A flexible architecture framework for FPGA based coprocessors

Andreas Koltes

23/08/2007

Contents

1 Introduction

1.1 Overview

The proposed architecture described in this document is targeted at providing a flexible framework for the development of FPGA based coprocessors. All core components of the platform are independent from the specific data type handeled as well as from the hardware platform used. The framework provides the ability to quickly create specialised FPGA based coprocessor implementations. The basic design of the architecture is optimised to be combined with a general purpose (multi-)processor through a high-speed interconnection fabric like a HyperTransport channel.

1.2 Design goals

The proposed architecture is designed around the following design goals.

- Ability to handle arbitrary data types through an on-chip register file
- Configurable data record size in multiples of 32 bits consisting of 1 to 256 words
- Configurable size of register file (8, 16, 32, 64, 128, 256 registers available)
- Plug&Play ability to combine functional units handling different data types and operations
- Support for parallel operation of functional units
- Ability to dispatch one operation to one functional unit in each clock cycle
- Ability to end one off-loaded operation at least every second clock cycle
- Ability to operate generic functional units handling polymorphic data types
- Configurable flag register file allowing for multiple streams of flag sensitive operations to be carried out in parallel
- Configurable size of flag register file (8, 16, 32, 64, 128, 256 flag registers available)
- Pure load/store architecture
- Functional units can receive up to three input values and one input flag vector
- Functional units can send up to two output values and one output flag vector
- Out-of-order execution and reordering of received commands
- Configurable command look-ahead queue length between 1 and 16 slots
- Highly pipelined design allowing the FPGA hardware to run at high clock frequencies
- Software generated decoder and dispatcher circuits controls arbitrary sets of functional units
- Multiple functional units of the same type can be handled to optimise parallelism

1.3 High-level on-chip organisation

The architecture is designed to be 'register-file-centric', i.e. most components of the framework are built around two configurable register files containing data records and flags, respectively. The components implemented on the FPGA can be divided into *interface*, *main pipeline*, *functional unit* and *register file* circuit groups. Figure 1.1 on page 4 shows an outline of the on-chip organisation of components.

The framework is designed to operate together with high-performance interconnection fabrics like HyperTransport channels. For testing purposes, the use of slower interface options is possible, too. However, using these will bind the reachable performance of the system. A reference testbench using a serial interface based on an UART will be available. For test cases not requiring host interaction, it is also feasible to cache coprocessor commands in on-chip or on-board memory to simulate a high-speed data channel between FPGA and main CPU. Currently, the input and output modules of the framework process one byte each clock cycle. For operation with high-performance communication channels, these should be adjusted accordingly.

The core of the coprocessor consists of a six stage pipeline split into *message buffer*, *decoder*, *dispatcher*, *execution stage*, *message encoder* and *message serializer*. The decoder transforms the received command and data words into a signal vector and an input data record providing information for the dispatcher. The dispatcher performs all register file reads, enforces register locking and off-loads user operations to functional units. The decoder stage uses the *functional unit table* and the *register usage table* to discover suitable functional units and required register slots.

1.4 Plug&Play capability

The register file specified by the framework is able to handle arbitrary sized data records whose size is a multiple of 32 bits. The flag register file has a fixed record width of 16 bits. The only parts of the architecture that are actually aware of the data types held by the register file and the semantics of the flags in the flag register file are the functional units. This fact makes it easy to plug in specialised functional units handling application specific data types and operations.

Since the framework only handles locking and dependency checking of the register file a wide variety of data types ranging from integral values over vectors to complex types are possible. The flag register can be used to indicate and handle application specific conditions which are the result of an executed operation.

The interface provides a generic way to access registers as well as flags and also has limited support for primitive conditional operations to increase the operation throughput. However, since there is no support for branching operations, complex conditional behaviour still has to be controlled by the connected general purpose processor.

1.5 Portability

All components of the framework are designed to run across a wide variety of FPGA platforms. The only special feature required by the architecture is the availability of on-chip true dual port SRAM blocks which are accessible within one clock cycle. The pipelining of the architecture is aimed at allowing high clock frequencies, in many cases up to the maximum frequency possible for the on-chip memory blocks.

