary channels for communicating transactions, but leaves it to the user to define higher level SystemC channels suited to their design needs.

Communication in the transaction level modeling platform is ensured by using a primitive channel while the synchronization is based on events, as mentioned in Chapter 3. The necessary building blocks for process synchronization and communication refinement are (user defined) interfaces, ports and channels. An interface defines a set of methods but does not implement these methods. It is a pure virtual object without any data, to avoid the anticipation of implementation details. A channel implements one or more interfaces. A port is defined in terms of an interface type which means that the port can be used only with channels implementing this interface type.

5.2 Hardware/software split

5.2.1 Motivation for hardware/software split

Systems are becoming more and more complex and interrelated. At the component and subsystem levels, these complexities are perhaps still manageable. At the system level, though, especially early in the product life cycle, the complexities of the product technology and implementation require a robust, efficient and consistent methodology. In terms of product life cycle, hardware/software specific solutions begin to appear. Co-verification becomes the companion of co-design. A design must be verified against requirements to be successful.
The key advantage of the parallel design and verification of tightly coupled hardware and software is the early detection of mistakes. In the development of wireless systems, finding functional and requirement based mistakes as early as possible in the design phase is essential to a successful product. Poorly written, incomplete and changing specifications result in logical and functional errors leading to hardware and software design faults. These are both costly and time consuming to fix.

5.2.2 Predefined hardware/software split for the UMTS slot synchronization

A hardware/software split needs to be defined for the UMTS slot synchronization. In this complex design, which is a part of a UMTS baseband chip, performance critical parts have to be implemented in hardware. Minor changes in the standard 3GPP documents should be implementable and modules which would need a huge effort to be realized in hardware, have to be made in software. All other, not performance critical and not flexible modules of the UMTS slot synchronization can be implemented either in hardware or in software. The hardware/software split is chosen to minimize the bus load between the hard-

ware and software part of the slot synchronization.
All blocks of UMTS slot synchronization, shown in Figure 4.5, except the *ThresMaskCalc* block have to be implemented in hardware. Figure 5.1 shows the block schematic of the predefined hardware/software split for the UMTS slot synchronization.

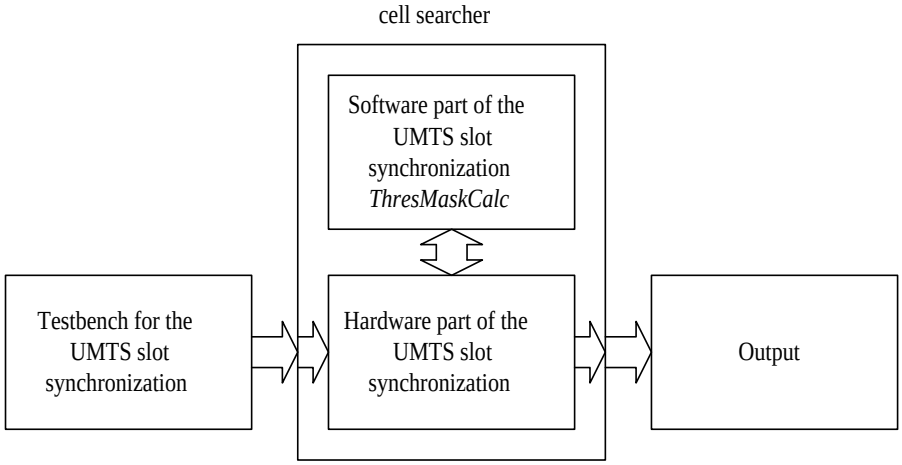

Figure 5.1: Block schematic for the hardware/software split in the cell searcher

5.3 Architecture

The software part of the UMTS slot synchronization can be implemented on a dedicated CPU or on a DSP. Thus, for this task a processor needs to be selected.
Based on the dominant market position of the ARM processor family, the ARM926 EJ-S processor was chosen for running the software part, the *ThresMaskCalc*, of the UMTS slot synchronization. Other reasons for choosing the ARM926 EJ-S processor were that this CPU is frequently used in embedded wireless systems. This specialized CPU, the ARM926 EJ-S, has an Advanced Microcontroller Bus Architecture (AMBA) bus interface which allows quick interacting with its peripherals.

5.3.1 Main features of the ARM926 EJ-S

The ARM926 EJ-S processor features a Jazelle technology enhanced 32-bit Reduced Instruction Set Core (RISC) CPU, flexible size instruction and data caches, Tightly Coupled Memory (TCM) interfaces and a Memory Management Unit (MMU). The ARM Jazelle technology for Java acceleration delivers an

excellent combination of Java performance on the 32-bit embedded reduced instruction set core architecture, which was also a reason why it is frequently used in embedded systems. This gives platform developers the freedom to run applications alongside established operating systems, middleware and application code on a single processor. The ARM926 EJ-S also provides separate instruction and data AMBA bus compliant AHB (Advanced High Performance) interfaces particularly suitable for multi layer AHB based systems. The ARM926 EJ-S with its low power needs also fulfills the power consumption constraints for wireless applications. Figure 5.2 shows the architecture of the ARM926 EJ-S.

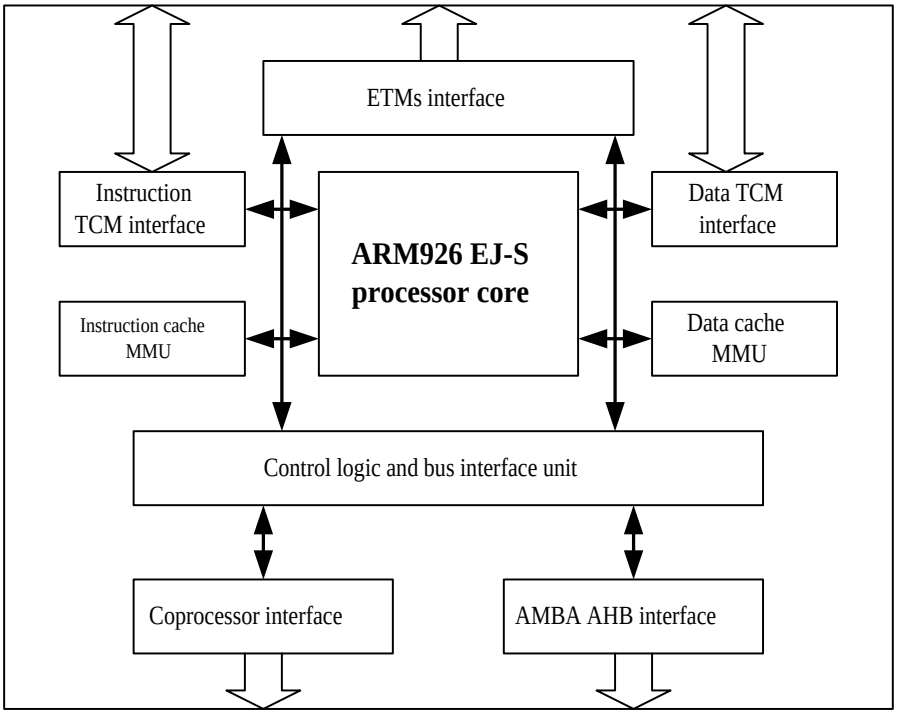

Figure 5.2: Architecture of the ARM926 EJ-S

Typical applications of the ARM926 EJ-S are:

- Smart phones, communicators and personal digital assistants (PDAs)
- 3G baseband and applications processor
- Platform operating system based devices
- Digital still camera
- Audio and video decoding
- Automotive infotainment

5.3.2 Main features of the AMBA

AMBA is an open standard, on chip bus specification offering a strategy for the interconnection and management of functional blocks that constitute a System on Chip (SoC). It facilitates development of embedded processors with one or more CPU/DSP processors and multiple peripherals. AMBA enhances the reusable design methodology by defining a common backbone for SoC modules. A typical AMBA based SoC consists of a high performance system bus and a peripheral bus[ARM].

- AMBA Advanced High performance Bus (AHB)
  The AHB system bus connects embedded processors such as an ARM core to high performance peripherals, Direct Memory Access (DMA) controllers, on-chip memory and interfaces. It is a high speed, high bandwidth bus supporting multi master bus management to maximize system performance. AHB serves the need for high performance SoC as well as aligning with current synthesis design flows.

- AMBA Advanced Peripheral Bus (APB)
  The AMBA peripheral bus is a simpler bus protocol designed for ancillary or general purpose peripherals. Connection to the system bus is via a bridge which helps reduce system power consumption. The AMBA peripheral bus supports ideally ancillary or general purpose peripherals such as timers, interrupt controllers, input/output ports, etc. The connection to the main system bus is achieved via a system to peripheral bus bridge.

5.4 Hardware/software co-design

The cell searcher concept implies that parts of the design have to be implemented in hardware and others in software. As previously mentioned, the *ThresMaskCalc* block has to be implemented in software. In Figure 5.3, the communication between hardware and software parts is shown from different modeling points of view.

Communication between hardware and software using the abstract bus protocol shown in Figure 5.3 part (a) is transaction based at the interface. As a result of using an abstract bus protocol, there is no notion of time, but rather just data sequences which can be translated to cycle accurate signals with the module bus interface. This is a model seen in Figure 5.3 part (a) of bus cycle accurate level of abstraction which was introduced in SystemC 2.0.

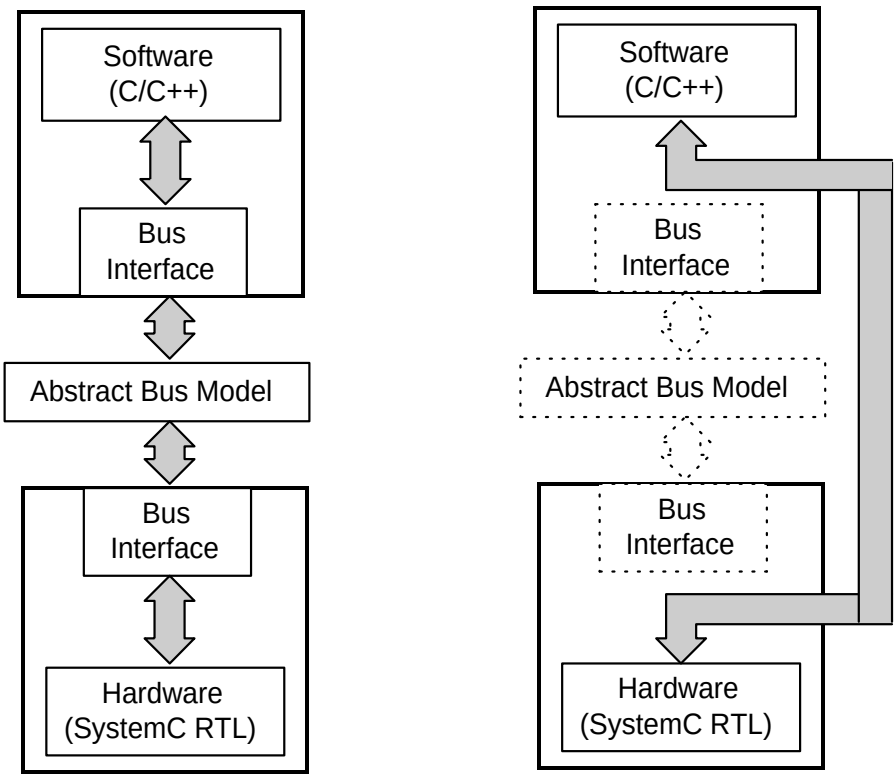

Figure 5.3: Block diagram of HW/SW co-design (a) using abstract bus model (b) using signals

5.4.1 Porting the given software part (*AND control model*) to SystemC 1.0 to implement the UMTS slot synchronization software part

The existing software part, written as an *AND control model*, explained in Chapter 4 (algorithmic and architectural modeling with CoCentric System Studio), was designed under an older version of CoCentric System Studio (version 2001-08) which did not support SystemC 2.0. Thus, it was not possible to use transaction level modeling, for which SystemC 2.0 is required. Only the calculation part of this existing software part, *AND control model*, was written in *prim* and can be found in more detail in Appendix B.

The software part of the UMTS slot synchronization as an *AND control model* is shown in Figure 5.4. All states of the software part are representing a control mechanism for the UMTS slot synchronization hardware part shown in Figure 5.5.

The *AND control model* communicates with the UMTS slot synchronization

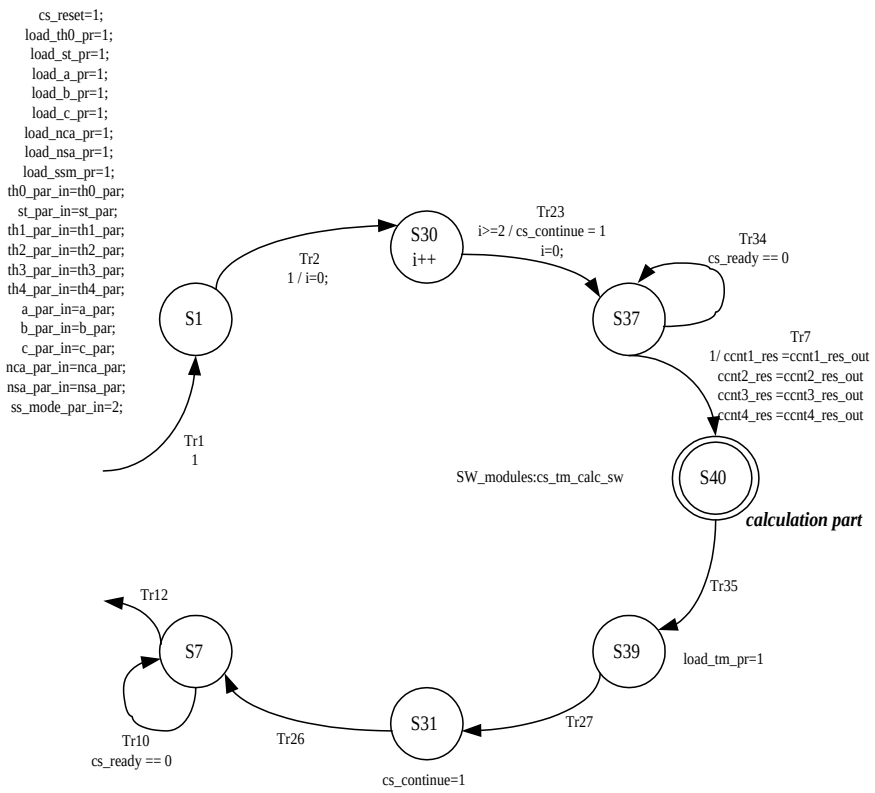

Figure 5.4: Software part (*AND Control Model*) of the UMTS slot synchronization for hardware/software co-design

hardware written in SystemC 1.0 via signals as shown in Figure 5.5.

The block *ConvertType* shown in Figure 5.5 is responsible for changing the data types to signals. The block *corr_iq* converts the incoming array of the I and Q data to signals and the block *cell_searcher_top* implements the UMTS slot synchronization hardware which is shown in Figure 5.5.

As a first step to port the *AND control model* of the UMTS slot synchronization software, the translation from this *AND control model* to a SystemC1.0 state machine is completed. The communication between software part and the hardware part, both written in SystemC, is done via signals. This is shown in Figure 5.3 part (b).

The ported *AND control model* together with its included calculation part of the block *ThresMaskCalc* in SystemC 1.0 can be found explained in more detail in Appendix B.

The hardware part of the UMTS slot synchronization co-simulation model (*prim* and SystemC 1.0) is shown on Figure 5.5.

