OpenCores G.729A Codec

Version 1.0

October 2013

Contents

Introduction

This document describes OpenCores G.729A codec core available at http://opencores.org/project,g729a_codec. The core performs multi-channel 8kbps voice compression based on ITU-T G.729A standard, in both half-duplex and full-duplex modes.

Hopefully this work can be of use to somebody.

License

In accordance with the existing version of the OpenCores G.729A codec core. This work is licensed under the GNU Lesser General Public License. The license can be obtained at http://www.gnu.org/licenses/lgpl.html . As such, the following applies to all source files added to the OpenCores G.729A codec core.

G.729 includes patents from several companies and is licensed by Sipro Lab Telecom. Sipro Lab Telecom is the authorized Intellectual Property Licensing Administrator for G.729 technology and patent pool. In a number of countries, the use of G.729 may require a license fee and/or royalty fee.

Conventions

The following conventions are used in this document:

- Channel: a sequence of data packet related to the same audio stream.
- Coding (or encoding): the process of compressing audio samples.
- Core: the G.729A codec described in this document.
- Data packet: any sequence of 16-bit word transferred between the core and outside world (hereinafter called user logic), data packets can carry un-encoded, coded (or encoded) or state data.
- Decoding: the process of un-compressing audio samples.
- User logic: any circuitry connected to the core, and used to control it and/or exchanging data with it.

General description

This G.729A codec core performs coding and decoding of 16-bit LPCM audio samples according to ITU-T G.729A standard.

When coding, the core takes as input data packets composed of 80 16-bit LPCM audio samples and outputs compressed data packets consisting of 5 16-bit words.

When decoding, the core takes as compressed data packets consisting of 5 16-bit words and outputs data packets composed of 80 16-bit LPCM audio samples.

The core can operate in both half-duplex (coding-only or decoding-only) and full-duplex (coding + decoding) modes.

The core can handle multiple channels by intermixing processing of their data packets, corruption of state information related to each channel is avoided by saving them (to external memory) after processing a packet belonging to a channel, and restoring them (from external memory) before processing the next packet belonging to the same channel.

The maximum number of channels which can be intermixed is limited only by the core operating frequency. Channel state save/restore operations are not needed for single-channel operations.

Source code language

All source code files are written in synthesizable VHDL language.

Although the source code doesn't include vendor-specific features, it has been written having FPGA's in mind and therefore is more suitable to implementation on FPGA's.

FPGA-oriented features include, among others, the use of inferred dual-port RAM memories with one read/write and one read-only port, and the mixing of logic and memory instances in VHDL code.

Core architecture

The core has been designed following an ASIP-style (Application Specific Instruction Processor) approach, and it consists of a simple processor, which RISC-like instruction set has been augmented with instructions dedicated to G.729A coding/decoding operations, plus instruction and data memories and an interface module managing the processor itself and the data transfers between the core and user logic.

Every data transfer between the core and the user logic is actually implemented as a Direct Memory Access (DMA) operation on the processor data memory.

Data memory stores three types of data: constant data (in a read-only portion of it), channel state data (to be retained across decoding/encoding runs) and scratchpad data (to be overwritten at each run).

Core interface

The core interface consists of the following signals:

- CLK_i: clock input.
- RST_i: synchronous reset input (reset completes in one cycle).
- STRT_i: start input.
- OPS_i[3-1:0]: operation selecting input.
- RE_i: read-enable input.
- WE_i: write-enable input.
- DI_i[16-1:0]: data input.
- BSY_o: core busy output.
- DMAE_o: DMA-enabled output
- STS_o[3-1:0]: status output.
- DV_o: data valid output
- DO_o[IO_WIDTH-1:0]: data output.

STRT_i starts a core operation and must be asserted by user logic for one cycle. STRT_i signal level is ignored when signal BSY_o is asserted.

OPS_i[3-1:0] selects the operation to be performed by the core and must asserted by user logic on the same cycle where STRT_i is asserted. Valid core operations are listed below.

RE_i and WE_i signals control data transfers between user logic and the core. User logic asserts WE_i for one cycle when a 16-bit data word needs to be written to the core. User logic asserts RE_i for one cycle when a 16-bit data word needs to be read from the core.

DI_i[16-1:0] is used to write 16-bit data words to the core, data loaded by user logic on DI_i are valid, and can be latched by the core, on the same cycle where WE_i signal is asserted.