Despite not being required for the core framework, implementations of the architecture benefit from available dedicated multiplexors included into the FPGA cells like they are available in many high-performance FPGA platforms.

1.6 Application integration

From the perspective of the application a coprocessor built using the presented framework behaves similiar to a dedicated floating point or vector processing unit. Operations running on the coprocessor run in parallel to operations on the general purpose CPU. However, the main CPU may have to stall its operation until requested data from the coprocessor becomes available.

Despite its out-of-order execution capabilities, the coprocessor is guaranteed to outwardly behave as if all commands sent to it would be executed strictly in the order they were sent to the unit without any parallelism.

A possible extension to the coprocessor, which may give significant performance improvements for some applications, might be the support for multiple separated command input and data output streams. This would be especially beneficial for multi-threaded or otherwise parallel applications in which multiple execution streams are operating based on the same data types. Using multiple command streams, the register files of the coprocessor could implicitly be partitioned into disjunctive register groups and each of the command streams could operate on one of the register groups. This way operations belonging to different execution flows on the main CPU could be spread over the available functional units and being processed in parallel. It would also be possible to further increase parallelism by including a write arbiter capable of writing multiple output data records to the register file within a single clock cycle.

As long as the outward size of the data types used is similiar, it is possible and sensible to include functional units handling different or even polymorphic data types into a single coprocessor implementation. This gives a maximum of flexibility and can be used to speed up different, unrelated operations carried out by an application without changing the coprocessor implementation.

The current framework specifications does not provide any means of command validity checking and is therefore not suited for use in security critical environments. The behaviour of the coprocessor on the receipt of malformed commands is undefined by specification.