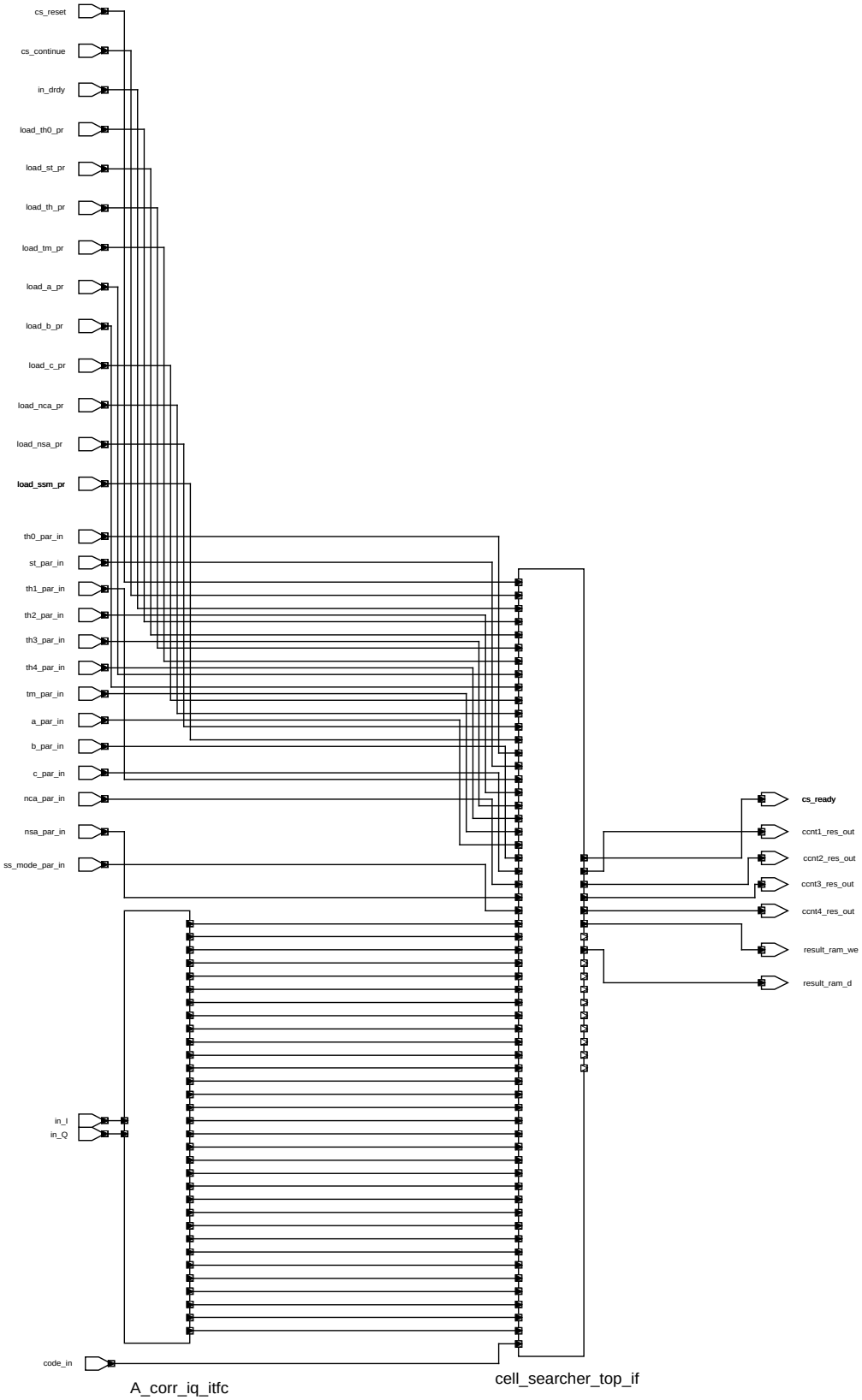

Figure 5.5: Hardware part of the UMTS slot synchronization for hardware/software co-cesign

To connect the UMTS slot synchronization model (hardware and software parts) in SystemC 1.0 to the existing functional test-bench, input and output modules need to be connected to the design under test to convert the FIFO channels to signals and also convert the output signals of the UMTS slot synchronization to FIFO channels. The UMTS cell searcher model with the predefined hardware/software split is shown in Figure 5.6.

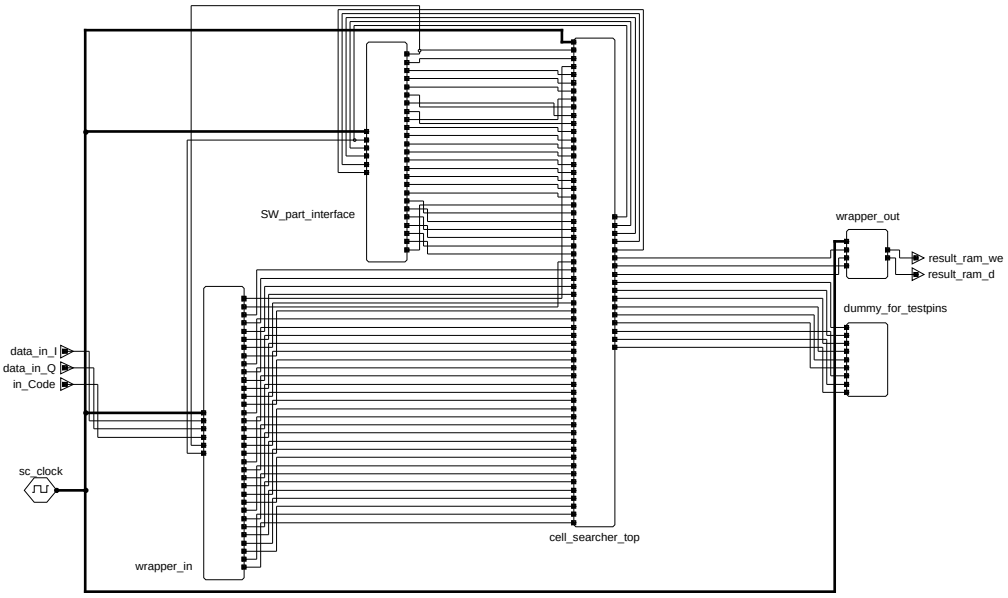

Figure 5.6: CoCentric System Studio schematic of the UMTS slot synchronization model with the predefined hardware/software split

The block *wrapper_in* in Figure 5.6 is responsible for converting the FIFO channel to a signal and also for feeding the test data as single signals instead of a vector to the UMTS slot synchronization model hardware part. The block *wrapper_out* in Figure 5.6 is responsible for converting the signal channels back to FIFO channels. The block *SW_part_interface* in Figure 5.6 represents the software part of the UMTS slot synchronization model and the block *cell_searcher_top* represents the hardware part of the UMTS slot synchronization model.

5.4.2 Verification of the UMTS slot synchronization model with the predefined hardware/software split in SystemC 1.0

The output files of the UMTS slot synchronization model with the predefined hardware/software split in SystemC 1.0 were compared to the output files of the UMTS slot synchronization model with the software part as the *AND con-*

trol model and the hardware part in SystemC 1.0. Also the output files from design under test were compared to a reference path, where a Dirac generator sent pulses over the multipath channel with the same coefficients (delay, attenuation) as in the untimed functional test-bench. It is expected that the output files of the UMTS slot synchronization with the predefined hardware/software split, in which the software part is modeled as an *AND control model* and the hardware part is in SystemC 1.0, are identical to the output files of the UMTS slot synchronization with the predefined hardware/software, where the hardware and software part are in SystemC 1.0. The test-bench for the UMTS slot synchronization, written in SystemC 2.0, is in Figure 5.7. Also shown is the UMTS slot synchronization model with predefined hardware/software split completely written in SystemC 1.0.

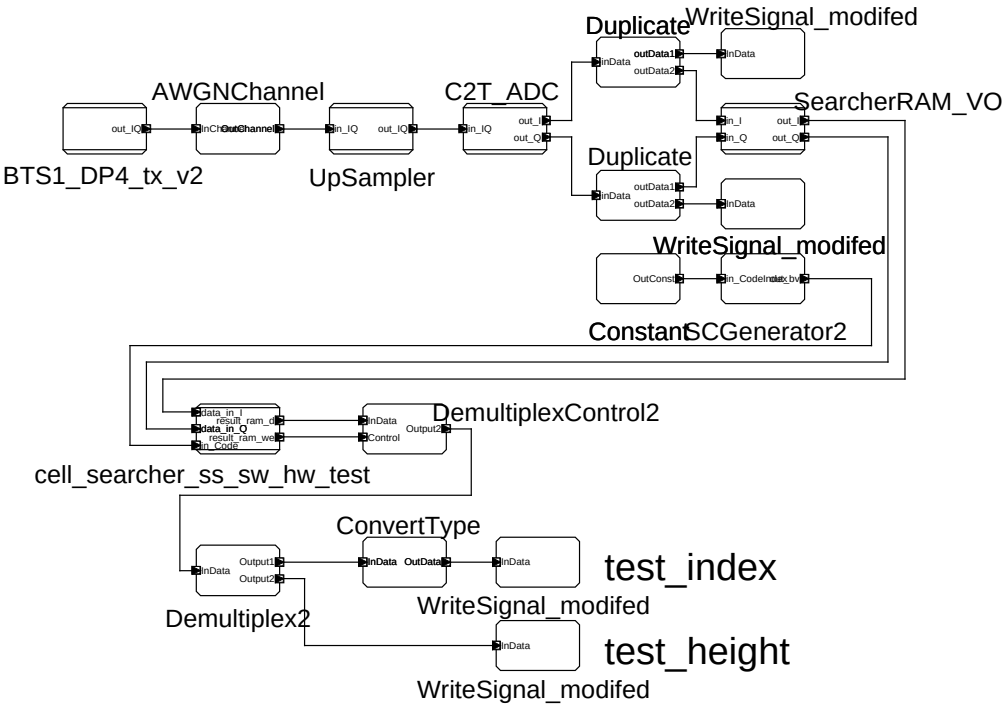

Figure 5.7: CoCenetric System Studio schematic of the UMTS slot synchronization model with the predefined hardware/software split in SystemC 1.0 and the untimed functional testbench for stimulating the design

Figure 5.8 shows the reference path utilized for comparing the result of the UMTS slot synchronization with output files obtained from simulations.

The output files of the reference path which are responsible for generating pulses to verify the results of the UMTS slot synchronization, written in Sys-

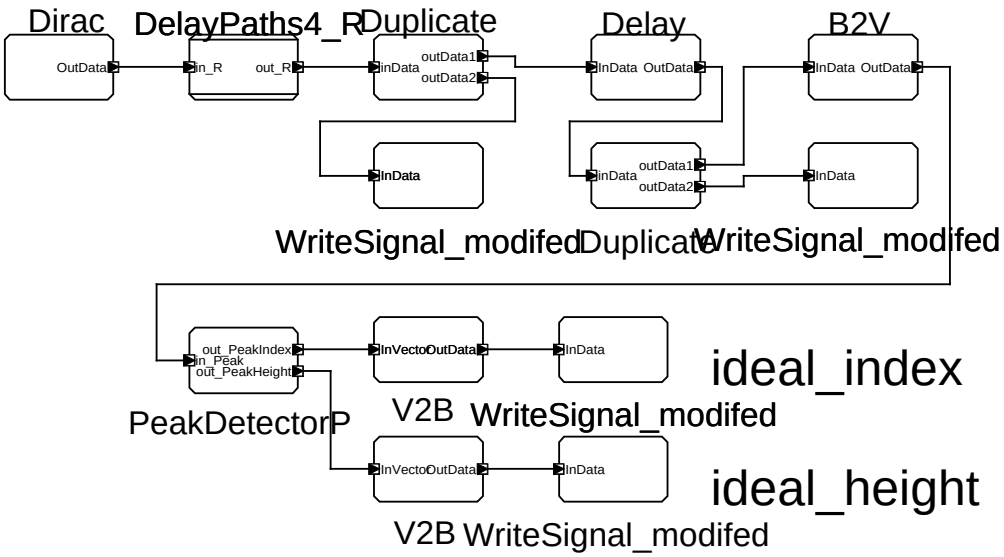

Figure 5.8: CoCenetric System Studio schematic of the reference path for verifying the results of the UMTS slot synchronization model with the predefined hardware/software split in SystemC 1.0 according the output files

temC 2.0, were compared to the output files of the reference path written in the specialized Synopsys language *prim*. Figure 5.9 shows the data which is generated in case of the SystemC 2.0 reference path.

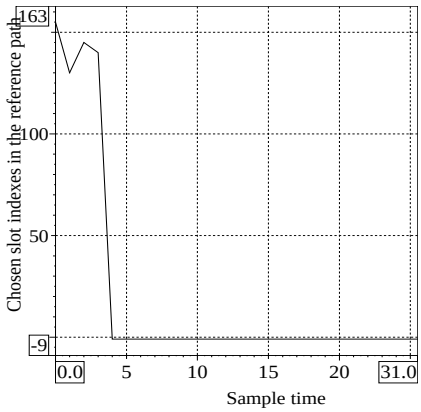

Figure 5.9: Slot indices which are sorted according to their peak height for the reference path to verify the output of the UMTS slot synchronization model in SystemC 2.0

The data stream generated from the SystemC test-bench is fed into the *cell_searcher_ss_sw_hw_dup_test1* block which represents the UMTS slot synchronization hardware and software parts. The I and Q data are fed into the

hardware part of the slot synchronization which communicates with the software part. The Figures 5.10, 5.11 and 5.12 show the handshake signals at the beginnung and at end of the UMTS slot synchronization procedure.

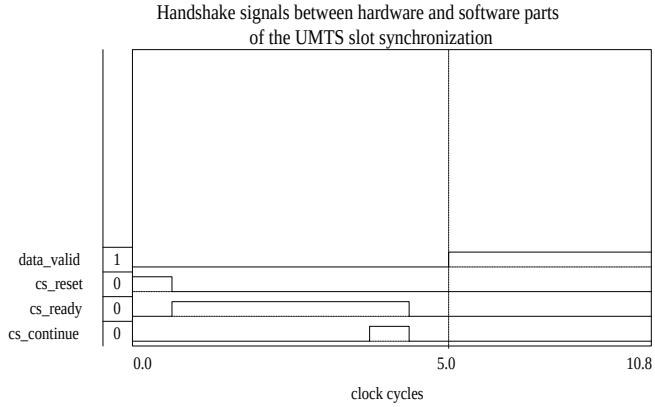

Figure 5.10: Handshake signal at the beginning of the communication between the hardware and the software written in SystemC 1.0 of the UMTS slot synchronization

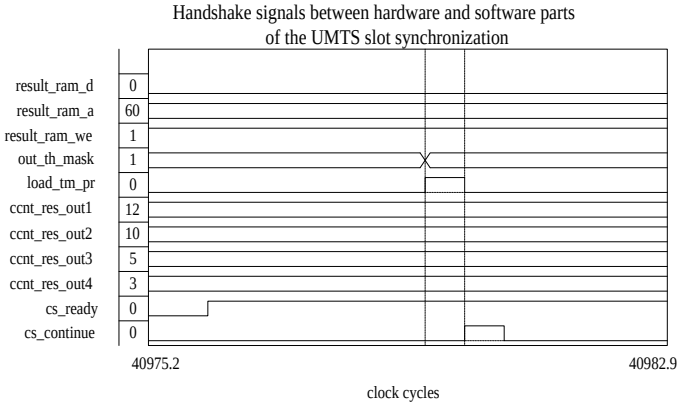

Figure 5.11: Handshake signal of the communication between the hardware and the software written in SystemC 1.0 of the UMTS slot synchronization

In Figure 5.10 the SystemC signal *data_valid* is responsible for signalling the hardware part that there are data values available at the input. The SystemC signal *cs_reset* implements a reset signal and the SystemC signals *cs_ready* and *cs_continue* are part of a handshake procedure between the software and hardware parts of the UMTS slot synchronization with the predefined hardware software split. The SystemC signals *ccnt_res_out1*, *ccnt_res_out2*, *ccnt_res_out3*

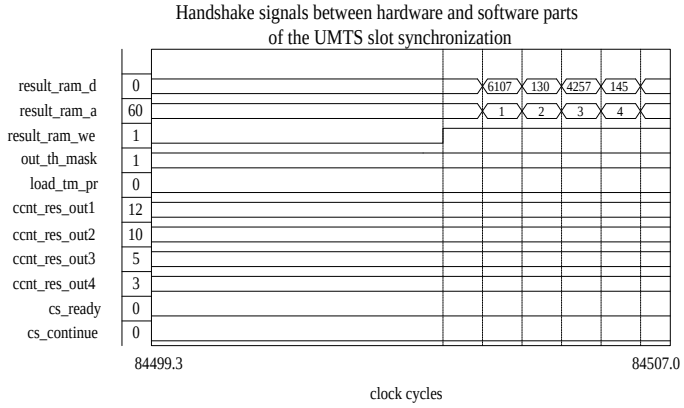

Figure 5.12: Handshake signal at the end of the communication between the hardware and the software written in SystemC 1.0 of the UMTS slot synchronization

and *ccnt_res_out4* shown in Figure 5.11, transmit data values to the software part which are needed for the calculation in the software part (*ThresMaskCalc*). The signals *load_tm_pr* and *out_th_mask* send the data values which are results of the software part, back to the hardware part of the UMTS slot synchronization. The SystemC signals *result_ram_d*, *result_ram_a* and *result_ram_we*, shown on Figures 5.11 and 5.12, are the data, the address and the write enable for the *RESULT_RAM* which is responsible for storing the results of the UMTS slot synchronization.
The output files of the UMTS slot synchronization with the predefined hardware/software split written in SystemC 1.0 were compared to the output files of the co-simulation of the UMTS slot synchronization with the predefined hardware/software split written in SystemC 1.0 with the test-bench in the Synopsys dataflow language *prim*. This comparison is shown in Figure 5.13.

In Figure 5.13 a small difference in the order of the detected slot indices is seen. The indices are the same, but in a different order, shown in Figure 5.13 part (a) and (b). The different order of the slot indices appears because of the different implementation of the sorter unit which is in the UMTS slot synchronization hardware. It is only important that the same slot indices were given as a result back to the test bench to achieve slot synchronization. The achieved results of the UMTS slot synchronization with software part as an *AND control model* and the hardware part in SystemC 1.0 are identical to the results of the UMTS slot synchronization with the hardware and software part in SystemC 1.0. This means that both designs were equivalent.