BSY_o acts as "core busy" flag and is asserted by the core while processing a data packet.

DMAE_o acts as "DMA-enabled" flag and is asserted by the core when it's in DMA mode (i.e. when data can be read/written using RE_i, WE_i, DI_i, and DO_o). This signal is currently redundant and can be ignored (left open).

STS_o[3-1:0] provides user logic with core status information specifying the type of data (decoded data packets, encoded data packet and state data packet) the core expects to be transferred and the direction of the transfer. Valid status values are listed below.

DV_o flags presence of valid data on DO_o. When DV_o is asserted, data on DO_o can be latched by user logic. This output is in some way redundant, because the information it provides can be obtained from RE_i too.

DO_o[16-1:0] is used to read 16-bit data words from the core, data loaded by the core on DO_o are valid, and can be latched by user logic, one cycle, or two cycles, after RE_i signal has been asserted (two cycles delay occurs when optional output registers are present, one cycle delay occurs when they're not).

All control signals are active-high: asserting a signal means driving it high, and de-asserting a signal means driving it low.

Valid OPS_i values

Value	Mnemonic	Purpose
000	RUNF	Run full-duplex (coding + decoding) operation
001	INIT	Initialize core (required before processing first packet of a channel)

010	RSTS	Restore channel state data
011	RUNC	Run half-duplex (coding-only) operation
100	RUND	Run half-duplex (decoding-only)
101	SAVS	Save channel state data
110	-	Reserved
111	-	Reserved

Valid `STS_o` values

Value	Mnemonic	Purpose
000	IDLE	Core idle (no operation in progress)
001	COD_DIN	Writing un-encoded data to core (encoding input)
010	COD_DOUT	Reading coded data from core (encoding output)
011	DEC_DIN	Writing coded data to core (decoding input)
100	DEC_DOUT	Reading en-encoded data from core (decoding output)
101	STT_DIN	Writing channel state data to core (restoring state)
110	STT_DOUT	Reading channel state data from core (dumping state)
111	RUN	Processing (no data transfer possible)

Core top-level module is named G729A_CODEC_SDP and is located in source file G729A_codec_sdp.vhd.

Top-level module provides the following configuration parameters (generics):

- `REGISTER_INPUTS`: when set to '1', input signals `STRT_i`, `OPS_i`, `RE_i`, `WE_i`, and `DI_i` are registered by the core. Default value is '0' (no registering).
- `REGISTER_OUTPUTS`: when set to '1', output signal `DO_o` is registered by the core. Default value is '0' (no registering).
- `SIMULATION_ONLY`: used for verification purposes only, ignore it or set to default value of '0'.
- `ST_FILE`: used for verification purposes only, ignore it.
- `WB_FILE`: used for verification purposes only, ignore it.
- `USE_ROM_MIF`: when set to '1', ROM memories data content is specified using a MIF (Memory Initialization File) format file (this option is suitable for synthesis with Altera tools). When set to '0', ROM memories data content is specified as a VHDL constant value (this option is suitable for simulation and for synthesis with Xilinx tools). Default value is '0' (no MIF file).

Flow of operations

Control signals

Interaction between the core and user logic mainly occurs by means of `OPS_i` input and `STS_o` output: `OPS_i` is used, by user logic, to send commands to the core (to be qualified by `STRT_i` signal), which responds by providing status information on `STS_o`. Status information is needed by user logic to detect when the core is ready to exchange data (either coded, un-encoded or state data) with it.
Additional status information is provided by `BSY_o` output, this signal informs user logic that an operation is in progress on the core: a new command can be issued only when `BSY_o` is de-asserted.

Single-channel operations, full-duplex mode

Before starting to perform encoding/decoding operations, the core must be initialized by issuing command INIT on OPS_i input and asserting STRT_i signal. The core responds by asserting signal BSY_o and driving RUN on STS_o output for the whole duration of the operation.

When core de-asserts BSY_o, actual encoding and decoding of next data packet can begin, this is accomplished by issuing command RUNF on OPS_i input and asserting STRT_i signal. The core responds by asserting signal BSY_o and driving DEC_DIN on STS_o output, thus informing user logic than encoded data are expected by the core.