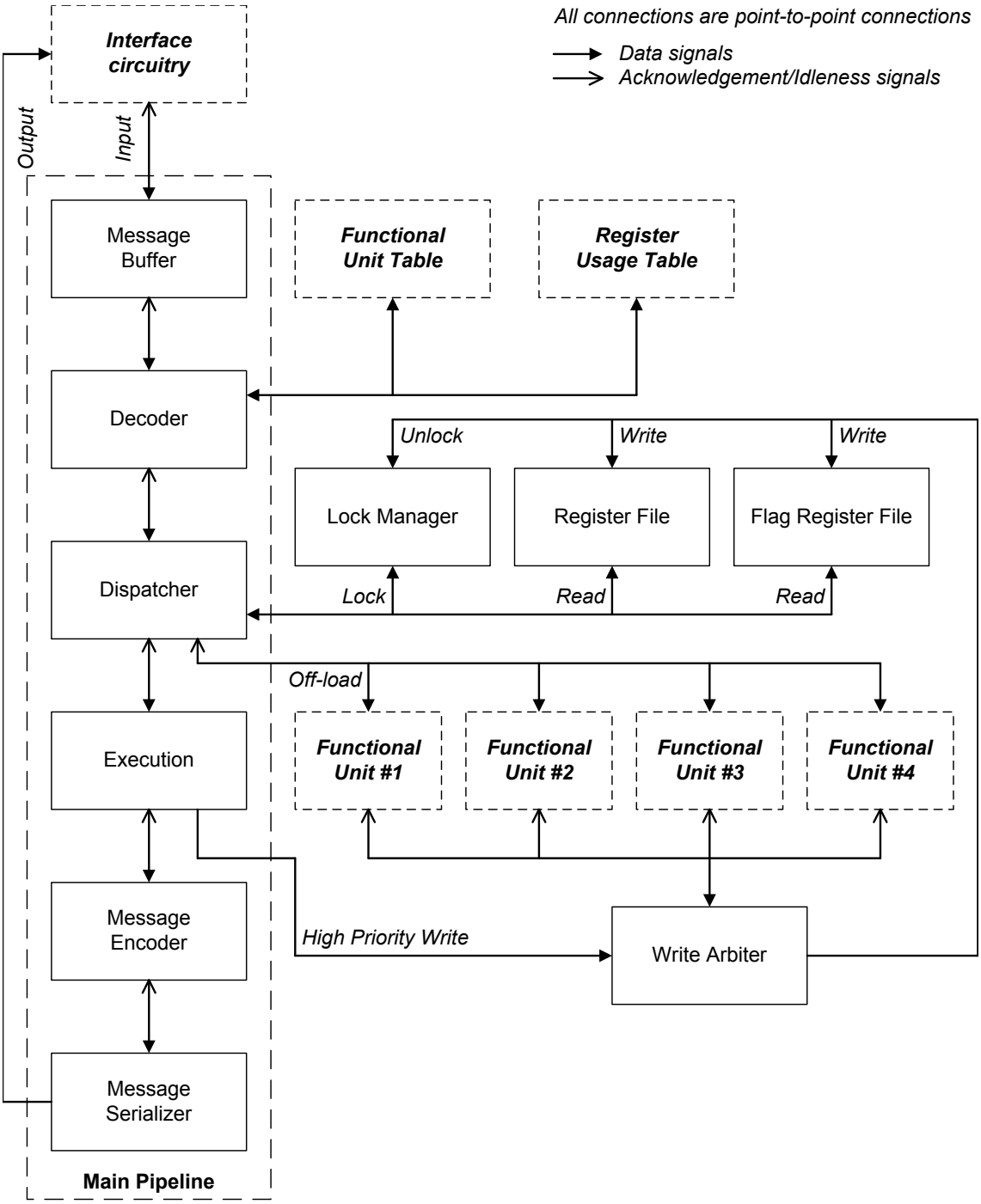

Figure 1.1: Outline of the on-chip organisation of components

2 Instruction set

2.1 Command format

All commands sent to the coprocessor consist of a single 64-bit command word which is optionally followed by one or more 32-bit data words depending on the requested operation. Command words are always expected to be sent to the coprocessor in big endian byte order (MSB first). The byte order of data words depends on the requested operation.

The highest bit of a command word specifies whether the requested operation is an I/O or core operation provided by the framework itself or whether it is a user operation being dispatched to a functional unit. In the first case the highest bit is set to 0, in the latter case, the highest bit is set to 1.

The rest of this document referrs to these instructions using the short forms given in the corresponding paragraphs.

2.2 I/O operations

I/O operations consist of load and store operations which are the only way of data communication between the coprocessor and the main CPU. Input instructions load data passed by the main CPU into a register of the register file or the flag register file. Output instructions send data stored in a register of the register file or the flag register file to the main CPU.

2.2.1 Register I/O instructions

The framework provides four instructions to load or to retrieve the full contents of a register. Since these instructions involve a large communication overhead, the partial register instructions should be used where sensible. The instructions support data word byte orders using either big or little endian encoding. The number of data words sent by the main CPU must match the size of the data records held by the register file, otherwise the behaviour of the coprocessor is undefined by specification. The endianess of the data coveres not only the individual 32-bit words but rather the whole data record, i.e. if using big endianess, the most significant data word must be sent first and if using little endianess, the least significant data word must be sent first, respectively. Figure 2.1 on page 5 shows the command word encodings of the instructions. The same applies to data words sent to the main CPU by the coprocessor.

	63	62	61	60	59	58	57	56	55	54	53 … 49	48	47 … 40	39 … 32	31 … 0
INB	0	1	0	0	0	0	0	0	0	0		0	Destination Register		
INL	0	1	0	0	0	0	0	0	0	0		1	Destination Register		
OUTB	0	0	1	0	0	0	0	0	0	0		0		Source Register	
OUTL	0	0	1	0	0	0	0	0	0	0		1		Source Register	

Figure 2.1: Encoding of register I/O instructions

2.2.2 Partial register I/O instructions

The partial register I/O instructions are useful to extract partial values from a register or to send small data amounts to the coprocessor. This is useful if the registers hold vectors or complex data types requiring only updates of single components. These instructions might also be useful to initialise registers with small values. Endianess is handled analogous to the full register I/O instructions. This means that the data word embedded within the command word can be sent in arbitrary byte order as long as its endianess is flagged correctly.

The indexes of the subwords of a data record are always counted from right to left starting with 0. So the least significant word has index 0 and the most siginificant word has index $n-1$ with n being the number of 32-bit words per data record.