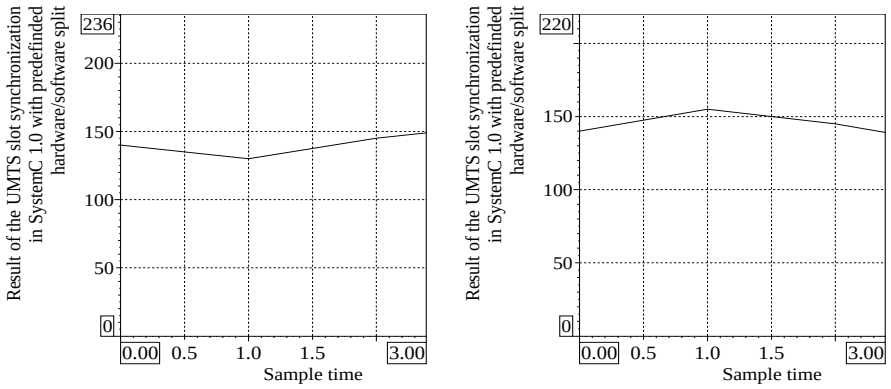

Figure 5.13: Slot indices which are sorted according to their peak values for the UMTS slot synchronization with the predefined hardware/software split (a) completely in SystemC (b) with the test-bench in *prim* (co-simulation)

5.4.3 Porting the given software part (*AND control model*) to the ARM926 EJ-S

The software part of the UMTS slot synchronization is ported by hand from the SystemC 1.0 implementation of the UMTS slot synchronization software part, which can be found in Appendix B, to a C program for the ARM926 EJ-S processor. This C code implements together with the bus interface shown in Figure 5.3 part (a) the state machine and calculation function of the UMTS slot synchronization software. The source code of this C-program for the ARM926 EJ-S can be found in Appendix C.

Also some definition files are written to define memory locations, data values, data types and variables for the UMTS slot synchronization. The source code of these files can also be found in Appendix C.

5.4.4 Design of the TLM bus interface between the hardware and software part of the UMTS slot synchronization

The TLM bus interface for the UMTS slot synchronization hardware needed to be defined as shown in Figure 5.3 part(a). The TLM bus interface of the software part (running on ARM926 EJ-S) is already included in the model of the ARM926 EJ-S processor. This source code of the TLM bus interface for the UMTS slot synchronization hardware can be found in Appendix D. In this source code, memory location to read from and to write to are defined. Also memory access procedures and transfer procedures over the transaction level

model of the AMBA bus are defined. This TLM bus interface builds an interface between the UMTS slot synchronization hardware model written in SystemC 1.0 and the abstract bus model of the AMBA bus at the abstraction level of transaction level modeling.

5.4.5 Verifying the software part of the UMTS slot synchronization together with the TLM bus interface between the hardware and software parts of the UMTS slot synchronization

The test environment for the UMTS slot synchronization software part is shown in Figure 5.14. The software part executes on a stand alone ARM926 EJ-S processor system. The block *SW_interface* represents the TLM bus interface between the UMTS slot synchronization software part (running on ARM926 EJ-S) and the UMTS slot synchronization hardware part. The ARM926 EJ-S processor model is seen in the block *DW_arm926ejs*. The blocks *WriteSignalClk* are used for verifying the output signal of the *SW_interface* block. Three blocks of *DW_AHB_Memory* are implementing stack memory, data memory and instruction memory. Two blocks of *DW_TCM_Memory* implement the TCM memory for data and instructions. The function of the abstract bus, the AHB bus, is seen in the block *DW_ahb_tlm*. For verifying the bus statistic and bus load, the block *DW_AHB_Monitor* is used. All of the blocks above described are shown in Figure 5.14.

5.4.6 Verifying the software and hardware parts of the UMTS slot synchronization at the abstraction level of transaction level modeling

The UMTS slot synchronization hardware was connected to the to block *SW_interface* which represents the TLM bus interface between the UMTS slot synchronization software part (running on ARM926 EJ-S) and the UMTS slot synchronization hardware part. The block *wrapper_in* in Figure 5.15 is responsible for converting the FIFO channel to a signal and also for feeding the test data as single signals instead of a vector to the UMTS slot synchronization model hardware part. The ARM926 EJ-S processor model is seen in the block *DW_arm926ejs*. Three blocks of *DW_AHB_Memory* are implementing stack memory, data memory and instruction memory. Two blocks of *DW_TCM_Memory* implement the TCM memory for data and instructions.

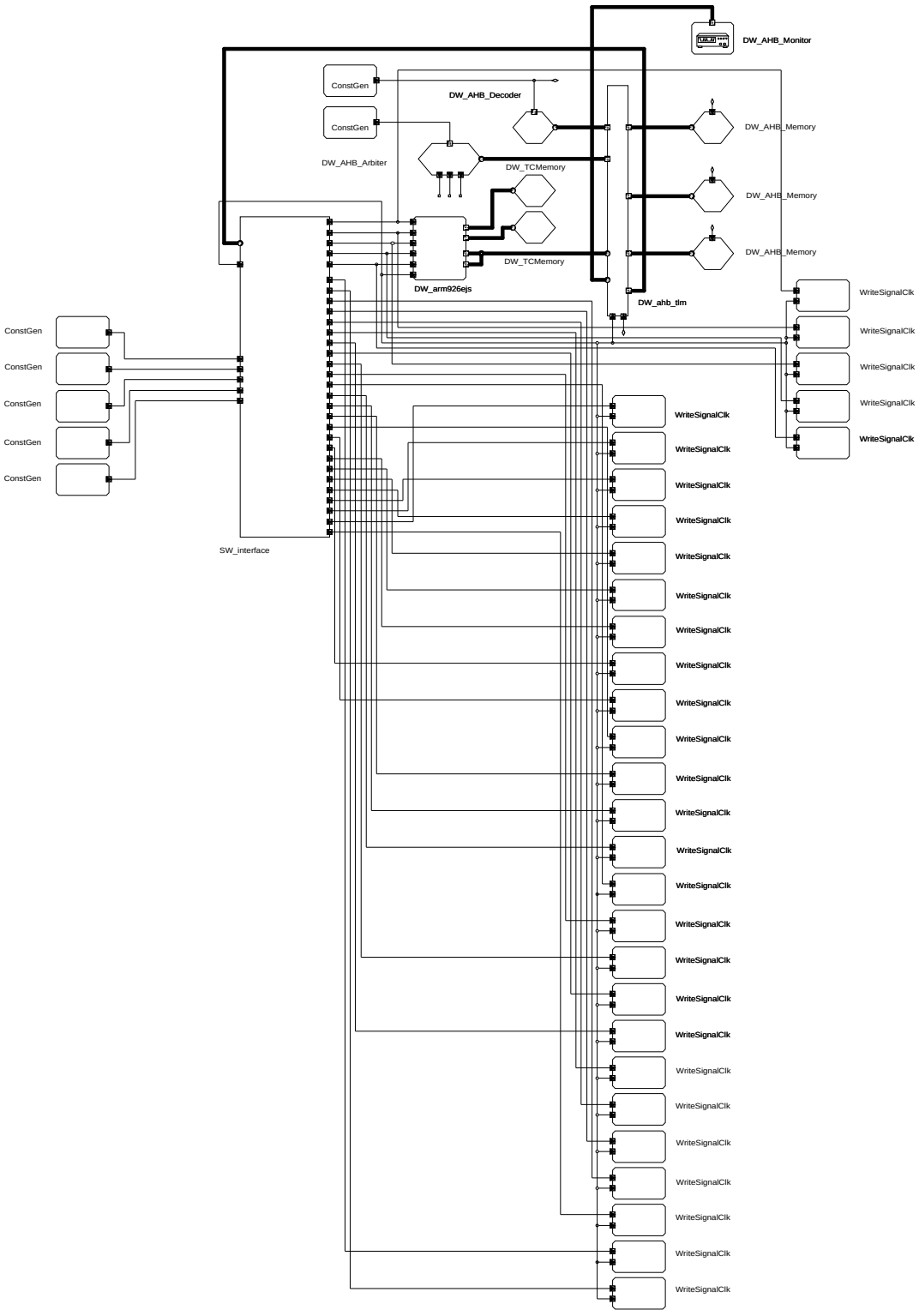

Figure 5.14: Environment for testing the UMTS slot synchronization software part running on ARM926 EJ-S

The function of the abstract bus, the AHB bus, is seen in the block *DW_ahb_tlm*. For verifying the bus statistic and bus load, the block *DW_AHB_Monitor* is used. The block *wrapper_out* in Figure 5.15 is responsible for converting the signal channels back to FIFO channels. All of the blocks above described are shown in Figure 5.15.

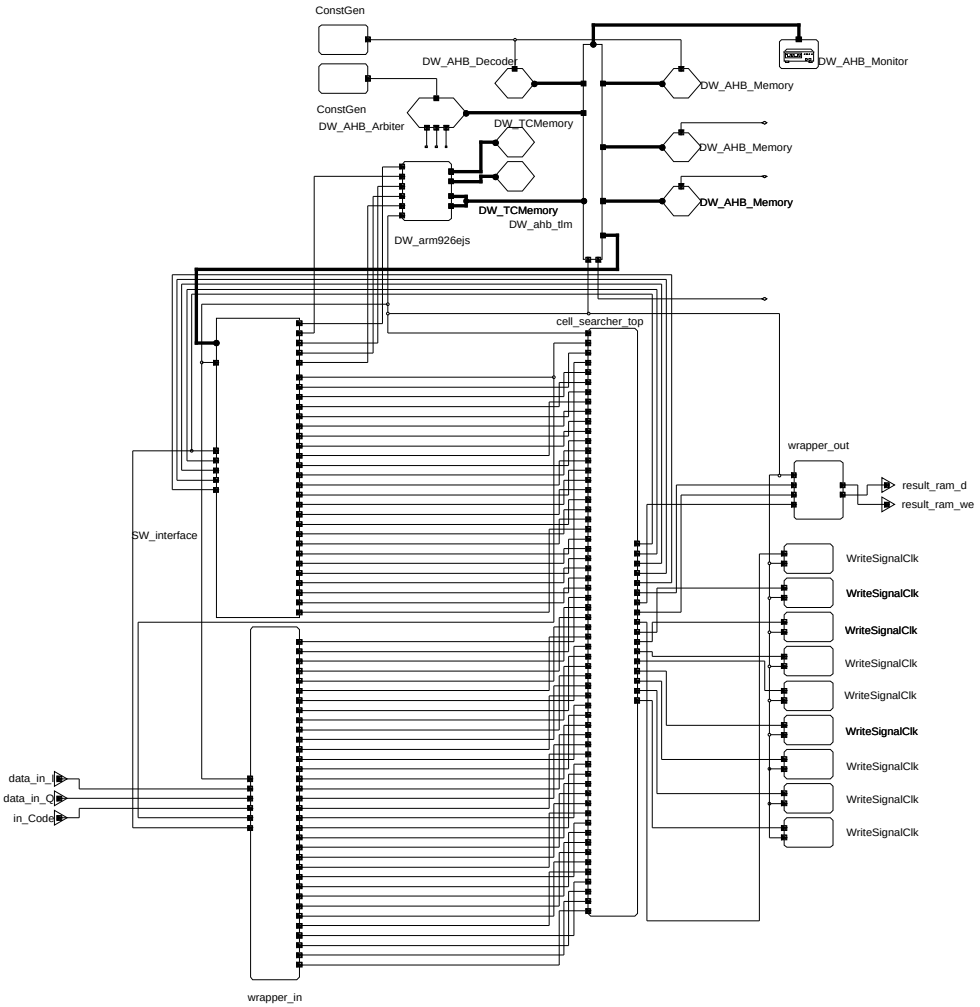

Figure 5.15: UMTS slot synchronization hardware and software part both connected to the AMBA bus

The test-bench at the functional abstraction level is connected to the design under test at the abstraction level of transaction level modeling which is the UMTS slot synchronization. This is shown in Figure 5.16. The block *slotsync_tlm_arm_dup* represents the design under test. The block *Demultiplex-Control2* processes the slot indices and the peak values of these slot indices as results (signal *result_ram_d* shown in Figure 5.15) of the UMTS slot synchronization if the *result_ram_we* signal shown in Figure 5.15 is true. The next

block, *Demultiplex2*, illustrated in Figure 5.16, extracts the slot indices and the peak heights of these slot indices and sends each of them to the block *WriteSignal_modified* to trace them.

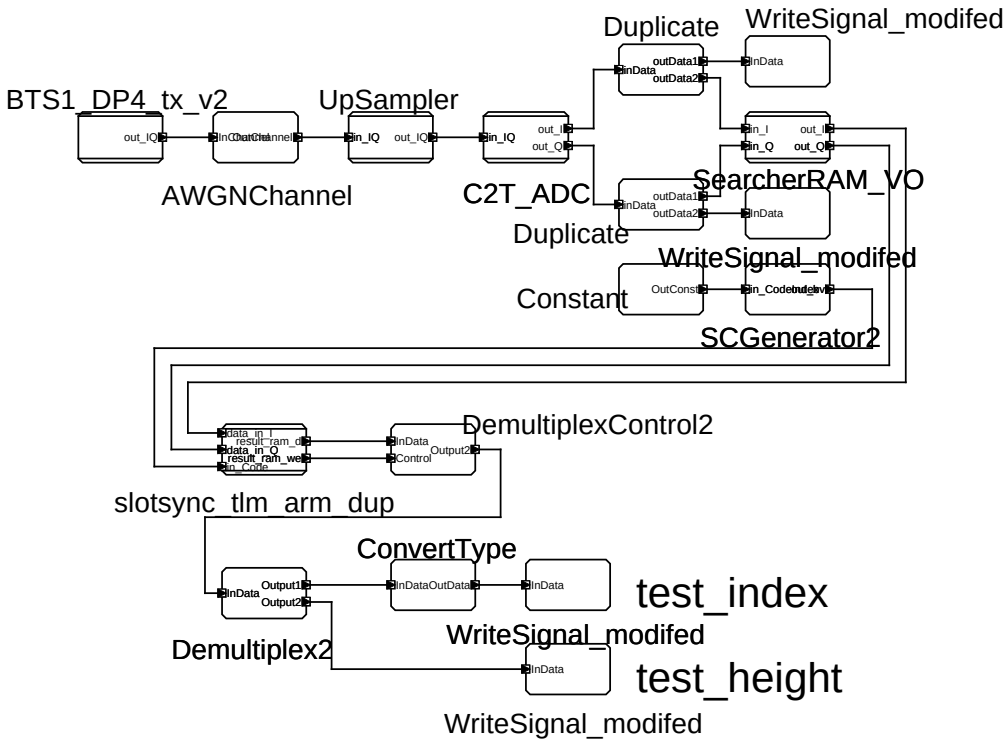

Figure 5.16: UMTS slot synchronization at the abstraction level of transaction level modeling together with the test-bench at the functional abstraction level

5.4.7 Verifying the chosen hardware/software split regarding the bus load of the AMBA bus in case of the UMTS slot synchronization

The software part of the UMTS slot synchronization communicates over the abstract bus with the hardware part of the UMTS slot synchronization. This means that the AMBA bus will be busy for several cycles. It has to be verified how much traffic will be on the AMBA bus.

Figure 5.17 shows idle, busy, read and write states in percentage which were traced from the block *DW_AHB_Monitor* in Figure 5.15 during the simulation of the UMTS slot synchronization procedure.