User logic detects change in STS_o value and writes an encoded data packet (5 16-bit words) to core using WE_i and DI_i . The core counts incoming data words and, when whole packet has been transferred, responds by driving RUN on STS_o output, thus informing user logic than decoding is in progress.

When decoding is complete, the core drives DEC_DOUT on STS_o output, thus informing user logic than decoded data are ready to be read from the core.

User logic detects change in STS_o value and read a decoded data packet (80 16-bit words) from core using RE_i and DO_o . The core counts out coming data words and, when whole packet has been transferred, responds by driving COD_DIN on STS_o output, thus informing user logic than un-encoded data are expected by core.

User logic detects change in STS_o value and (if encoding is requested) writes an un-encoded data packet (80 16-bit words) to core using WE_i and DI_i . The core counts incoming data words and, when whole packet has been transferred, responds by driving RUN on STS_o output, thus informing user logic than encoding is in progress.

When encoding is complete, the core drives COD_DOUT on STS_o output.

User logic detects change in STS_o value and read an encoded data packet (5 16-bit words) from core using RE_i and DO_o . The core counts out coming data words and, when whole packet has been transferred, responds by de-asserting signal BSY_o and driving IDLE on STS_o output, thus informing user logic than the core is back to idle state and can perform a new operation.

Single-channel operations, half-duplex mode

Before starting to perform encoding/decoding operations, the core must be initialized by issuing command INIT on OPS_i input and asserting STRT_i signal. The core responds by asserting signal BSY_o and driving RUN on STS_o output for the whole duration of the operation.

When core de-asserts BSY_o, actual encoding/decoding of next data packet can begin, this is accomplished by issuing command RUNC (encoding-only), or RUND (decoding-only), on OPS_i input and asserting STRT_i signal. The core responds by asserting signal BSY_o and driving DEC_DIN, or COD_DIN, on STS_o output, thus informing user logic than encoded, or un-encoded, data are expected by the core.

User logic detects change in STS_o value and writes an encoded (5 16-bit words), or an un-encoded (80 16-bit words), data packet to core using WE_i and DI_i . The core counts incoming data words and, when whole packet has been transferred, responds by driving RUN on STS_o output, thus informing user logic than decoding is in progress.

When decoding is complete, the core drives DEC_DOUT, or COD_DOUT, on STS_o output.

User logic detects change in STS_o value and read a decoded (80 16-bit words), or coded (5 16-bit words), data packet from core using RE_i and DO_o .

The core counts out-coming data words and, when whole packet has been transferred, responds by de-asserting signal BSY_o and driving IDLE on STS_o output, thus informing user logic than the core is back to idle state and can perform a new operation.

Multiple-channel operations, full-duplex mode

Multiple-channel operations differ from single-channel ones because the core must be initialized for every channel and state data must be saved after every encoding/decoding operation and restored before every encoding/decoding operation on the same channel.

Before starting to perform encoding/decoding operations, the core must be initialized for first channel, by issuing command INIT on OPS_i input and asserting STRT_i signal. The core responds by asserting signal BSY_o and driving RUN on STS_o output for the whole duration of the operation.

When core de-asserts BSY_o, actual encoding and decoding of next data packet can begin, this is accomplished by issuing command RUNF on OPS_i input and asserting STRT_i signal. The core responds by asserting signal BSY_o and driving DEC_DIN on STS_o output, thus informing user logic than encoded data are expected by the core.

User logic detects change in STS_o value and writes an encoded data packet (5 16-bit words) to core using WE_i and DI_i . The core counts incoming data words and, when whole packet has been transferred, responds by driving RUN on STS_o output, thus informing user logic than decoding is in progress.

When decoding is complete, the core drives DEC_DOUT on STS_o output, thus informing user logic than decoded data are ready to be read from the core.

User logic detects change in STS_o value and read a decoded data packet (80 16-bit words) from core using RE_i and DO_o . The core counts out coming data words and, when whole packet has been transferred, responds by driving COD_DIN on STS_o output, thus informing user logic than un-encoded data are expected by core.

User logic detects change in STS_o value and (if encoding is requested) writes an un-encoded data packet (80 16-bit words) to core using WE_i and DI_i . The core counts incoming data words and, when whole packet has been transferred, responds by driving RUN on STS_o output, thus informing user logic than encoding is in progress.

When encoding is complete, the core drives COD_DOUT on STS_o output.