Figure 2.2 on page 6 shows the command word encodings of the instructions. The instructions having a `Z` as the last character of the short name fill all other words of the data record with 0 bits.

	63	62	61	60	59	58	57	56	55	54	53 … 50	49	48	47 … 40	39 … 32	31 … 0
INSB	0	0	0	1	0	0	0	0	0	0		0	0	Destination Register	Word index (LSW is 0)	Data word
INSL	0	0	0	1	0	0	0	0	0	0		0	1	Destination Register	Word index (LSW is 0)	Data word
OUTSB	0	0	0	0	1	0	0	0	0	0		0	0	Word index (LSW is 0)	Source Register	
OUTSL	0	0	0	0	1	0	0	0	0	0		0	1	Word index (LSW is 0)	Source Register	
INSZB	0	0	0	1	0	0	0	0	0	0		1	0	Destination Register	Word index (LSW is 0)	Data word
INSZL	0	0	0	1	0	0	0	0	0	0		1	1	Destination Register	Word index (LSW is 0)	Data word

Figure 2.2: Encoding of partial register I/O instructions

2.2.3 Flag register I/O instructions

The flag register I/O instructions provided by the framework transfer the contents of an entire flag register from the main CPU to the coprocessor or vice versa. Please note that the indexes of flags are always counted from right to left starting with 0. So the least significant flag bit has index 0 and the most significant flag bit has index 15. The flags must be included in the command word in big endian byte order. Flags sent back to the main CPU are padded with zero bits to a length of 32 bits. Figure 2.3 on page 6 shows the command word encodings of the instructions.

	63	62	61	60	59	58	57	56	55	54	53 … 32	31 … 24	23 … 16	15 … 0
INFL	0	0	0	0	0	1	0	0	0	0		Destination Flag Register		Flags
OUTFL	0	0	0	0	0	0	1	0	0	0			Source Flag Register	

Figure 2.3: Encoding of flag register I/O instructions

2.3 Register modification operations

2.3.1 Flag instructions

The flag operations copy the contents of the specified flag register into another one and optionally modify them on-the-fly according to the bit mask sent to the coprocessor. All flags whose corre-

sponding bit in the flag mask are set are modified. The remaining flags stay unchanged. Setting the bit mask to all 0 bits is effectively a pure copy operation in all three cases. The contents of the source flag register stay unchanged unless source and destination registers are the same.

The instruction `STFL` sets all selected flag bits to 1 and `CLFL` sets them to 0, respectively. `CMFL` toggles the selected flag bits. Figure 2.4 on page 7 shows the command word encodings of the instructions.

	63	62	61	60	59	58	57	56	55	54	53 … 50	49	48	47 … 32	31 … 24	23 … 16	15 … 0
STFL	0	0	0	0	0	0	0	1	0	0		1	0		Destination Flag Register	Source Flag Register	Flag Mask
CLFL	0	0	0	0	0	0	0	1	0	0		0	0		Destination Flag Register	Source Flag Register	Flag Mask
CMFL	0	0	0	0	0	0	0	1	0	0			1		Destination Flag Register	Source Flag Register	Flag Mask

Figure 2.4: Encoding of flag instructions

2.3.2 Move instructions

Move instructions copy an entire data record from one register to another. It is possible to bind this copy process to a condition specified in terms of flags. The contents of the source register are copied to the destination register only, if one of the flag bits in the specified flag register, which was selected by the specified bit mask, is set to 1. In the case of the conditional move instructions containing an `A` in their name, all of the selected flag bits have to be set to 1. The variants of the conditional move instructions having a `Z` in their name behave like the normal conditional move instruction with the difference, that the target register is set to all 0 bits, if the specified condition is not met. Figure 2.5 on page 7 shows the command word encodings of the instructions.