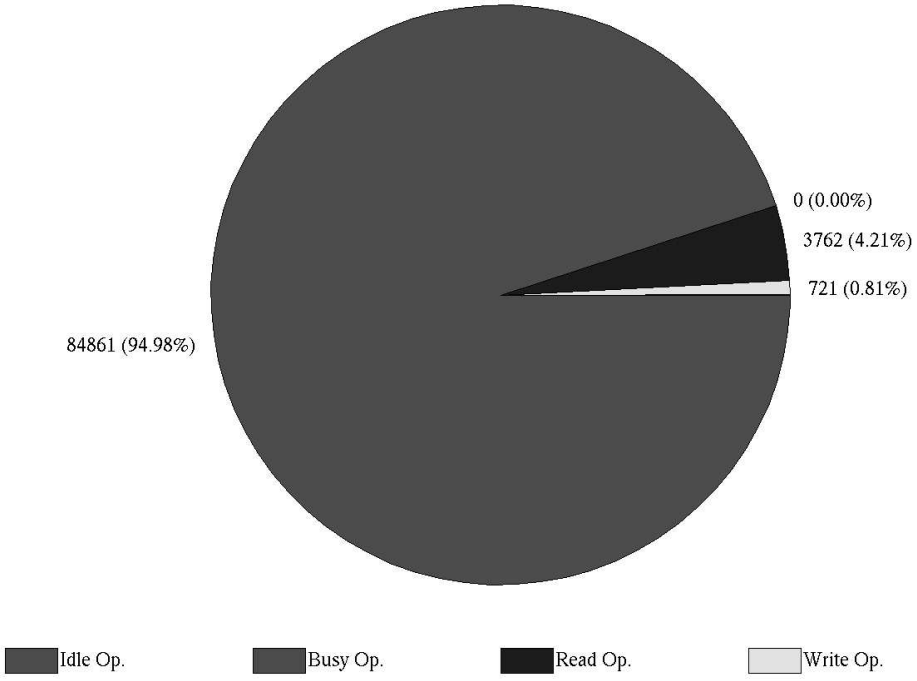

Figure 5.17: AMBA bus load reflects the traffic between hardware and software of the UMTS slot synchronization

In Figure 5.17, the AMBA bus is in the idle state for 84681 clock cycles (= 94.98%) during simulation of the UMTS slot synchronization. Exactly 3762 clock cycles or 4.21%, are used for reading from the UMTS slot synchronization hardware and 721 clock cycles, or 0.81% are used for writing from the UMTS slot software part to UMTS slot synchronization hardware part.

5.5 Summary

The software part of the UMTS slot synchronization was ported sucessfully to the predefined hardware ARM926 EJ-S and the AMBA bus. The hardware part of the UMTS slot synchronization together with the software at TLM are used for verifying the chosen hardware/software split. The bus load investigation of the AMBA bus for the chosen hardware/software split has shown that other parts of the baseband chip can also use the ARM926 EJ-S and the AMBA bus to run software as part of a dedicated block on this specialized architecture.

Chapter 6

Conclusion

In this diploma thesis, the implementation of the UMTS slot synchronization with preselection as a part of a UMTS base band chip was presented on different abstraction levels using the Synopsys system-level design tool CoCentric System Studio together with SystemC. The functional model of the UMTS slot synchronization and the test-bench for the UMTS slot synchronization were ported from the specialized functional description language *prim* to SystemC 2.0. The result of the UMTS slot synchronization in *prim* were compared to results of the UMTS slot synchronization in SystemC and were the same.
A second goal in this diploma thesis was the porting of the UMTS slot synchronization implementation from an *AND model* and SystemC 1.0 part to an existing predefined system architecture. A hardware/software split was chosen according to the cell searcher concept which implied that one specified calculation part will run as software on a digital signal processor, the ARM926 EJ-S. This software part communicates over an abstract bus model with the rest of the UMTS slot synchronization implemented in hardware. The hardware part was written in SystemC 1.0. The whole bus system and the ARM926 EJ-S was implemented on the abstraction level of transaction level modeling. Hence, a bus interface was defined to allow the UMTS slot synchronization hardware access to the abstract bus model realized as an AMBA bus. The results of this simulation were verified with to the results of the co-simulation of the UMTS slot synchronization in SystemC 1.0 and *prim*. The verification has shown that the results of the UMTS slot synchronization with the predefined hardware/software split are correct. Also the chosen bus and processor architecture together with its predefined hardware/software split has proven the ability to run the UMTS slot synchronization software.
SystemC was chosen as a common language approach for describing the UMTS slot synchronization. This new design language has proven, together with the

system level design tool CoCentric System Studio from Synopsys, to be well suited for designing SoCs.

Appendix A

SystemC 2.0 code of the UMTS slot synchronization at untimed functional abstraction level

A.1 Ported *ParCorrIQ* block from *prim* to SystemC2.0

The ported SystemC2.0 model of the *ParCorrIQ* block shown in Figure 4.7 is listed below:

```
// ParCorrIQ.h: header file
#ifndef __ParCorrIQ_h
#define __ParCorrIQ_h
#include <systemc.h>
#include "../../ArrayOfTemplate.h"
#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif

#ifdef CCSS_USE_SC_CTOR
#define CCSS_INIT_MEMBERS_PREFIX :
#undef CCSS_USE_SC_CTOR
#else
#define CCSS_INIT_MEMBERS_PREFIX ,
#endif

#define CCSS_INIT_MEMBERS  CCSS_INIT_MEMBERS_PREFIX \
  in_I("in_I") \
  , in_Q("in_Q") \
  , in_Code("in_Code") \
  , out_I("out_I") \
  , out_Q("out_Q")

template<class T = float , int DataWidth = 16  > class ParCorrIQ :
public sc_module {

public:
// parameters
```

```
// ports
sc_fifo_in<ArrayOfTemplate<T,DataWidth> > in_I;
sc_fifo_in<ArrayOfTemplate<T,DataWidth> > in_Q;
sc_fifo_in<ArrayOfTemplate<int,DataWidth> > in_Code;
sc_fifo_out<T> out_I;
sc_fifo_out<T> out_Q;

// initialize parameters
virtual void InitParameters() {
}
ArrayOfTemplate<T,DataWidth> temp_in_I, temp_in_Q;
ArrayOfTemplate<int,DataWidth> temp_in_Code;
void Do_ParCorrIQ()
{
  T sum_I,sum_Q;
  while(true)
  {
    sum_I = (T) 0;
    sum_Q = (T) 0;
    in_I -> read(temp_in_I);
    in_Q -> read(temp_in_Q);
    in_Code -> read(temp_in_Code);

    for (int i = 0; i < DataWidth; i++)
    {
      sum_I += (T)temp_in_Code.Data[i] * (temp_in_I.Data[i] - temp_in_Q.Data[i]);
      sum_Q += (T)temp_in_Code.Data[i] * (temp_in_I.Data[i] + temp_in_Q.Data[i]);
    }
    out_I -> write(sum_I);
    out_Q -> write(sum_Q);
  }
}

// default constructor
SC_CTOR(ParCorrIQ)
{
  InitParameters();
  // process declarations
  SC_THREAD(Do_ParCorrIQ);
}


}; // end module ParCorrIQ
#undef CCSS_INIT_MEMBERS_PREFIX
#undef CCSS_INIT_MEMBERS
#endif


// ParCorrIQ.cpp: source file

#include "ParCorrIQ.h"

// Explicit instantiation makes your compiler perform a complete processing
// of the template class with the given arguments. For example:
// template class ParCorrIQ<, int>;
```

A.2 Specialized written data type *ArrayOfTemplate* for sending array over a FIFO channel in SystemC2.0

In this code a specialized data type the *ArrayOfTemplate* was used:

```
#ifndef _ARRAY_OF_TEMPLATE
#define _ARRAY_OF_TEMPLATE

template<class T, int ArraySize> class ArrayOfTemplate {
  public:T Data[ArraySize];
  ostream& operator<<(ostream& s) {};
  friend bool operator==(const ArrayOfTemplate &x1, const ArrayOfTemplate &x2)
  {
    for (int i=0; i<ArraySize; i++)
      if (x1.Data[i] != x2.Data[i])
        return false;
    return true;
  };
};

template<class T, int ArraySize> ostream& operator<<
(ostream& s,ArrayOfTemplate<T,ArraySize> a)
{
  for (long iterator=0; iterator<ArraySize; iterator++)
    s << iterator << " " << a.Data[iterator] << "\n";
return s;
};

#endif
```

A.3 Ported *PreSelMain* block from *prim* to SystemC2.0

The ported SystemC2.0 model of the *PreSelMain* block shown in Figure 4.5 reads as follows:

```
// PreSelMain.h: header file

    #ifndef __PreSelMain_h
    #define __PreSelMain_h
    #include <systemc.h>
    #include "../../ArrayOfTemplate.h"
    #ifndef SYNTHESIS
    #include <ccss_systemc.h>
    #endif

    #ifdef CCSS_USE_SC_CTOR
    #define CCSS_INIT_MEMBERS_PREFIX :
    #undef CCSS_USE_SC_CTOR
    #else
    #define CCSS_INIT_MEMBERS_PREFIX ,
    #endif

    #define CCSS_INIT_MEMBERS  CCSS_INIT_MEMBERS_PREFIX \
      in_cr("in_cr") \
      , out_sh("out_sh") \
      , out_th0_eval("out_th0_eval") \
```

```
  , out_cand_per_slot_eval("out_cand_per_slot_eval")

template<class T = long , int data_width = 16  > class PreSelMain
: public sc_module {

public:
// parameters
CCSS_PARAMETER(bool, TestBenchMode);
CCSS_PARAMETER(T, st_par);
CCSS_PARAMETER(long, a_par);
CCSS_PARAMETER(int, b_par);
CCSS_PARAMETER(int, c_par);
CCSS_PARAMETER(T, th0_par);
CCSS_PARAMETER(int, num_of_acc);

// ports
sc_fifo_in<ArrayOfTemplate<T,data_width> > in_cr;
sc_fifo_out<ArrayOfTemplate<int,data_width> > out_sh;
sc_fifo_out<T> out_th0_eval;
sc_fifo_out<int> out_cand_per_slot_eval;

// initialize parameters
virtual void InitParameters() {
  bool _tmp_TestBenchMode = false;
  TestBenchMode.conditional_init(_tmp_TestBenchMode);
}
ArrayOfTemplate<int,data_width> TEMP_RAM;
ArrayOfTemplate<T,data_width> temp_in_cr;

// default constructor
SC_CTOR(PreSelMain)
{
  InitParameters();
  // process declarations
  SC_THREAD(Do_PreSelMain);
}
void Do_PreSelMain()
{
  int sub;
  T tmp_Threshold0;
  while (true)
  {
    tmp_Threshold0 = th0_par;
      sub = 0;
      for (int i=0;i < data_width;i++)
        TEMP_RAM.Data[i] = 0;

        for (int j=0;j < num_of_acc;j++)
        {
          int OvflDetected = 0;
          int NumOfCandPerSlot = 0;
          T STDevPerSlot = 0;

          in_cr -> read(temp_in_cr);
          for (int i=0; i< data_width; i++)
          {
            T CorrResult = temp_in_cr.Data[i];
            int inc = (CorrResult > tmp_Threshold0) ? 1 : 0;
            int SlotHist = TEMP_RAM.Data[i] + inc - sub;

            //overflow
            if (SlotHist == 15)
              OvflDetected = 1;

             //underflow
             if (SlotHist < 0)
```

```
                SlotHist = 0;

              TEMP_RAM.Data[i] = SlotHist;

              //Threshold control
              NumOfCandPerSlot +=inc;

              //Standard Deviation
              if ((CorrResult - tmp_Threshold0) > 0)
                STDevPerSlot++;
            }

            out_th0_eval -> write(tmp_Threshold0);
            out_cand_per_slot_eval -> write(NumOfCandPerSlot);
            sub = OvflDetected;

            // Threshold adjustment
            T arg2;
            if (TestBenchMode)
              arg2 = (STDevPerSlot > (T)a_par) ? st_par << c_par : st_par;
            else
              arg2 = (STDevPerSlot > (T)a_par) ? st_par * c_par : st_par;

            if(NumOfCandPerSlot > b_par)
              tmp_Threshold0 += arg2;
            else
              tmp_Threshold0 -= arg2;
          }
          // Output to PreSelStates
          out_sh -> write(TEMP_RAM);
        }
    }

};// end module PreSelMain
#undef CCSS_INIT_MEMBERS_PREFIX
#undef CCSS_INIT_MEMBERS

#endif



// PreSelMain.cpp: source file

#include "PreSelMain.h"

// Explicit instantiation makes your compiler perform a complete processing
// of the template class with the given arguments. For example:
// template class PreSelMain<, , , , , , , , int>;
```

In the block *PreSelMain*, there is a bit width of internal counter equal to four considered and therefore detection of overflow as well as underflow is implemented.

A.4 Ported *PreSelStatistics* block from *prim* to SystemC2.0

The following source code of the ported module from the data flow description language *prim* to SystemC 2.0 represents this *PreSelStatistics* block:

```
// PreSelStatistics.h: header file
```

```
#ifndef __PreSelStatistics_h
#define __PreSelStatistics_h
#include <systemc.h>
#include "../../ArrayOfTemplate.h"
#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif

#ifdef CCSS_USE_SC_CTOR
#define CCSS_INIT_MEMBERS_PREFIX :
#undef CCSS_USE_SC_CTOR
#else
#define CCSS_INIT_MEMBERS_PREFIX ,
#endif

#define CCSS_INIT_MEMBERS  CCSS_INIT_MEMBERS_PREFIX \
  in_slot_hist("in_slot_hist") \
  , out_ccnt("out_ccnt")

template<int data_width = 5120 , int num_th = 4  > class
PreSelStatistics : public sc_module {

public:
// parameters
CCSS_PARAMETER(int*, th_par);

// ports
sc_fifo_in<ArrayOfTemplate<int,data_width> > in_slot_hist;
sc_fifo_out<int> out_ccnt;

// initialize parameters
virtual void InitParameters() {
}

void Do_PreSelStatistics()
{
  ArrayOfTemplate<int,num_th> ccnt;
  ArrayOfTemplate<int,data_width> temp_in_slot_hist;
  while(true)
  {
    for (int i=0 ; i < num_th; i++)
      ccnt.Data[i] = 0;
    in_slot_hist -> read(temp_in_slot_hist);
    for (int i = 0 ; i < data_width; i++)
    {
      int SlotHist = temp_in_slot_hist.Data[i];
      for (int j = 0; j < num_th; j++)
        if (SlotHist > th_par[j]) ccnt.Data[j] ++;
    }
    for (int i=0; i < num_th; i++)
      out_ccnt -> write(ccnt.Data[i]);

  }
}

// default constructor
SC_CTOR(PreSelStatistics)
{
  InitParameters();
  // process declarations
  SC_THREAD(Do_PreSelStatistics);
}

}; // end module PreSelStatistics
#undef CCSS_INIT_MEMBERS_PREFIX
```

```
#undef CCSS_INIT_MEMBERS
#endif


// PreSelStatistics.cpp: source file

#include "PreSelStatistics.h"

// Explicit instantiation makes your compiler perform a complete processing
// of the template class with the given arguments. For example:
// template class PreSelStatistics<, 0, >;
```

Appendix B

SystemC 2.0 code of the UMTS slot synchronization at TLM abstraction level

B.1 Calculation part of the UMTS slot synchronization in a *AND control model* written in *prim*

The calculation part of the software part of the UMTS slot synchronization in a *AND control model* reads as follows:

```
prim_model cs_tm_calc_sw {
  param read_on_reset sc_uint<4> th1_par = 1;
  param read_on_reset sc_uint<4> th2_par = 2;
  param read_on_reset sc_uint<4> th3_par = 4;
  param read_on_reset sc_uint<4> th4_par = 8;
  param read_on_reset int num_required = 1280;
  param const int num_th = 4;
  port in int in_ccnt1;
  port in int in_ccnt2;
  port in int in_ccnt3;
  port in int in_ccnt4;
  port out sc_uint<4> out_th_mask;
  main_action
  {
    int ccnt[num_th], tmp_diff, cand_index_pos;
    read(in_ccnt1); ccnt[0] = in_ccnt1;
    read(in_ccnt2); ccnt[1] = in_ccnt2;
    read(in_ccnt3); ccnt[2] = in_ccnt3;
    read(in_ccnt4); ccnt[3] = in_ccnt4;
    bool pos_occur = 0;
    for(int i = 0; i < num_th; i++)
    {
      int diff = num_required - (int)ccnt[i];
      if(diff >= 0)
      {
        if(pos_occur)
        {
          if(tmp_diff > diff)
          {
```

```
            tmp_diff = diff;
            cand_index_pos = i;
          }
        }
        else
        {
          tmp_diff = diff;
          cand_index_pos = i;
          pos_occur = 1;
        }
      }
    }
    sc_uint<4> th_a[num_th], tmp;
    th_a[0] = th1_par;
    th_a[1] = th2_par;
    th_a[2] = th3_par;
    th_a[3] = th4_par;
    if(pos_occur)
      tmp = th_a[cand_index_pos];
    else
      tmp = th_a[0];
    out_th_mask = tmp;
    write(out_th_mask);
  }
}
```

B.2 Ported *AND control model* together with its included calculation part to SystemC 1.0