User logic detects change in STS_o value and read an encoded data packet (5 16-bit words) from core using RE_i and DO_o . The core counts out-coming data words and, when whole packet has been transferred, responds by de-asserting signal BSY_o and driving IDLE on STS_o output, thus informing user logic than the core is back to idle state and can perform a new operation.

User logic must now save state data for the current channel, this is accomplished by issuing command SAVS on OPS_i input and asserting STRT_i signal.

The core responds by asserting signal BSY_o and driving STT_DOUT on STS_o output, thus informing user logic than state data are expected to be read from the core (to be stored in some external memory).

User logic detects change in STS_o value and reads state data packet (1679 16-bit words) from core using RE_i and DO_o . The core counts out-coming data words and, when whole packet has been transferred, responds by driving IDLE on STS_o output, thus informing user logic that the core is back to idle state and can perform a new operation.

The same sequence of operations (initialization, coding/decoding and state saving) must then be repeated for each one of the remaining channels.

When such sequence has been performed for every channel (i.e., when first data packet from every channel has been processed), state data for first channel must be restored, this is accomplished by issuing command RSTS on `OPS_i` input and asserting `STRT_i` signal.

The core responds by asserting signal `BSY_o` and driving STT_DIN on `STS_o` output, thus informing user logic than state data are expected to be written to the core (from some external memory).

User logic detects change in `STS_o` value and writes a state data packet (1679 16-bit words) to core using `WE_i` and `DI_i` . The core counts incoming data words and, when whole packet has been transferred, responds by driving IDLE on `STS_o` output, thus informing user logic that the core is back to idle state and can perform a new operation.

A new data packet for first channel can now be processed in the same way described above for single-channel operations. When core completes this operation and returns to IDLE state, channel state has to be saved again.

This restore-run-save operation sequence must then be repeated for each of the remaining channels.

Multiple-channel operations, half-duplex mode

Multiple-channel operations in half-duplex mode are not described in details, differences between them and multiple-channel operations in full-duplex mode being the same existent between single-channel half- and full-duplex operations.

Performance

Minimum clock frequency for single-channel full-duplex operations is 27.5 MHz (actual minimum frequency is a bit lower, but the number of instructions executed in each encoding + decoding run varies from run to run, so it's safer to add some margin).

Rule-of-thumb for multiple-channel full-duplex operations is to add 27.5 MHz for each channel (55 MHz for two channels, 82.5 MHz for three channels, and so on).

Rule-of-thumb for half-duplex operations is to budget ~5.5 MHz/channel for decoding operations and ~22 MHz/channel for encoding operations.

Sample timing diagrams

The following timing diagrams assume optional input/output registers are not present.

Core reset

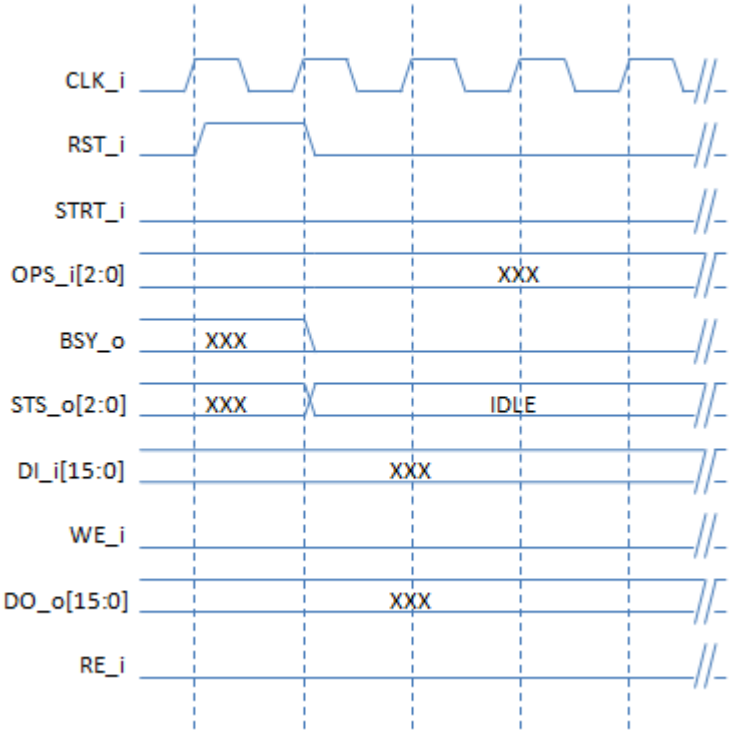

Fig. 1: reset timing diagram.