	63	62	61	60	59	58	57	56	55	54	53 … 51	50	49	48	47 … 40	39 … 32	31 … 24	23 … 16	15 … 0
MOV	0	0	0	0	0	0	0	0	0	1		0	0	0	Destination Register	Source Register			
CMOV	0	0	0	0	0	0	0	0	0	1		0	0	1	Destination Register	Source Register		Flag Reg.	Flag Mask
CMOVZ	0	0	0	0	0	0	0	0	0	1		1	0	1	Destination Register	Source Register		Flag Reg.	Flag Mask
CMOVA	0	0	0	0	0	0	0	0	0	1		0	1	1	Destination Register	Source Register		Flag Reg.	Flag Mask
CMOVAZ	0	0	0	0	0	0	0	0	0	1		1	1	1	Destination Register	Source Register		Flag Reg.	Flag Mask

Figure 2.5: Encoding of move instructions

2.4 User operations

The proposed framework supports for embedding of up to 256 different types of functional units into the coprocessor. It is possible to include multiple functional units of the same type to reach a higher degree of parallelism.

Each functional unit can support up to three input values, up to two output values as well as an input and an output flag vector allowing for a maximum degree of flexibility and parallelism. Further details about the implementation of functional units is given in the next chapter.

There are four different modes available for encoding commands to the functional units. Besides the register indexes, the commands contain the function code of the requested functional unit as well as a variety code. The function code is handled by the dispatcher whereas the variety

code is sent to the functional unit as is. Please note that the maximum number of encodable functional unit types having three input parameters is 8 and that there are restrictions regarding the number of supported variety codes. All variety codes whose encoding is shorter than 8 bits get zero-extended to a length of 8 bits before they are sent to the functional unit for processing. Figure 2.6 on page 8 shows the available command word encodings of the user instructions.

	63	62	61	60–59–58–57–56–55	53–52–48	47–40	39–32	31–24	23–16	15–8	7–0
Mode A	1	0	0	FUNC	VAR	Destination Register #1	Source Register #1	Destination Flag Register	Source Flag Register	Destination Register #2	Source Register #2
Mode B	1	0	1	FUNC	VAR	Destination Register #1	Source Register #1	Destination Flag Register	Source Flag Register	Destination Register #2	Source Register #2
Mode C	1	1	0	FN VAR	Source Register #3	Destination Register #1	Source Register #1	Destination Flag Register	Source Flag Register	Destination Register #2	Source Register #2
Mode D	1	1	1	FUNC VAR	Source Register #3	Destination Register #1	Source Register #1	Destination Flag Register	Source Flag Register	Destination Register #2	Source Register #2

Figure 2.6: Encoding of user instructions

3 Functional Units

3.1 Overview

The functional units are the components of the coprocessor implementing the actual application logic. They are the only components of the coprocessor that are aware of the data type of the data records stored in the register file. They are also the only part of the framework being aware of the semantics of the flag bits in the flag registers.

The proposed framework supports for embedding of up to 256 different types of functional units into the coprocessor. It is possible to include multiple functional units of the same type to reach a higher degree of parallelism.

Each functional unit can support up to three input values, up to two output values as well as an input and an output flag vector allowing for a maximum degree of flexibility and parallelism. In the case of functional units requiring three different input values, the maximum number of encodable functional unit types having three input parameters is 8 if the variety code uses 2 bits. If the variety code requires 3 bits, the maximum number of encodable functional units having three input registers is 4. Therefore the lower 8 or 4 function codes, respectively, should be reserved for functional units of this kind.

None of the input or output registers are required by the framework. It is theoretically possible to build a functional unit without any inputs or outputs, despite not being of any use in practical applications.

Functional units have to be created as VHDL modules following the interface signature described in Section 3.2. The creation of the full coprocessor implementation is done automatically by a software tool assembling the functional units using *unit description files* as specified in Section 3.3. The tool also automatically generates the functional unit table and the register usage table used by the decoder and the top-level module instantiating the various components and configuring them using VHDL generics.

3.2 Interface

For the coprocessor build tool to be able to automatically assemble the coprocessor specification, all functional units have to follow the following interface specification.

It is absolutely necessary that every functional unit samples and processes its inputs in a single clock cycle. The dispatcher asserts the `dispatch` signal for a single clock cycle and routes the required data records and flag bits to the input ports of the functional unit. The functional unit must clear the `idle` signal to the next clock edge to signal the dispatcher that the unit is busy if the functional unit can only handle one request in parallel. If a functional unit is able to handle multiple requests internally, e.g. using a pipelined architecture, it is allowed to keep the `idle` signal asserted. However, even if accepting multiple inputs, a functional unit must not make any assumptions about the point in time the write arbiter will forward its write requests to the register files. A functional unit accepting multiple inputs must therefore be able to buffer all required data internally for an unknown amount of time.