The ported *AND control model* together with its included calculation part of the block *ThresMaskCalc* to SystemC 1.0 reads as follows:

```
// SW_part_interface.h: header file
#ifndef __SW_part_interface_h
#define __SW_part_interface_h
#include <systemc.h>
#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif
#ifdef CCSS_USE_SC_CTOR
#define CCSS_INIT_MEMBERS_PREFIX :
#undef CCSS_USE_SC_CTOR
#else
#define CCSS_INIT_MEMBERS_PREFIX ,
#endif

#define CCSS_INIT_MEMBERS  CCSS_INIT_MEMBERS_PREFIX \
  clk("clk") \
  , cs_ready("cs_ready") \
  , ccnt1_res_in("ccnt1_res_in") \
  , ccnt2_res_in("ccnt2_res_in") \
  , ccnt3_res_in("ccnt3_res_in") \
  , ccnt4_res_in("ccnt4_res_in") \
  , cs_reset("cs_reset") \
  , cs_continue("cs_continue") \
  , load_th0_pr("load_th0_pr") \
  , load_st_pr("load_st_pr") \
  , load_th_pr("load_th_pr") \
  , load_a_pr("load_a_pr") \
  , load_b_pr("load_b_pr") \
  , load_c_pr("load_c_pr") \
  , load_tm_pr("load_tm_pr") \
```

```
  , load_nca_pr("load_nca_pr") \
  , load_nsa_pr("load_nsa_pr") \
  , load_ssm_pr("load_ssm_pr") \
  , th0_par_in("th0_par_in") \
  , st_par_in("st_par_in") \
  , th1_par_in("th1_par_in") \
  , th2_par_in("th2_par_in") \
  , th3_par_in("th3_par_in") \
  , th4_par_in("th4_par_in") \
  , a_par_in("a_par_in") \
  , b_par_in("b_par_in") \
  , c_par_in("c_par_in") \
  , nca_par_in("nca_par_in") \
  , nsa_par_in("nsa_par_in") \
  , ss_mode_par_in("ss_mode_par_in") \
  , out_th_mask("out_th_mask")

template<int num_th = 4  > class SW_part_interface : public
sc_module
{
  public:
  // parameters
  CCSS_PARAMETER(int, th0_par);
  CCSS_PARAMETER(int, st_par);
  CCSS_PARAMETER(int, th1_par);
  CCSS_PARAMETER(int, th2_par);
  CCSS_PARAMETER(int, th3_par);
  CCSS_PARAMETER(int, th4_par);
  CCSS_PARAMETER(int, a_par);
  CCSS_PARAMETER(int, b_par);
  CCSS_PARAMETER(int, c_par);
  CCSS_PARAMETER(int, nca_par);
  CCSS_PARAMETER(int, nsa_par);
  CCSS_PARAMETER(int, num_required);

  // ports
  sc_in<bool> clk;
  sc_in<bool> cs_ready;
  sc_in<int> ccnt1_res_in;
  sc_in<int> ccnt2_res_in;
  sc_in<int> ccnt3_res_in;
  sc_in<int> ccnt4_res_in;
  sc_out<bool> cs_reset;
  sc_out<bool> cs_continue;
  sc_out<bool> load_th0_pr;
  sc_out<bool> load_st_pr;
  sc_out<bool> load_th_pr;
  sc_out<bool> load_a_pr;
  sc_out<bool> load_b_pr;
  sc_out<bool> load_c_pr;
  sc_out<bool> load_tm_pr;
  sc_out<bool> load_nca_pr;
  sc_out<bool> load_nsa_pr;
  sc_out<bool> load_ssm_pr;
  sc_out<sc_uint<16> > th0_par_in;
  sc_out<sc_uint<16> > st_par_in;
  sc_out<sc_uint<4> > th1_par_in;
  sc_out<sc_uint<4> > th2_par_in;
  sc_out<sc_uint<4> > th3_par_in;
  sc_out<sc_uint<4> > th4_par_in;
  sc_out<sc_uint<32> > a_par_in;
  sc_out<sc_uint<13> > b_par_in;
  sc_out<sc_uint<3> > c_par_in;
  sc_out<sc_uint<5> > nca_par_in;
  sc_out<sc_uint<6> > nsa_par_in;
  sc_out<sc_uint<2> > ss_mode_par_in;
```

```
sc_out<sc_uint<4> > out_th_mask;

// initialize parameters
virtual void InitParameters()
{
  int _tmp_th0_par = 100;
  th0_par.conditional_init(_tmp_th0_par);
  int _tmp_st_par = 10;
  st_par.conditional_init(_tmp_st_par);
  int _tmp_th1_par = 1;
  th1_par.conditional_init(_tmp_th1_par);
  int _tmp_th2_par = 2;
  th2_par.conditional_init(_tmp_th2_par);
  int _tmp_th3_par = 4;
  th3_par.conditional_init(_tmp_th3_par);
  int _tmp_th4_par = 8;
  th4_par.conditional_init(_tmp_th4_par);
  int _tmp_a_par = 10000;
  a_par.conditional_init(_tmp_a_par);
  int _tmp_b_par = 32;
  b_par.conditional_init(_tmp_b_par);
  int _tmp_c_par = 1;
  c_par.conditional_init(_tmp_c_par);
  int _tmp_nca_par = 7;
  nca_par.conditional_init(_tmp_nca_par);
  int _tmp_nsa_par = 19;
  nsa_par.conditional_init(_tmp_nsa_par);
  int _tmp_num_required = 1280;
  num_required.conditional_init(_tmp_num_required);
}
int state;
int i;
int ccnt1_res, ccnt2_res, ccnt3_res, ccnt4_res;
int ccnt[num_th];
sc_uint<4> th_a[num_th];

void Do_SW()
{
  switch(state)
  {
    case 1:
    {
      cs_reset.write(true);
      load_th0_pr.write(true);
      load_st_pr.write(true);
      load_th_pr.write(true);
      load_a_pr.write(true);
      load_b_pr.write(true);
      load_c_pr.write(true);
      load_nca_pr.write(true);
      load_nsa_pr.write(true);
      load_ssm_pr.write(true);
      th0_par_in.write((sc_uint<16>) th0_par);
      st_par_in.write((sc_uint<16>) st_par);
      th1_par_in.write((sc_uint<4>) th1_par);
      th2_par_in.write((sc_uint<4>) th2_par);
      th3_par_in.write((sc_uint<4>) th3_par);
      th4_par_in.write((sc_uint<4>) th4_par);
      a_par_in.write((sc_uint<32>) a_par);
      b_par_in.write((sc_uint<13>) b_par);
      c_par_in.write((sc_uint<3>) c_par);
      nca_par_in.write((sc_uint<5>) nca_par);
      nsa_par_in.write((sc_uint<6>) nsa_par);
      ss_mode_par_in.write(2);
      load_tm_pr.write(false);
      cs_continue.write(false);
```

```
  state = 2;
  i = 0;
  break;
}

case 2:
{
  cs_reset.write(false);
  load_th0_pr.write(false);
  load_st_pr.write(false);
  load_th_pr.write(false);
  load_a_pr.write(false);
  load_b_pr.write(false);
  load_c_pr.write(false);
  load_nca_pr.write(false);
  load_nsa_pr.write(false);
  load_ssm_pr.write(false);
  cs_continue.write(false);
  if (i>=2)
  {
    i = 0;
    cs_continue.write(true);
    state = 31;
    break;
  }
  i++;
  break;
}

case 31:
{
  cs_continue.write(false);
  state = 3;
  break;
}

case 3:
{
  if (cs_ready.read() == true)
  {
    ccnt1_res =  ccnt1_res_in.read();
    ccnt2_res =  ccnt2_res_in.read();
    ccnt3_res =  ccnt3_res_in.read();
    ccnt4_res =  ccnt4_res_in.read();
    state = 4;
  }
  break;
}

case 4:
{
  ccnt[0] = ccnt1_res;
  ccnt[1] = ccnt2_res;
  ccnt[2] = ccnt3_res;
  ccnt[3] = ccnt4_res;
  int tmp_diff;
  int cand_index_pos;
  bool pos_occur = 0;
  for(int i = 0; i < num_th; i++)
  {
    int diff = num_required - (int) ccnt[i];
    if (diff >= 0)
    {
      if(pos_occur)
      {
        if(tmp_diff > diff)
```

```
            {
              tmp_diff = diff;
              cand_index_pos = i;
            }
          }
          else
          {
            tmp_diff = diff;
            cand_index_pos = i;
            pos_occur = 1;
          }
        }
      }
      th_a[0] = (sc_uint<4>) th1_par;
      th_a[1] = (sc_uint<4>) th2_par;
      th_a[2] = (sc_uint<4>) th3_par;
      th_a[3] = (sc_uint<4>) th4_par;

      if (pos_occur)
      {
        out_th_mask -> write(th_a[cand_index_pos]);
      }
      else
      {
        out_th_mask -> write(th_a[0]);
      }
      load_tm_pr.write(true);
      state = 5;
      break;
    }

    case 5:
    {
      load_tm_pr.write(false);
      cs_continue.write(true);
      state = 61;
      break;
    }

    case 61:
    {
      cs_continue.write(false);
      state = 6;
      break;
    }

    case 6:
    {
      if (cs_ready.read() == true)
      {
        state = 7;
      }
      break;
    }

    case 7:
    {
      break;
    }
  }
}
// default constructor
SC_CTOR(SW_part_interface)
{
  InitParameters();
  // process declarations
```

```
    SC_METHOD(Do_SW);
    sensitive_pos << clk;
    state = 1;
  }

}; // end module SW_part_interface
#undef CCSS_INIT_MEMBERS_PREFIX
#undef CCSS_INIT_MEMBERS
#endif



// SW_part_interface.cpp: source file
#include "SW_part_interface.h"

// Explicit instantiation makes your compiler perform a complete processing
// of the template class with the given arguments. For example:
// template class SW_part<0>;
```

Appendix C

C - source code for the UMTS slot synchronization software part running on ARM926 EJ-S

C.1 Main application file C - source code file "*examples_ss.c*" for the UMTS slot synchronization software part running on ARM926 EJ-S

The main application source file "*examples_ss.c*" code reads as follows:

```
//examples_ss.c
#include "definition.h"
#include "examples_ss.h"
#include <stdio.h>
#include <string.h>

/* These are used for interrupt testing */
int intr_level = 0;
int intr_testing = 0;
int intr_received = 0;
int sio_testing = 0;
int test_fail = 0;
int loop_iter = 2500;
int PULSE_TEST = 0;


extern int init_interrupts(void);
extern int fiqtest(void);
extern int irqtest(void);
extern int dataaborttest(void);
extern int resettest(void);

#define ENDIANTEST    1
#define RAM8TEST      2
#define RAM16TEST     3
```

```
#define RAM32TEST     4
#define ITCMTEST      5
#define DTCMTEST      6
#define FIQTEST       7
#define IRQTEST       8
#define DATAABORTTEST 9
#define RESETTEST     10

int fail_count = 0;
int s2_count = 0;
U32 ccnt1_res = 0;
U32 ccnt2_res = 0;
U32 ccnt3_res = 0;
U32 ccnt4_res = 0;
U32 data2 = 0;

pU32 p_ccnt1_res = &ccnt1_res;
pU32 p_ccnt2_res = &ccnt2_res;
pU32 p_ccnt3_res = &ccnt3_res;
pU32 p_ccnt4_res = &ccnt4_res;
pU32 p_data2 = &data2;

int failure(int test_num)
{
  fail_count++;
  /* Use the debugger to print out the test_num that failed!
  */
  return(test_num);
}

int passed()
{
  /* Use the debugger to print out global fail_count.  It should
  * be 0 if everything passed...
  */
  return(0);
}

#define LOCALADDR 0x90000000

int endiantest(unsigned int addr)
{
  unsigned int *wptr = (unsigned int *) addr;
  unsigned short *sptr = (unsigned short *) addr;
  unsigned char *bptr = (unsigned char *) addr;
  unsigned int wdata;
  unsigned short sdata;
  unsigned char bdata;
  int big = 0;
  *wptr = 0x12345678;
  wdata = *wptr;
  if (wdata != 0x12345678)
  {
    failure(ENDIANTEST);
  }
  /*
  * Test reads
  */
  bdata = *bptr;
  if (big && (bdata != 0x12))
  {
    failure(ENDIANTEST);
  }
  else
  if (!big && (bdata != 0x78))
  {  /* little endian */
```

```
  failure(ENDIANTEST);
}
bdata = *(bptr+1);
if (big && (bdata != 0x34))
{
  failure(ENDIANTEST);
}
else
if (!big && (bdata != 0x56))
{ /* little endian */
  failure(ENDIANTEST);
}
bdata = *(bptr+2);
if (big && (bdata != 0x56))
{
  failure(ENDIANTEST);
}
else
if (!big && (bdata != 0x34))
{ /* little endian */
  failure(ENDIANTEST);
}
bdata = *(bptr+3);
if (big && (bdata != 0x78))
{
  failure(ENDIANTEST);
}
else
if (!big && (bdata != 0x12))
{ /* little endian */
  failure(ENDIANTEST);
}
sdata = *(sptr+1);
if (big && (sdata != 0x5678))
{
  failure(ENDIANTEST);
}
else
if (!big && (sdata != 0x1234))
{
  failure(ENDIANTEST);
}
/*
* Test writes
*/
*sptr = 0x8765;
*(sptr+1) = 0x4321;
wdata = *wptr;
if (big && (wdata != 0x87654321))
{
  failure(ENDIANTEST);
}else
if (!big &&(wdata != 0x43218765))
{ /* little */
  failure(ENDIANTEST);
}
*bptr = 01;
*(bptr+1) = 2;
*(bptr+2) = 3;
*(bptr+3) = 4;
wdata = *wptr;
if (big && (wdata != 0x01020304))
{
  failure(ENDIANTEST);
}
else
```

```
  if (!big && (wdata != 0x04030201))
  {
    failure(ENDIANTEST);
  }
  *wptr = 0x12345678;
  return 1;
}


int dtcmtest()
{
  int total, size;
  U32 answer;
  pU32 i;
  pU32 pAnswer = &answer;
  U32 data;
  /* Test each memory address by writing and reading a unique value
  * from each 8 bit data location.
  */
  data = 1;
  size = 0;
  total = 0;
  for (i=(pU32) DTCMSTART; i < (pU32)(DTCMSTART + CHK_ADDRESSES);i++)
  {
    MWRITE32(data, i);
    MREAD32(pAnswer, i);
    MWRITE32((U32) 0x0, i);
    if (answer != data)
    {
      failure(DTCMTEST);
    }
    data++;
    size++;
    total++;
  }
  return(0);
}

int itcmtest()
{
  int total, size;
  U32 answer;
  pU32 i;
  pU32 pAnswer = &answer;
  U32 data;
  /* Test each memory address by writing and reading a unique value
  * from each 8 bit data location.
  */
  data = 1;
  size = 0;
  total = 0;
  for (i=(pU32) ITCMSTART; i < (pU32)(ITCMSTART + CHK_ADDRESSES);i++)
  {
    MWRITE32(data, i);
    MREAD32(pAnswer, i);
    MWRITE32((U32) 0x0, i);
    if (answer != data)
    {
      failure(ITCMTEST);
    }
    data++;
    size++;
    total++;
  }
  return(0);
}
```

```
int S1()
{
  U32 data;
  /* State one for the software part of the UMTS slot synchronization*/
  data = (U32) th0_par;
  MWRITE32(data,(pU32) INTERFACESTART+0x1);
  data = (U32) st_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x2);
  data = (U32) th1_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x3);
  data = (U32) th2_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x4);
  data = (U32) th3_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x5);
  data = (U32) th4_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x6);
  data = (U32) a_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x7);
  data = (U32) b_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x8);
  data = (U32) c_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0x9);
  data = (U32) nca_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0xa);
  data = (U32) nsa_par;
  MWRITE32(data,((pU32) INTERFACESTART)+0xb);
  data = (U32) ss_mode_par_in;
  MWRITE32(data,((pU32) INTERFACESTART)+0xc);
  data = (U32) 0x100;
  MWRITE32(data,(pU32) INTERFACESTART);
  return(0);
}

int S2()
{
  U32 data;
  /* State two for the software part of the UMTS slot synchronization*/
  data = (U32) 0x200;
  MWRITE32(data,(pU32) INTERFACESTART);
  if (s2_count >= 2)
  {
    data = (U32) 0x400;
    MWRITE32(data,(pU32) INTERFACESTART);
    return(1);
  }
  s2_count ++;
  return(0);
}

int S3()
{
  U32 data;
  /* State three for the software part of the UMTS slot synchronization*/
  data = (U32) 0x800;
  MWRITE32(data,(pU32) INTERFACESTART);
  return(0);
}

int S31()
{
  /* State three-one for the software part of the UMTS slot synchronization*/
  MREAD32(p_data2, ((pU32) INTERFACESTART) + 0xd);
  if (data2 == 0xf0f0)
  {
    U32 data;
```

```
    data = (U32) 0x1000;
    MWRITE32(data,(pU32) INTERFACESTART);
    return(1);
  }
  else
  {
    return(0);
  }
}

int S4()
{
  /* State four(calculation part)
  /*for the software part of the UMTS slot synchronization*/
  int ccnt[4];
  int tmp_diff;
  int cand_index_pos;
  int th_a[4];
  bool pos_occur = 0;
  U32 data;
  int  i;
  MREAD32(p_ccnt1_res, ((pU32) INTERFACESTART) + 0x1);
  MREAD32(p_ccnt2_res, ((pU32) INTERFACESTART) + 0x2);
  MREAD32(p_ccnt3_res, ((pU32) INTERFACESTART) + 0x3);
  MREAD32(p_ccnt4_res, ((pU32) INTERFACESTART) + 0x4);
  ccnt[0] = (int) ccnt1_res;
  ccnt[1] = (int) ccnt2_res;
  ccnt[2] = (int) ccnt3_res;
  ccnt[3] = (int) ccnt4_res;
  for(i = 0; i < num_th; i++)
  {
    int diff = num_required - (int) ccnt[i];
    if (diff >= 0)
    {
      if (pos_occur)
      {
        if (tmp_diff > diff)
        {
          tmp_diff = diff;
          cand_index_pos = i;
        }

      }
      else
      {
        tmp_diff = diff;
        cand_index_pos = i;
        pos_occur = 1;
      }
    }
  }
  th_a[0] = th1_par;
  th_a[1] = th2_par;
  th_a[2] = th3_par;
  th_a[3] = th4_par;
  if (pos_occur)
  {
    data = (U32) th_a[cand_index_pos];
    MWRITE32(data,((pU32) INTERFACESTART+0xe));
  }
  else
  {
    data = (U32) th_a[0];
    MWRITE32(data,((pU32) INTERFACESTART+0xe));
  }
  data = (U32) 0x2000;
```

```
  MWRITE32(data,(pU32) INTERFACESTART);
  return(0);
}

int S5()
{
  /* State five for the software part of the UMTS slot synchronization*/
  U32 data;
  data = (U32) 0x4000;
  MWRITE32(data,(pU32) INTERFACESTART);
  return(0);
}

int S6()
{
  /* State six for the software part of the UMTS slot synchronization*/
  U32 data;
  data = (U32) 0x8000;
  MWRITE32(data,(pU32) INTERFACESTART);
  return(0);
}

int ram8test()
{
  int total, size;
  U8 answer;
  pU8 i;
  pU8 pAnswer = &answer;
  U8 data;
  /* Test each memory address by writing and reading a unique value
  * from each 8 bit data location.
  */
  data = 1;
  size = 0;
  total = 0;
  for (i=(pU8) RAMSTART16; i < (pU8)(RAMSTART16 + CHK_ADDRESSES);  i++)
  {
    MWRITE8(data, i);
    MREAD8(pAnswer, i);
    MWRITE8((U8) 0x0, i);
    if (answer != data)
    {
      failure(RAM8TEST);
    }
    data++;
    size++;
    total++;
  }
  return(0);
}

int ram16test()
{
  int total, size;
  U16 answer;
  pU16 i;
  pU16 pAnswer = &answer;
  U16 data;
  /* Test each memory address by writing and reading a unique value
  * from each 8 bit data location.
  */
  data = 0xa5a5;
  size = 0;
  total = 0;
  for (i=(pU16) RAMSTART16; i < (pU16)(RAMSTART16 + CHK_ADDRESSES);  i++)
  {
```

```
    MWRITE16(data, i);
    MREAD16(pAnswer, i);
    MWRITE16((U16) 0x0, i);
    if (answer != data)
    {
      failure(RAM16TEST);
    }
    data++;
  }
  return(0);
}

int ram32test()
{
  int total, size;
  U32 answer;
  pU32 i;
  pU32 pAnswer = &answer;
  U32 data;
  /* Test each memory address by writing and reading a unique value
  * from each 8 bit data location.
  */
  data = 0xaabbccdd;
  size = 0;
  total = 0;
  for (i=(pU32) RAMSTART32; i < (pU32)(RAMSTART32 + CHK_ADDRESSES);  i++)
  {
    MWRITE32(data, i);
    MREAD32(pAnswer, i);
    MWRITE32((U32)0x0, i);
    if (answer != data)
    {
      failure(RAM32TEST);
    }
    data++;
  }
  return(0);
}

#if (NEED_FUNC_PROTOTYPES == 1)
  int main()
#else
  int main()
#endif
{
  int check;
  #if defined(BIGEND)
    /* Change the endian mode to big. */
    MWRITE32( CONFIG_BIG_END , CONFIG_BIGEND );
    MWRITE32( CONFIG_BIG_END , START_BIGEND );
  #endif

  #ifndef NOINTERRUPTS
    init_interrupts();
  #endif

  /*
  //  endiantest(LOCALADDR);
  //  ram8test();
  //  ram16test();
  //  ram32test();
  //  itcmtest();
  //  dtcmtest();
  //  fiqtest();
  //  irqtest();
  //  dataaborttest();
```

```
  //  passed();
  //  resettest(); */

  S1();
  do
  {
    check = S2();
  } while (check == 0);
  S3();
  do
  {
    check = S31();
  } while (check == 0);
  S4();
  S5();
  S6();
  return(0);
}
```