Initialization

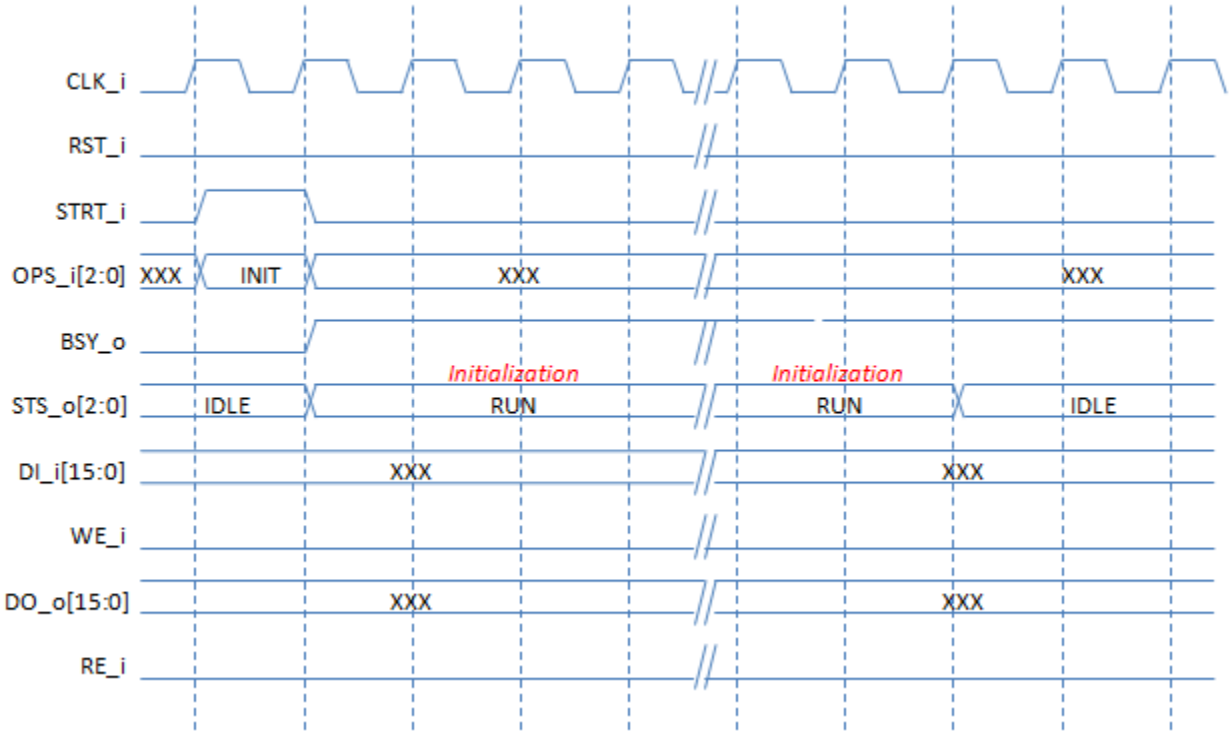

Fig. 2: Initialization timing diagram.