The function `endiantest` is responsible for testing if the memory and the peripherals are in the little endian mode. The functions `ram8test()`, `ram16test` and `ram32test` test the memory for 8-bit, 16-bit and 32-bit read and writes. For testing the instruction and data TCM memory, the functions `itcmtest` and `dtcmtest` are written. The interrupt functions `figtest()` and `irqtest()` are defined to test the interrupt funtions of the ARM926 EJ-S processor. The function `dataaborttest()` is responsible for detecting an access to an undefined memory area. For testing the reset functions of the processor the `resettest()` function is written. The procedure `passed()` is responsible for detecting any errors from one of the functions described above. The functions `S1()`, `S2()`, `S3()`, `S31()`, `S4()`, `S5()` and `S6()` represent the state diagram of the UMTS slot synchronization software. The state function `S1()` passes variables to software interface from which it is transmitted to the UMTS slot synchronization hardware. The state functions `S2()`, `S3()`, `S5()` and `S6()` are responsible for performing the handshake between UMTS slot synchronization hardware and UMTS slot synchronization software. The state function `S4()` represents the *ThresMaskCalc* block.

C.2 C - header code file "*examples_ss.h*" for the UMTS slot synchronization software part running on ARM926 EJ-S

Definition files for the memory locations, data values and variables for the UMTS slot synchronization software were also written. The header file "*examples_ss.h*" for the "*examples_ss.c*" source code file looks like:

```
//examples_ss.h
```

```
/*  defines a64 bit word in LSB and MSB data words */
/*
* Handle function declarations for C, ANSI C, and C++
*/
#ifndef _EI_FUNC_PROTO_H_
  #define _EI_FUNC_PROTO_H_
  #ifndef NEED_FUNC_PROTOTYPES
    #if defined(FUNCPROTO) || defined(__STDC__) || \
        defined(__cplusplus) || defined(c_plusplus) || \
        defined(_MSC_VER)
      #define NEED_FUNC_PROTOTYPES 1
    #else  /* FUNCPROTO */
      #define NEED_FUNC_PROTOTYPES 0
    #endif /* FUNCTPROTO */
  #endif /* NEED_FUNC_PROTOTYPES */
  #ifndef FUNC_PROTOTYPE_BEG
    #ifdef __cplusplus
      #define FUNC_PROTOTYPE_BEG extern "C"
      {
        # define FUNC_PROTOTYPE_END
      }
      #else /* __cplusplus */
        #define FUNC_PROTOTYPE_BEG
        #define FUNC_PROTOTYPE_END
      #endif /*__cplusplus */
    #endif /* FUNC_RPTOTYPE_BEG */
#endif  /* _EI_FUNC_PROTO_H */

typedef char  bool;
typedef char  CHAR, *pCHAR;
typedef unsigned char U8,*pU8;
typedef unsigned short U16,*pU16;
typedef long int S32,*pS32;
typedef unsigned long int U32,*pU32;
typedef struct _vspU64
{
  U32 vspHigh;
  U32 vspLow;
} U64, *pU64;
typedef U32 VSPADR,*pVSPADR;
typedef U64 VSPSIMTIME,*pVSPSIMTIME;
/*
* extern defns
* /
extern int ram8test(void);
extern int ram16test(void);
extern int ram32test(void);
extern int S1(void);
extern int S2(void);
extern int S3(void);
extern int S31(void);
extern int S4(void);
extern int S5(void);
extern int S6(void);

#define MWRITE8(data, addr)*((pU8) (addr)) = (data)
#define MREAD8(data, addr)*(data) = *((pU8) (addr))
#define MWRITE16(data, addr)*((pU16) (addr)) = (data)
#define MREAD16(data, addr)*(data) = *((pU16) (addr))
#define MWRITE32(data, addr)*((pU32) (addr)) = (data)
#define MREAD32(data, addr)*(data) = *((pU32) (addr))
#define MWRITE64(data, addr)*((pU64) (addr)) = (data)
#define MREAD64(data, addr)*(data) = *((pU64) (addr))

struct int64
{
```

```
  int lw0;
  int lw1;
};
/*  Memory Mapping */
#define ROMSTART        0x90000000
#define ROMEND          0x900003ff
#define RAMSTART16      0x90000000
#define RAMEND16        0x9000000f
#define RAMSTART32      0x90000000
#define RAMEND32        0x9000000f
#define ITCMSTART       0xFF000
#define ITCMEND         0xFF00f
#define DTCMSTART       0x20000
#define DTCMEND         0x2000f
#define CHK_ADDRESSES   0x10
/*Startadress for the interface to the UMTS slot synchronization hardware*/
#define INTERFACESTART  0xb0000000
/*Endadress for the interface to the UMTS slot synchronization hardware*/
#define INTERFACEEND    0xb00003ff
#define SemiSWI         0x123456
/*address to try read/write to cause data abort*/
#define DABORTADDR      0x0d000000
/*Data abort vector*/
#define DABORTVEC       0x10
/*RESET vector*/
#define RESETVEC        0x08
/*IRQ vector*/
#define IRQVEC          0x18
/*FIQ vector*/
#define FIQVEC          0x1c
/*Design address of control reg */
#define CNTLREG         0xf0000000
/* Write a character */
__swi(SemiSWI) void _WriteC(unsigned op, const char *c);
#define PrintChar(c) _WriteC (0x3,c)

/* Write a string */
__swi(SemiSWI)void _Write0(unsigned op, const char *string);
#define PrintStr(string) _Write0 (0x4,string)

extern int term_write_str(char *);
```

The `INTERFACESTART` and the `INTERFACEEND` defines the start address and the end address of the interface between the UMTS slot synchronization software (calculation running on ARM926 EJ-S) and the UMTS slot synchronization hardware.

C.3 C - header code file "*definition.h*" for the UMTS slot synchronization software part running on ARM926 EJ-S

In the source code of header file "*definition.h*" are variables for the UMTS slot synchronization hardware defined which reads as follows:

```
/*defintion.h file is used for declaration of parameters which has
to passed to UMTS slot synchronization hardware */
```

```
#define th0_par          100
#define st_par           10
#define th1_par          1
#define th2_par          2
#define th3_par          4
#define th4_par          8
#define a_par            10000
#define b_par            32
#define c_par            1
#define nca_par          7 //num_corr_acc-1
#define nsa_par          19//num_slot_acc-1
#define ss_mode_par_in   2
#define num_th           4//size of the th(1,2,3,4)_par array
#define num_required     31//=num_req_psel_peaks=temp_ram_size-1
```

Appendix D

TLM model of the bus interface between the UMTS slot synchronisation hardware in SystemC 1.0 and the abstract bus model, the AMBA bus, in SystemC 2.0