Encoding

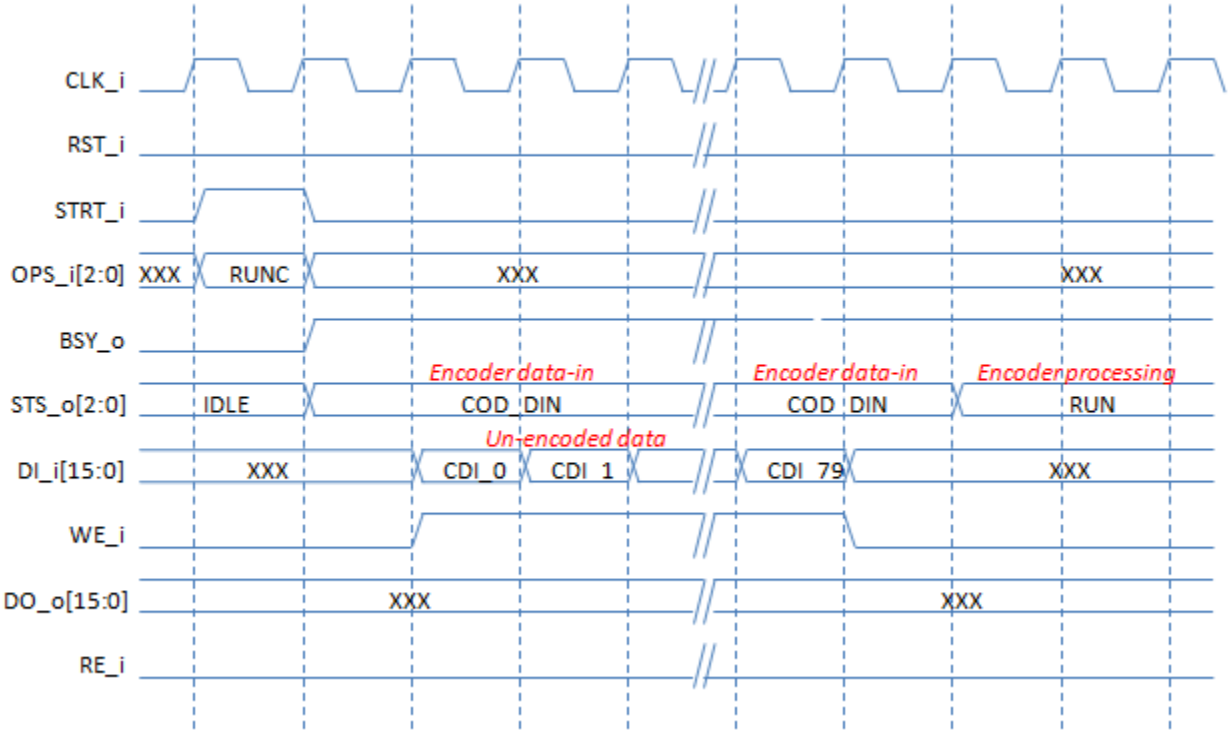

Fig. 3a: Encoding timing diagram (part I).

Decoding operation timing diagram differs from encoding one only for the command that is issued (RUND, instead of RUNC) and for input data type (encoded data vs. un-encoded data).

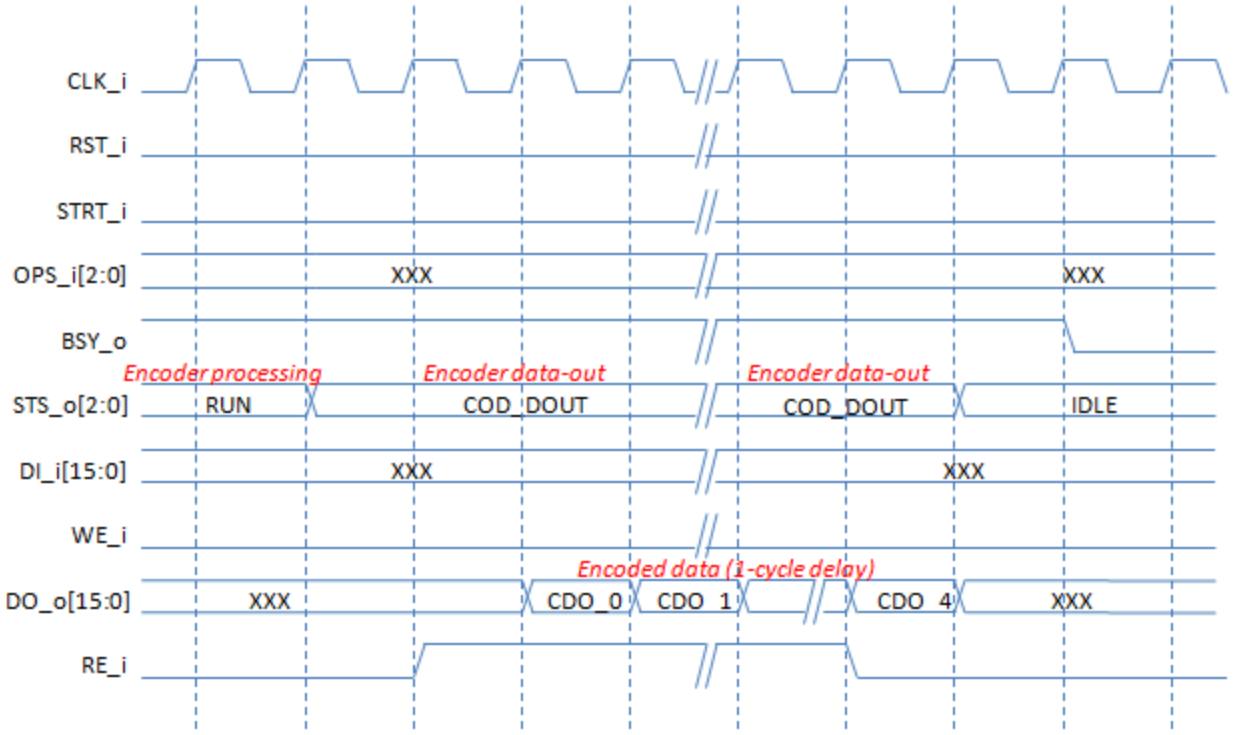

Fig. 3b: Encoding timing diagram (part II).

Decoding operation timing diagram differs from encoding one only for the output data type (un-encoded data vs. encoded data).

Channel state saving operation

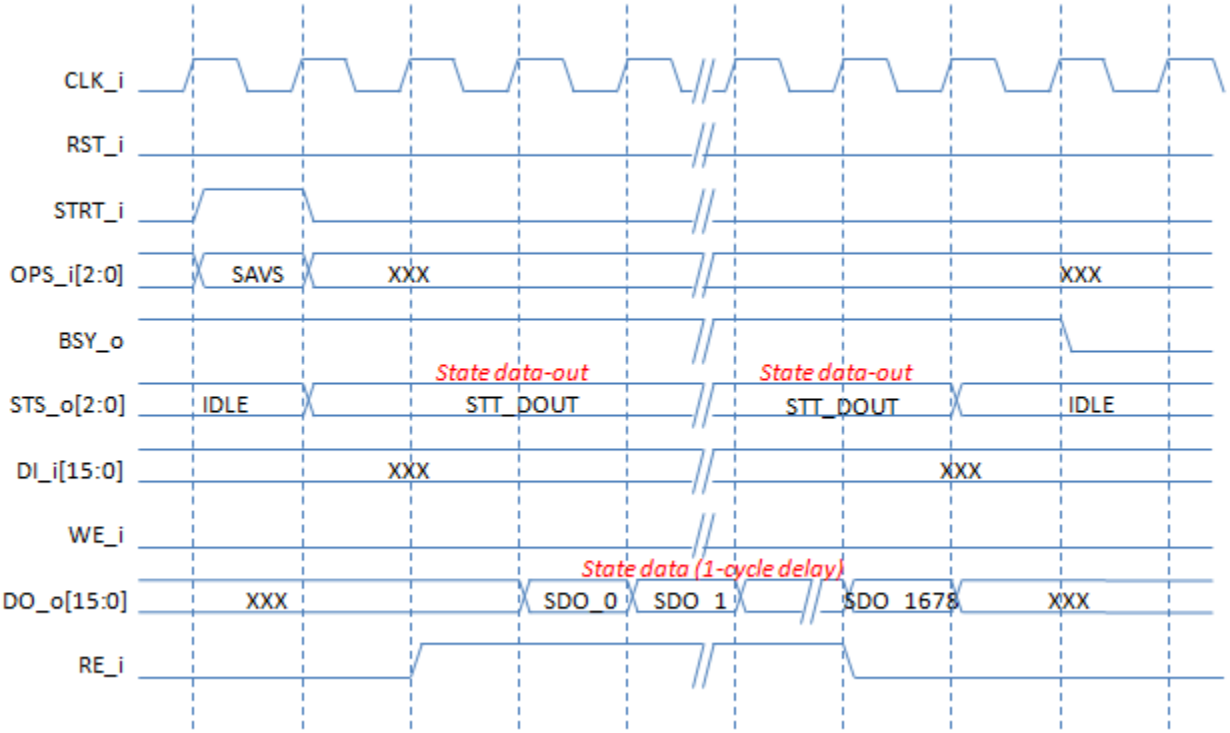

Fig. 4: Channel state saving operation timing diagram.