D.1 Source code of the TLM bus interface

Following source code implements the read and write procedures from and to the abstract bus. Memory access procedures and different transaction procedures were also implemented to read and write data values to and from the UMTS slot synchronization hardware written in SystemC 1.0. The source code reads as follows:

```
// SW_interface.h
#ifndef __SW_interface_h
#define __SW_interface_h

#include <systemc.h>
#include <ccss_systemc.h>
#include "ahb_slave_if.h"

#ifdef CCSS_USE_SC_CTOR
  #define CCSS_INIT_MEMBERS_PREFIX :
#undef CCSS_USE_SC_CTOR
#else
  #define CCSS_INIT_MEMBERS_PREFIX ,
#endif
```

```
#define CCSS_INIT_MEMBERS  CCSS_INIT_MEMBERS_PREFIX \
  nFIQ("nFIQ") \
  , nIRQ("nIRQ") \
  , nRESET("nRESET") \
  , VINITHI("VINITHI") \
  , clk("clk") \
  , INITRAM("INITRAM") \
  , cs_reset("cs_reset") \
  , cs_continue("cs_continue") \
  , load_th0_pr("load_th0_pr") \
  , load_st_pr("load_st_pr") \
  , load_th_pr("load_th_pr") \
  , load_a_pr("load_a_pr") \
  , load_b_pr("load_b_pr") \
  , load_c_pr("load_c_pr") \
  , load_tm_pr("load_tm_pr") \
  , load_nca_pr("load_nca_pr") \
  , load_nsa_pr("load_nsa_pr") \
  , load_ssm_pr("load_ssm_pr") \
  , th0_par_in("th0_par_in") \
  , st_par_in("st_par_in") \
  , th1_par_in("th1_par_in") \
  , th2_par_in("th2_par_in") \
  , th3_par_in("th3_par_in") \
  , th4_par_in("th4_par_in") \
  , a_par_in("a_par_in") \
  , b_par_in("b_par_in") \
  , c_par_in("c_par_in") \
  , nca_par_in("nca_par_in") \
  , nsa_par_in("nsa_par_in") \
  , ss_mode_par_in("ss_mode_par_in") \
  , out_th_mask("out_th_mask") \
  , cs_ready("cs_ready") \
  , ccnt1_res_in("ccnt1_res_in") \
  , ccnt2_res_in("ccnt2_res_in") \
  , ccnt3_res_in("ccnt3_res_in") \
  , ccnt4_res_in("ccnt4_res_in")

// This memory is a slave module that implements an interrupt/signal generator.
// Interrupts/signals supported are nFIQ, nIRQ, nRESET and VINITHI.
// It was modeled after a memory model, so behaves similarly.
// An arbitrary number of wait states can be specified for read and write
// data transactions by the parameter ReadWaitStates and WriteWaitStates
// with a default value of zero. This module implements the slave interface
// and the debug interface.
//
// The constructor has the following arguments:
//
//   sc_module_name name_                  // name of the module
//   abm_addr_t start_address              // start address of the memory
//   abm_addr_t end_address                // end address of the memory
//   const sc_string& file_name     = ""   // file name to read initial values
//   bool BigEndian              = false // little endian memory addressing
// The start and end address define a memory segment. The start_address,
// the end_address+1 and thus, the memory size have to be alligned to the
// 1k address boundary. The start_address has also be less than the
// end_address.
//
// If the monitor feature is used, the trace_start_address and trace_end_address
// specify a subsegment of the memory where the contents can be monitored. If
// monitoring is enabled by the DoTrace parameter the simulation may slow down.
//
// Bit assignments at address 0xb0000000:
//  0    IRQ      --   write to 0xb0000000 with data 0x1 causes nIRQ
//  1    FIQ      --   write to 0xb0000000 with data 0x2 causes nFIQ
//  2    RESET    --   write to 0xb0000000 with data 0x4 causes nRESET
```

```
//  3    VINITHI  --   write to 0xb0000000 with data 0x8 causes VINITHI
//  4    INITRAM  --   write to 0xb0000000 with data 0x10 causes INITRAM
class SW_interface// Interrupt Generator
  : public sc_module
  , public ahb_slave_if
{

public:
  // parameters
  // This parameter specifies the number of data cycle wait
  // states to complete a data transfer.
  CCSS_PARAMETER(int, ReadWaitStates);
  // This parameter specifies the number of data cycle wait
  // states to complete a data transfer.
  CCSS_PARAMETER(int, WriteWaitStates);
  // Parameter to enable the read only mode (Read Only Memory)
  CCSS_PARAMETER(bool, ReadOnly);
  // This flag enables monitoring features if compiled with debug option.
  CCSS_PARAMETER(bool, DoTrace);
  CCSS_PARAMETER(int, SlaveID);
  // ports
  // Active low output pin for fast interrupt signal to processor
  sc_out<bool> nFIQ;
  // Active low output pin for interrupt request signal to processor
  sc_out<bool> nIRQ;
  // Active low output pin for reset signal to processor
  sc_out<bool> nRESET;
  // Active high output pin for initializing processor to high vectors.
  sc_out<bool> VINITHI;
  // Clock input
  sc_in_clk clk;
  // Active high output for turning TCM functionality on in processor.
  sc_out<bool> INITRAM;
  //Signals for the cell searcher
  sc_out<bool> cs_reset;
  sc_out<bool> cs_continue;
  sc_out<bool> load_th0_pr;
  sc_out<bool> load_st_pr;
  sc_out<bool> load_th_pr;
  sc_out<bool> load_a_pr;
  sc_out<bool> load_b_pr;
  sc_out<bool> load_c_pr;
  sc_out<bool> load_tm_pr;
  sc_out<bool> load_nca_pr;
  sc_out<bool> load_nsa_pr;
  sc_out<bool> load_ssm_pr;
  sc_out<sc_uint<16> > th0_par_in;
  sc_out<sc_uint<16> > st_par_in;
  sc_out<sc_uint<4> > th1_par_in;
  sc_out<sc_uint<4> > th2_par_in;
  sc_out<sc_uint<4> > th3_par_in;
  sc_out<sc_uint<4> > th4_par_in;
  sc_out<sc_uint<32> > a_par_in;
  sc_out<sc_uint<13> > b_par_in;
  sc_out<sc_uint<3> > c_par_in;
  sc_out<sc_uint<5> > nca_par_in;
  sc_out<sc_uint<6> > nsa_par_in;
  sc_out<sc_uint<2> > ss_mode_par_in;
  sc_out<sc_uint<4> > out_th_mask;
  sc_in<bool> cs_ready;
  sc_in<int> ccnt1_res_in;
  sc_in<int> ccnt2_res_in;
  sc_in<int> ccnt3_res_in;
  sc_in<int> ccnt4_res_in;
  // initialize parameters
  virtual void InitParameters()
```

```
{
  int _tmp_ReadWaitStates = 0;
  ReadWaitStates.conditional_init(_tmp_ReadWaitStates);
  int _tmp_WriteWaitStates = 0;
  WriteWaitStates.conditional_init(_tmp_WriteWaitStates);
  bool _tmp_ReadOnly = false;
  ReadOnly.conditional_init(_tmp_ReadOnly);
  bool _tmp_DoTrace = false;
  DoTrace.conditional_init(_tmp_DoTrace);
  int _tmp_SlaveID = 0;
  SlaveID.conditional_init(_tmp_SlaveID);
}
SC_HAS_PROCESS(SW_interface);

// constructor
SW_interface(sc_module_name name_,            // name of the module
  ahb_addr_t start_address,        // start address of the memory
  ahb_addr_t end_address,          // end address of the memory
  const sc_string& file_name = "", // file name to read initial values
  bool  big_endian = false);        // little endian memory encoding

~SW_interface();
// interface methods
// Register a port, make sure that only one port
// is connected to the interface.
void
register_port(sc_port_base&,
  const char*);

int
register_bus(int,int,int,int);
// Query the status response of the transaction.

bool
response(int, ahb_status&);
// Initiate a read transaction from the slave.

void
read(int,                   // bus handle, unused here
  ahb_addr_t,           // address
  int);                 // number of bytes

// Write data to the slave.
void
write(int,                  // bus handle, unused here
  ahb_addr_t,          // address
  int);                // number of bytes

// This method provides additional control information, e.g.
// the burst mode and the transfer type.
virtual void
control_info(int,                // id of the bus
  ahb_burstmode,      // mode of the active burst
  ahb_trans,          // the transfer type
  ahb_prot,           // access protection information
  int,                // master id
bool);              // master locks the bus

// Set the data pointer where the data shoud be
// - written to for a read transaction
// - read from for a write transaction
void
set_data(int,              // bus handle, unused here
//       ahb_data_t*);      pointer to the data
ahb_data_t);
```

```
  // Read the memory from a file
  void
  read_from_file(const sc_string&);

  // Write the memory contents to a file
  void
  write_to_file(const sc_string&);

  // Query the address range of the memory.
  ahb_addr_t
  start_address() const;

  ahb_addr_t
  end_address() const;

  // Direct read from the memory for debugging purposes
  bool
  direct_read(int,
    ahb_addr_t,           // address
    ahb_data_t,          // data
    int = 4 );            // number of bytes

  // Direct write to the memory for debugging purposes
  bool
  direct_write(int,
    ahb_addr_t,          // address
    ahb_data_t,         // data
  int = 4);            // number of bytes

  // Return the memory map of the slave
  //virtual const vector<ahb_address_region>&
  //get_memory_map(int, int& id_) {
  //  id_ = SlaveID;
  //  return _memory_map;
  //}

  // Return the memory map of the slave
  virtual const vector<ahb_address_region>&
  get_address_map(int) {
    return _address_map;
  }

  // Query the name of the slave
  const char*
  name() const
  {
    return sc_module::name();
  }


protected:

  bool
  get_physical_address(ahb_addr_t,
  ahb_addr_t &);

  vector<ahb_address_region> _address_map; // vector of up to 8 memory regions
  unsigned int        _region_offset[8];
  // start addresses of each memory region in
  // the real linear memory _mem

  // parameter change handler
  void
  do_trace_action();

  void
```

```
create_memory();

void
clear_mem();

bool
get_physical_address(unsigned int &);

bool
get_mem(unsigned int, unsigned int*, int, int);

bool
put_mem(unsigned int, unsigned int*, int, int);

// main process for interrupt sampling.
void
main_action();

// Clear the memory contrents to zero
void
_clear_mem();

// Update the table monitor for the memory contents
void
set_monitor();
void
set_monitor(int);

ahb_addr_t          _start_address;
ahb_addr_t          _end_address;
sc_string           _file_name;

bool                _big_endian;

//ahb_data_t*         _mem;
unsigned int *      _mem;
unsigned int        _mem_length;

unsigned int        _address;
int                 _address_width_bytes;
bool                _rflag;
int                 _number_of_bytes;

ahb_data_t          _data;
ahb_data_t          _bus_data;
int                 _data_width_bytes;

int                 _waitstates;
bool                _pending;
ahb_status          _s;

int                 mask2[4];
int                 mask4[4];

//const ahb_data_t*   mask_8b;
//const ahb_data_t*   mask_16b;

const unsigned int* mask_8b;
const unsigned int* mask_16b;

ccss_monitor_table* _mem_monitor_table;

sc_port_base*       _slave_port;

bool _okay;
bool _idlebusy;
```

```
      bool _ready;

      int reset_cnt;
      int irq_pulse;
      int fiq_pulse;
      bool once;

    };
    // end module SW_interface
    #undef CCSS_INIT_MEMBERS_PREFIX
    #undef CCSS_INIT_MEMBERS

    #endif


    //SW_interface.cpp
    #include "SW_interface.h"
    static const unsigned int big_endian_mask_8b[4] = {
      0xff000000, 0x00ff0000, 0x0000ff00, 0x000000ff
    };

    static const unsigned int big_endian_mask_16b[4] = {
      0xffff0000, 0xdeadbeef, 0x0000ffff, 0xdeadbeef
    };

    static const unsigned int little_endian_mask_8b[4] = {
      0x000000ff, 0x0000ff00, 0x00ff0000, 0xff000000
    };

    static const unsigned int little_endian_mask_16b[4] = {
      0x0000ffff, 0xdeadbeef, 0xffff0000, 0xdeadbeef
    };

    #include <typeinfo>
    #include <string.h>
    #include <stdio.h>

    SW_interface::SW_interface(sc_module_name   name_,
      ahb_addr_t       start_address,
      ahb_addr_t       end_address,
      const sc_string& file_name,
      bool             big_endian)
      : sc_module            (name_)
      , _start_address       (start_address)
      , _end_address         (end_address)
      , _file_name           (file_name)
      , _big_endian          (big_endian)
      , _mem_monitor_table   (0)
      , _slave_port          (0)
    {
      // check the address range
      if(start_address >= end_address)
      {
        cerr << " ERROR: start address must be greater than the end address. <"
<< name() << ">" << endl;
        sc_stop();
      }

      // check the parameter address 1k alignment
      if(start_address&0x3FF)
      {
        cerr << " ERROR: start address must be 1k aligned. <" << name() << ">"
<< endl;
        sc_stop();
      }

      if((end_address+1)&0x3FF)
```

```
      {
        cerr << " ERROR: end address must be 1k aligned. <" << name() << ">"
<< endl;
        sc_stop();
      }

      ahb_address_region tmp(start_address, end_address);
      _address_map.push_back(tmp);
      _region_offset[0] = 0;
      _mem_length = end_address-start_address+1;
      _region_offset[0] = 0;
      _waitstates = 0;

      create_memory();
      SC_METHOD( main_action );
        sensitive_pos << clk;
        once = true;
    }


    void SW_interface::main_action()
    {
      if (once)
        cs_reset.write(true);

      if(_mem[0] & 0x00000001)
      {
        // cout << "state: " << _mem[0] << "\n";
        if (irq_pulse)
        {
          irq_pulse--;
          if (!irq_pulse)
          {
            _mem[0] &= _mem[0] & 0x000fff0e;
          }
        }
        nIRQ.write(0);
      }
      else
      {
        nIRQ.write(1);
      }

      if(_mem[0] & 0x00000002)
      {
        // cout << "state: " << _mem[0] << "\n";
        if (fiq_pulse)
        {
          fiq_pulse--;
          if (!fiq_pulse)
          {
            _mem[0] &= _mem[0] & 0xfff0000d;
          }
        }
        nFIQ.write(0);
      }
      else
      {
        nFIQ.write(1);
      }

      if(_mem[0] & 0x00000004)
      {
        // cout << "state: " << _mem[0] << "\n";
        if (reset_cnt <= 1)
        {
```

```
    nRESET.write(1);
    _mem[0] &= 0xfff0000b;
  }
  else
  {
    nRESET.write(0);
    reset_cnt--;
  }
}
else
{
  nRESET.write(1);
  reset_cnt = 30;
}

if(_mem[0] & 0x00000008)
{
  // cout << "state: " << _mem[0] << "\n";
  VINITHI.write(1);
}
else
{
  VINITHI.write(0);
}

if(_mem[0] & 0x00000010)
{
  // cout << "state: " << _mem[0] << "\n";
  INITRAM.write(1);
}
else
{
  INITRAM.write(0);
}
// begin of state machine for the UMTS slot synchronization hardware
if(_mem[0] & 0x00000100)
{
  // cout << "state: " << _mem[0] << "\n";
  cs_reset.write(true);
  load_th0_pr.write(true);
  load_st_pr.write(true);
  load_th_pr.write(true);
  load_a_pr.write(true);
  load_b_pr.write(true);
  load_c_pr.write(true);
  load_nca_pr.write(true);
  load_nsa_pr.write(true);
  load_ssm_pr.write(true);
  load_tm_pr.write(false);
  th0_par_in.write((sc_uint<16>) _mem[1]);
  // cout << "_mem[1] th0_par_in: " << (sc_uint<16>)_mem[1] << "\n";
  st_par_in.write((sc_uint<16>) _mem[2]);
  // cout << "_mem[2] st_par_in: " << (sc_uint<16>)_mem[2] << "\n";
  th1_par_in.write((sc_uint<4>) _mem[3]);
  // cout << "_mem[3] th1_par_in: " << (sc_uint<4>)_mem[3] << "\n";
  th2_par_in.write((sc_uint<4>) _mem[4]);
  // cout << "_mem[4] th2_par_in: " << (sc_uint<4>)_mem[4] << "\n";
  th3_par_in.write((sc_uint<4>) _mem[5]);
  // cout << "_mem[5] th3_par_in: " << (sc_uint<4>)_mem[5] << "\n";
  th4_par_in.write((sc_uint<4>) _mem[6]);
  // cout << "_mem[6] th4_par_in: " << (sc_uint<4>)_mem[6] << "\n";
  a_par_in.write((sc_uint<32>) _mem[7]);
  // cout << "_mem[7] a_par_in: " << (sc_uint<32>)_mem[7] << "\n";
  b_par_in.write((sc_uint<13>) _mem[8]);
  // cout << "_mem[8] b_par_in: " << (sc_uint<13>)_mem[8] << "\n";
  c_par_in.write((sc_uint<3>) _mem[9]);
```

```
  // cout << "_mem[9] c_par_in: " << (sc_uint<3>)_mem[9] << "\n";
  nca_par_in.write((sc_uint<5>) _mem[10]);
  // cout << "_mem[10] nca_par_in: " << (sc_uint<5>)_mem[10] << "\n";
  nsa_par_in.write((sc_uint<6>) _mem[11]);
  // cout << "_mem[11] nsa_par_in: " << (sc_uint<6>)_mem[11] << "\n";
  ss_mode_par_in.write((sc_uint<2>) _mem[12]);
  // cout << "_mem[12] ss_mode_par_in: " << (sc_uint<2>)_mem[12] << "\n";
  cs_continue.write(false);
  once = false;
}

if(_mem[0] & 0x00000200)
{
  // cout << "state: " << _mem[0] << "\n";
  cs_reset.write(false);
  load_th0_pr.write(false);
  load_st_pr.write(false);
  load_th_pr.write(false);
  load_a_pr.write(false);
  load_b_pr.write(false);
  load_c_pr.write(false);
  load_nca_pr.write(false);
  load_nsa_pr.write(false);
  load_ssm_pr.write(false);
  cs_continue.write(false);
}

if(_mem[0] & 0x00000400)
{
  // cout << "state: " << _mem[0] << "\n";
  cs_continue.write(true);
}

if(_mem[0] & 0x00000800)
{
  // cout << "state: " << _mem[0] << "\n";
  cs_continue.write(false);
  if (cs_ready.read() == true)
  {
    // cout << "mem[13]: " << _mem[13] << "\n";
    _mem[13] = 0xf0f0;
  }
  else
  {
    _mem[13] = 0x0000;
  }
  // cout << "mem[13]: " << _mem[13] << "\n";
}

if(_mem[0] & 0x00001000)
{
  // cout << "state: " << _mem[0] << "\n";
  _mem[1] = (int) ccnt1_res_in;
  // cout << "mem[1]:" << _mem[1] << "\n";
  _mem[2] = (int) ccnt2_res_in;
  // cout << "mem[2]:" << _mem[2] << "\n";
  _mem[3] = (int) ccnt3_res_in;
  // cout << "mem[3]:" << _mem[3] << "\n";
  _mem[4] = (int) ccnt4_res_in;
  // cout << "mem[4]:" << _mem[4] << "\n";
}

if(_mem[0] & 0x00002000)
{
  // cout << "state: " << _mem[0] << "\n";
  out_th_mask.write((sc_uint<4>) _mem[14]);
```

```
        load_tm_pr.write(true);
      }

      if(_mem[0] & 0x00004000)
      {
        // cout << "state: " << _mem[0] << "\n";
        cs_continue.write(true);
        load_tm_pr.write(false);
      }

      if(_mem[0] & 0x00008000)
      {
        // cout << "state: " << _mem[0] << "\n";
        cs_continue.write(false);
      }
      // end of state machine for the UMTS slot synchronization hardware
    }

    void SW_interface::create_memory()
    {
      // we might want to check for memory overlaps here ...
      _mem = new unsigned int [_mem_length/4];
      if(_mem == 0)
      {
        cerr << name() << " ERROR: unable to allocate " << _mem_length
<< " bytes." << endl;
        sc_stop();
      }
      // reset the ahb_memory contents to '0'
      clear_mem();
      if(_file_name != "")
      {
        // read the initial ahb_memory from file
        read_from_file(_file_name);
      }
      if (_big_endian)
      {
        mask_16b = big_endian_mask_16b;
        mask_8b = big_endian_mask_8b;
      }
      else
      {
        mask_16b = little_endian_mask_16b;
        mask_8b = little_endian_mask_8b;
      }

      SC_METHOD(do_trace_action);

        sc_string colstring("");
        for(int i=0; i<16; i++)
        {
          colstring += sc_string::to_string("%1X\t", i);
        }

        sc_string rowstring("");
        int num_rows = 0;
        for(unsigned int j=0; j<_address_map.size(); j++)
        {
          for(unsigned int i=_address_map[j].boot_start_addr;
  i<=_address_map[j].boot_end_addr; i+=16)
          {
            rowstring += sc_string::to_string("%08X\n", i);
            num_rows++;
          }
        }
        _mem_monitor_table = new ccss_monitor_table("memory_table",
```

```
        "Memory Contents",
        num_rows,
        16,
        rowstring.c_str(),
        colstring.c_str());
      if(_mem_monitor_table == 0)
      {
        cerr << " ERROR: unable to allocate monitor object. <" << name()
<< ">" << endl;
        sc_stop();
      }
  }

  // ----------------------------------------------------------------------------
  //  Destructor : ahb_memory::~ahb_memory
  // ----------------------------------------------------------------------------

  SW_interface::~SW_interface()
  {
    // free the allocated memory
    if(_mem != 0)
    {
      delete[] _mem;
      _mem = 0;
    }
    if(_mem_monitor_table != 0)
    {
      delete _mem_monitor_table;
      _mem_monitor_table = 0;
    }
  }

  //void
  //DW_IntrGen::on_reset_event(void)
  //{
  //  _waitstates = 0;
  //  clear_mem();
  //}

  // ----------------------------------------------------------------------------
  //  Process: memory::do_trace_action
  //
  //  handler for the DoTrace parameter value change
  // ----------------------------------------------------------------------------
  void SW_interface::do_trace_action()
  {
    // display the full monitor contents if changed to true
    if(DoTrace)
      set_monitor();

    next_trigger(DoTrace.value_changed_event());
  }

  // ----------------------------------------------------------------------------
  //  Interface Method: ahb_memory::register_port
  // ----------------------------------------------------------------------------
  void SW_interface::register_port(sc_port_base& port_,
    const         char*)
  {
    if (typeid(sc_port<ahb_slave_if>) == typeid(port_))
    {
      if(_slave_port != 0)
      {
        cout << "ERROR: AHB slave interface was already bound. <" << name()
<< ">" << endl;
        sc_stop();
```

```
    }
    _slave_port = &port_;
  }
}

int SW_interface::register_bus(int id,
  int priority,
  int address_width,
  int data_width)
{
  _address_width_bytes = address_width/8;
  _data_width_bytes    = data_width/8;
  _data = new unsigned int[data_width/32];
  return SlaveID;
}

// ----------------------------------------------------------------------------
//  Interface Method: ahb_memory::status
// ----------------------------------------------------------------------------
bool SW_interface::response(int handle_, ahb_status& status_)
{
  if(_idlebusy)
  {
    _idlebusy = false;
    status_ = AHB_OKAY;
    return true;
  }
  // check if the set_data method reported an error
  if(_okay)
  {
    status_ = AHB_OKAY;
    _waitstates--;
    // no error, count the number of wait cycles.
    if(_waitstates > 0)
    {
      _ready = false;
    }
    else
    {
      _ready = true;
      if(_bus_data)
        *_bus_data = *_data;
    }
    return _ready;
  }
  else
  {
    status_ = AHB_ERROR;
    _ready = true;
  }
  return _okay;
}
// ----------------------------------------------------------------------------
//  Interface Method: ahb_memory::read
// ----------------------------------------------------------------------------
void SW_interface::read(int        handle_,
  ahb_addr_t address,
  int        number_of_bytes)
{
  _address         = address;
  _number_of_bytes = number_of_bytes;
  _rflag           = true;
  if(_waitstates == 0)
  _waitstates = ReadWaitStates+1;
}
```

```
// ------------------------------------------------------------------------
//  Interface Method : ahb_memory::write
// ------------------------------------------------------------------------
void SW_interface::write(int handle_,
  ahb_addr_t address,
  int number_of_bytes)
{
  _address         = address;
  _number_of_bytes = number_of_bytes;
  _rflag = false;
  if(_waitstates == 0)
    _waitstates = WriteWaitStates+1;
}

void SW_interface::control_info(int,
  ahb_burstmode,
  ahb_trans transf_type_,
  ahb_prot,
  int,
  bool)
{
  if(_ready)
  {
    // explicit busy or idle command, response must be AHB_OKAY
    switch(transf_type_)
    {
        case AHB_BUSY:
          _idlebusy = true;
          _waitstates--;
          break;
        case AHB_IDLE:
          _idlebusy = true;
          _waitstates = 0;
          break;
        default:
          break;
    }
  }
}

// ------------------------------------------------------------------------
//  Interface Method : ahb_memory::set_data
// ------------------------------------------------------------------------
void SW_interface::set_data(int handle_,
  ahb_data_t data_)
{
  // error if writing to ReadOnly ahb_memory
  if (ReadOnly && !_rflag)
  {
    _okay = false;
    return;
  }
  _okay = get_physical_address (_address);
  if(!_okay)
  {
    return;
  }
  if (_rflag)
  {
    // temporary set the value to a dummy
    //_bus_data = data_;
    //data_[0] = 0x0;
    _bus_data = 0;
    _okay =  get_mem(_address, data_, _data_width_bytes, _number_of_bytes);
    //_okay =  get_mem(_address, _data, _data_width_bytes, _number_of_bytes);
  }
```

```
    else
    {
      _bus_data = 0;
      _okay =  put_mem(_address, data_, _data_width_bytes, _number_of_bytes);
    }
    return;
  }

  // ------------------------------------------------------------------------
  //  Interface Method : ahb_memory::direct_read
  // ------------------------------------------------------------------------
  bool SW_interface::direct_read(int        handle_,
    ahb_addr_t address,
    ahb_data_t data_,
    int        number_of_bytes_)
  {
    unsigned int tmp_addr = address;
    get_physical_address (tmp_addr);
    assert ((tmp_addr>=0) && (tmp_addr<_mem_length));
    return get_mem(tmp_addr, data_, _data_width_bytes, number_of_bytes_);
  }

  // ------------------------------------------------------------------------
  //  Interface Method : ahb_memory::direct_write
  // ------------------------------------------------------------------------

  bool SW_interface::direct_write(int        handle_,
    ahb_addr_t address,
    ahb_data_t data_,
    int        number_of_bytes_)
  {
    unsigned int tmp_addr = address;
    // error if writing to ReadOnly ahb_memory
    if (ReadOnly)
      return false;
    get_physical_address (tmp_addr);
    assert ((tmp_addr>=0) && (tmp_addr<_mem_length));
    return put_mem(tmp_addr, data_, _data_width_bytes, number_of_bytes_);
  }

  // ------------------------------------------------------------------------
  //  Local Method: ahb_memory::get_physical_address
  //  translates the address in the memory regions into an address in
  //  physical memory _mem (all values character based)
  // ------------------------------------------------------------------------
  bool SW_interface::get_physical_address(unsigned int& abs_address)
  {
    unsigned int i = 0;
    do
    {
      if ((abs_address>=_address_map[i].boot_start_addr) &&
        (abs_address<=_address_map[i].boot_end_addr))
      {
        abs_address = abs_address-_address_map[i].boot_start_addr+
_region_offset[i];
        return true; //found = true;
      }
      i++;
    } while (i < _address_map.size());
    return false;
  }

  bool SW_interface::get_mem(unsigned int  idx,
    unsigned int* data_,
    int           transf_size,
    int           number_of_bytes)
```

```
  {
    if(transf_size > 4)
    {
      int offset = idx&(transf_size-1);
      transf_size /= 2;
      if(number_of_bytes > transf_size)
      {
        if(!get_mem(idx, data_, transf_size, number_of_bytes/2)) return false;
        if(!get_mem(idx+transf_size, data_+transf_size/4,
transf_size, number_of_bytes/2)) return false;
      }
      else
      {
        if(offset < transf_size)
        {
          if(!get_mem(idx, data_, transf_size, number_of_bytes)) return false;
        }
        else
        {
          if(!get_mem(idx, data_+transf_size/4, transf_size, number_of_bytes))
  return false;
        }
      }
    }
    else
    {
      unsigned int index  = idx/4;
      unsigned int offset = idx%4;
      sc_assert(index < _mem_length);
      // invalid start address
      if ( ((number_of_bytes==4)&&(offset))
        || ((number_of_bytes==2)&&(offset%2)) )
      {
        return false;
      }
      if(number_of_bytes == 4)
      {
        *data_ = _mem[index];
      }
      else
        if((number_of_bytes == 2))
        {
          *data_ &= ~mask_16b[offset];
          *data_ |= _mem[index]&mask_16b[offset];
        }
        else
        { //
          *data_ &= ~mask_8b[offset];
          *data_ |= _mem[index]&mask_8b[offset];
        }
    }
    return true;
  }

  bool SW_interface::put_mem(unsigned int  idx,
    unsigned int* data_,
    int           transf_size,
    int           number_of_bytes)
  {
    if(transf_size > 4)
    {
      int offset = idx&(transf_size-1);
      transf_size /= 2;
      if(number_of_bytes > transf_size)
      {
        if(!put_mem(idx, data_, transf_size, number_of_bytes/2)) return false;
```

```
        if(!put_mem(idx+transf_size, data_+transf_size/4,
transf_size, number_of_bytes/2)) return false;
      }
      else
      {
        if(offset < transf_size)
        {
          if(!put_mem(idx, data_, transf_size, number_of_bytes)) return false;
        }
        else
        {
          if(!put_mem(idx, data_+transf_size/4, transf_size, number_of_bytes))
  return false;
        }
      }
    }
    else
    {
      unsigned int index  = idx/4;
      unsigned int offset = idx%4;
      sc_assert(index < _mem_length);
      // invalid start address
      if ( ((number_of_bytes==4)&&(offset))
        || ((number_of_bytes==2)&&(offset%2)) )
      {
        return false;
      }
      if(number_of_bytes == 4)
      {
        _mem[index] = *data_;
      }
      else
        if ((number_of_bytes == 2))
        {
          _mem[index] &= ~mask_16b[offset];
          _mem[index] |= *data_&mask_16b[offset];
        }
        else
        {
          // number_of_bytes == 1
          _mem[index] &= ~mask_8b[offset];
          _mem[index] |= *data_&mask_8b[offset];
        }
      if(DoTrace)
        set_monitor(idx);
    }
    return true;
  }

  // ------------------------------------------------------------------------
  //  Local Method : ahb_memory::clear_mem
  // ------------------------------------------------------------------------
  void SW_interface::clear_mem()
  {
    memset(_mem, 0, _mem_length);
  }

  // ------------------------------------------------------------------------
  //  Interface Method : ahb_memory::read_from_file
  // ------------------------------------------------------------------------
  void SW_interface::read_from_file(const sc_string& fn)
  {
    FILE *stream;
    int status;
    unsigned int i = 0;
    unsigned int buffer;
```

```
  stream = fopen(expand(fn), "r");
  if(stream == 0)
  {
    cout << "ERROR: the data file: " << fn;
    cout << " can not be opened for reading. <" << name() << ">" << endl;
    sc_stop();
  }
  if (_big_endian)
  {
    while ((status=fscanf(stream, "%02X", &buffer)) && (i < _mem_length))
    {
      if(status == EOF)
        break;
      _mem[i>>2] |= buffer<<((3-(i%4))*8);
      i++;
    }
  }
  else
  {
    while ((status=fscanf(stream, "%02X", &buffer)) && (i < _mem_length))
    {
      if(status == EOF)
        break;
      _mem[i>>2] |= buffer<<((i%4)*8);
      i++;
    }
  }
  fclose(stream);
}

// -----------------------------------------------------------------------------
//  Interface Method : ahb_memory::write_to_file
// -----------------------------------------------------------------------------
void SW_interface::write_to_file(const sc_string& fn)
{
  unsigned int  i, j;
  char *tabstring = new char[_mem_length*3+1];
  char *p_tabstring = tabstring;
  FILE *stream;
  stream = fopen(expand(fn), "w");
  if(stream == 0)
  {
    cout << "ERROR: the data file: " << fn;
    cout << " can not be opened for reading. <" << name() << ">" << endl;
    sc_stop();
  }
  if (_big_endian)
  {
    // MSB is mapped to mem[0]
    for(i=0; i<_mem_length;)
    {
      for(j=0; (j<15) && (i<(_mem_length-1)); j++, i++)
      {
        fprintf(stream ,"%02X\t", (_mem[i>>2]>>((3-(i%4))*8))&0xff);
        p_tabstring += 3;
      }
      fprintf(stream, "%02X\n", (_mem[i>>2]>>((3-(i%4))*8))&0xff);
      p_tabstring += 3;
      i++;
    }
  }
  else
  {
    // LSB is mapped to mem[0]
    for(i=0; i<_mem_length;)
    {
```

```
        for(j=0; (j<15) && (i<(_mem_length-1)); j++, i++)
        {
          fprintf(stream ,"%02X\t", (_mem[i>>2]>>((i%4)*8))&0xff);
          p_tabstring += 3;
        }
        fprintf(stream, "%02X\n", (_mem[i>>2]>>24)&0xff);
        p_tabstring += 3;
        i++;
      }
    }
    fprintf(stream, "%c\n", '\0');
    fclose(stream);
  }

  // ------------------------------------------------------------------------
  //  Interface Method : ahb_memory::set_monitor
  // ------------------------------------------------------------------------
  void SW_interface::set_monitor()
  {
    int  i, j;
    int offset = 0;
    int trace_length = _mem_length;
    char *tabstring = new char[trace_length*3+1];
    if(tabstring == 0)
    {
      cout << "ERROR: unable to allocate memory. <" << name() << ">" << endl;
      sc_stop();
    }
    char *p_tabstring = tabstring;
    if (_big_endian)
    {
      // big endian, MSB is mapped to mem[0]
      for(i=0; i<trace_length;)
      {
        for(j=0; (j<15) && (i<(trace_length-1)); j++, i++)
        {
          sprintf(p_tabstring,"%02X\t",
  (_mem[(offset+i)>>2]>>((3-(i%4))*8))&0xff);
          p_tabstring += 3;
        }
        sprintf(p_tabstring,"%02X\n",
(_mem[(offset+i)>>2]>>((3-(i%4))*8))&0xff);
        p_tabstring += 3;
        i++;
      }
    }
    else
    {
      // LSB is mapped to mem[0]
      for(i=0; i<trace_length;)
      {
        for(j=0; (j<15) && (i<(trace_length-1)); j++, i++)
        {
          sprintf(p_tabstring,"%02X\t", (_mem[(offset+i)>>2]>>((i%4)*8))&0xff);
          p_tabstring += 3;
        }
        sprintf(p_tabstring,"%02X\n", (_mem[(offset+i)>>2]>>24)&0xff);
        p_tabstring += 3;
        i++;
      }
    }
    tabstring[trace_length*3]='\0';
    _mem_monitor_table->set(tabstring);
    delete[] tabstring;
  }
```

```
// ------------------------------------------------------------------------
//  Interface Method : memory::set_monitor(ahb_addr_t)
// ------------------------------------------------------------------------
void SW_interface::set_monitor(int idx_)
{
  char tabstring [12];
  int offset = idx_&(~0x3);
  if (_big_endian)
  {
    // big endian, MSB is mapped to mem[0]
    for(int i=0; i<4; ++i)
    {
      sprintf(tabstring,"%02X", (_mem[(offset+i)>>2]>>((3-(i%4))*8))&0xff);
      _mem_monitor_table->change(offset/16, (offset+i)%16, tabstring);
    }
  }
  else
  {
    // LSB is mapped to mem[0]
    for(int j=0; j<4; ++j)
    {
      sprintf(tabstring,"%02X", (_mem[(offset+j)>>2]>>((j%4)*8))&0xff);
      _mem_monitor_table->change(offset/16, (offset+j)%16, tabstring);
    }
  }
  _mem_monitor_table->apply();
}
```