Channel state restoring operation

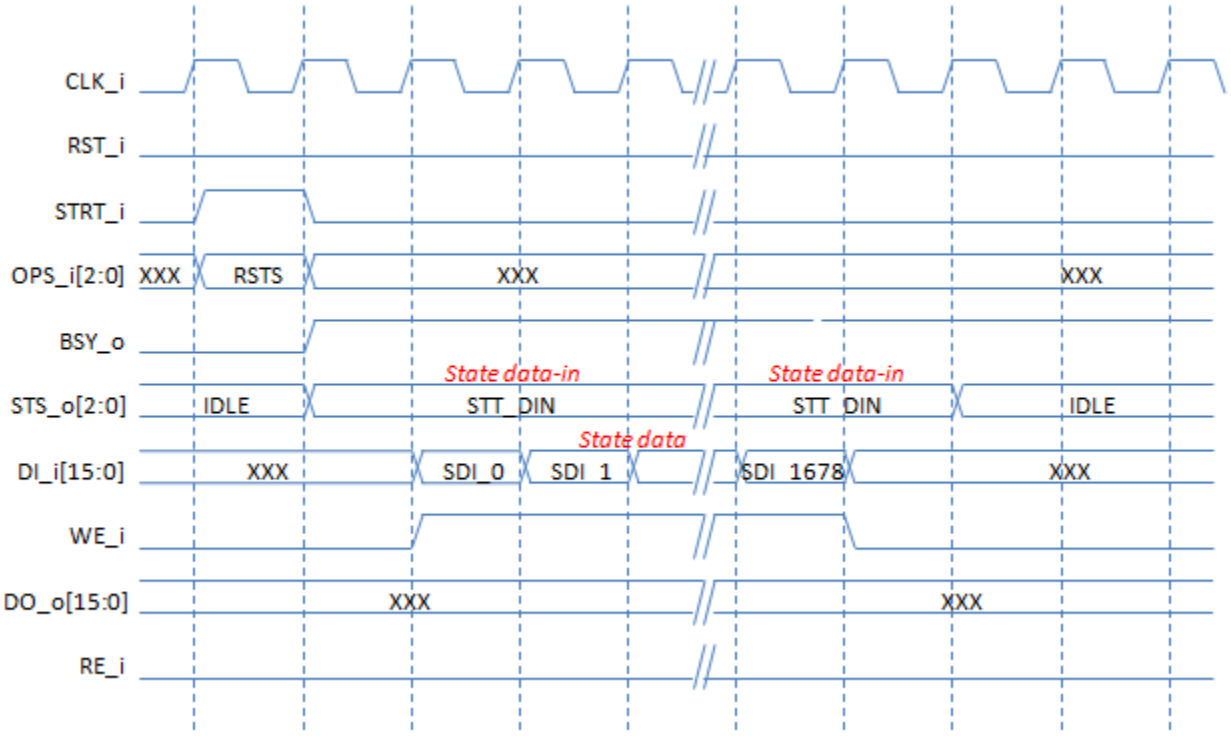

Fig. 5: Channel state restoring operation timing diagram.

Appendix A: Altera© Quartus II 9.1 synthesis test

Top-level module: `G729A_CODEC_SDP_SYN` (synthesis test-bench) from file `G729A_codec_sdp_SYN.vhd`.
Configuration parameters:

- `REGISTER_INPUTS` = '0'
- `REGISTER_OUTPUTS` = '0'
- `USE_ROM_MIF` = '1'.

Target device: EP3C25F324C8 (the one used by NEEK development board)
Target Fmax: 100 MHz (default synthesis options).
Results:

- Total logic elements: 6175 (24%)
- Total registers: 1728 (7%)
- Total memory bits: 319488 (53%)
- Embedded multipliers: 6 (5%)
- Fmax (worst case): 87.5 MHz

Appendix B: Xilinx© ISE 14.1 synthesis test

Top-level module: `G729A_CODEC_SDP_SYN` (synthesis test-bench) from file `G729A_codec_sdp_SYN.vhd`.
Configuration parameters:

- `REGISTER_INPUTS` = '0'
- `REGISTER_OUTPUTS` = '0'
- `USE_ROM_MIF` = '0'.

Target device: xc6vlx75t-2ff484.
Target Fmax: 143MHz (default synthesis options)..
Results:

- Number of slice registers: 1574 (1%)
- Number of slice LUTs: 4284 (9%)
- Number of RAMB36: 18 (11%)
- Number of DSP48E1s: 4 (1%)
- Fmax (worst case): 143 MHz (target met).