The memory location `_mem[0]` is used for transferring the actual state of the UMTS slot synchronization software state machine. All other memory locations `_mem[x]` are used to pass data between the UMTS slot synchronization software and hardware parts.

References

[25.01a] 3GPP TS 25.201. Physical layer general description. Technical report, 3GPP, 2001.

[25.01b] 3GPP TS 25.211. Physical channels and mapping of transport channels onto physical channels (FDD). Technical report, 3GPP, 2001.

[25.01c] 3GPP TS 25.213. Spreading and modulation (FDD). Technical report, 3GPP, 2001.

[25.01d] 3GPP TS 25.214. Physical layer procedures (FDD). Technical report, 3GPP, 2001.

[25.01e] 3GPP TS 25.221. Multiplexing and channel coding (TDD). Technical report, 3GPP, 2001.

[25.01f] 3GPP TS 25.223. Multiplexing and channel coding (TDD). Technical report, 3GPP, 2001.

[25.01g] 3GPP TS 25.301. Multiplexing and channel coding (TDD). Technical report, 3GPP, 2001.

[25.01h] 3GPP TS 25.401. UTRAN overall description. Technical report, 3GPP, 2001.

[ARM] ARM. `http://www.arm.com`.

[CoC] Synopsys. *CoCentric System Studio User Guide version 2002-05, June 2002.*

[DV] Frank Schirrmeister Dietrik Verkest, Joachim Kunkel. System level design using C++. `http://www.sigda.acm.org`.

[Fit] Alan Fitch. Application of SystemC to HW/SW Co-Design. `http://www.home.cs.utwente.nl/~smit/codesign/SystemC_Fitch.pdf`.

[Goi99a] Alois M. J. Goiser. *Code Divison Multiple Access*. Technische Universitaet Wien, Oesterreich, 1999.

[Goi99b] Alois M. J. Goiser. *Spread Spectrum Kommunikation*. Technische Universitaet Wien, Oesterreich, 1999.

[Hor00] Ivor Hortons. *Beginning Visual C++ 6*. Wrox Press Inc., United States of America, 2000.

[HT00] Harri Holma and Antti Toskala. *WCDMA for UMTS, Radio Access For Third Generation Mobile Communications*. John Wiley & Sons, England, 2000.

[Pas] Sudeep Pasricha. Transaction level modeling of SoC with SystemC 2.0. `http://www.st.com`.

[Sysa] Synopsys, CoWare, Frontier Design. *SystemC Version 1.0 User Guide*.

[Sysb] Synopsys, CoWare, Frontier Design. *SystemC Version 2.0 User Guide*.

[TGS02] Grant Martin Thorsten Groetker, Stan Liao and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, Netherland, 2002.

[UMT] TECHCOM Consulting. *UMTS Introduction*.