As soon as the flag output ports are stabilised on the output flag bits the unit has to assert the `flags_ready` signal which tells the write arbiter to write the new flag bits to the flag register file and to unlock the corresponding flag register. The write arbiter acknowledges the flag write process by asserting the `flags_acknowledge` signal for a single clock cycle. The functional unit has to clear the `flags_ready` signal to the next clock edge. Implementation of conditional logic is supported

by the `flags_abort` signal. Its semantics are identical to those of the `flags_ready` signal with the exception, that the contents of the destination register are left unchanged. Assertion of the `flags_ready` and `flags_abort` is mutually exclusive.

As soon as the data output ports are stabilised on the output data of the unit the `data_ready` signal must be asserted. This signal must stay asserted until the `acknowledge` signal gets asserted by the write arbiter. As soon as this signal gets asserted the functional unit must behave in one of two ways: If the functional unit outputs two data records, the second output data set has to be routed to the data output ports. The `data_ready` signal can either stay asserted if this is done within a single clock cycle or can be cleared and reasserted later. After the last output data record got acklowledged by the arbiter, the functional unit has to clear the `data_ready` signal to the next clock edge. Implementation of conditional logic is supported by the `data_abort` signal. Its semantics are identical to those of the `data_ready` signal with the exception, that the contents of the destination register are left unchanged. Assertion of the `data_ready` and `data_abort` is mutually exclusive.

The unit must not change the output register indexes passed to it by the dispatcher and has to place the received output register indexes on the corresponding output ports during the output of results.

When all output flags and data records got acknowledged by the write arbiter and the functional unit gets ready for operation again, it has to reassert the `idle` signal to indicate to the dispatcher that it is ready to execute the next instruction.

3.3 Unit description files

The coprocessor build tool uses *unit description files* to generate the functional unit table and the register usage table used by the decoder and to assemble the various components of the coprocessor. A unit description file contains information about the inputs and outputs of a functional unit as well as supported data record sizes. The information is specified as a simple text file containing key-value pairs. Figure 3.1 on page 11 shows an example of an unit description file.

The function code of the functional unit is specific as decimal number. The variety codes are specified in decimal form as well and are followed by the name and register requirements of the corresponding subfunction in the following order: Instruction Name, Flag Input, Data Input #1, Data Input #2, Data Input #3, Flag Output, Data Output #1, Data Output #2.

```
name=Large integer arithmetic unit
description=This functional unit provides a double-level hybrid
  carry-look-ahead adder supporting addition and subtraction
  as well as comparision of large integers.
file_name=liparith.vhdl
module_name=LIPArithmeticUnit
supported_word_counts=1-8
function_code=16
variety=4,ADD,No,Yes,Yes,No,Yes,Yes,No
variety=5,ADC,Yes,Yes,Yes,No,Yes,Yes,No
variety=38,SUB,No,Yes,Yes,No,Yes,Yes,No
variety=39,SBB,Yes,Yes,Yes,No,Yes,Yes,No
variety=12,INC,No,Yes,No,No,Yes,Yes,No
variety=44,DEC,No,Yes,No,No,Yes,Yes,No
variety=54,NEG,No,No,Yes,No,Yes,Yes,No
variety=34,CMP,No,Yes,Yes,No,Yes,No,No
variety=35,CMPB,Yes,Yes,Yes,No,Yes,No,No
```

Figure 3.1: Example unit description file

Input port	Type	Required	Comments
clock	STD_LOGIC	Yes	The module is expected to sample all input signal at the rising clock edge of the clock signal
reset	STD_LOGIC	Yes	The reset condition is signaled by the signal being asserted which is guaranteed to be the case for at least one clock cycle at the beginning of coprocessor operation
dispatch	STD_LOGIC	Yes	Gets asserted by the dispatcher for one clock cycle to signal the module to start operation - the module has to sample all input ports within a single clock cycle
variety_code	STD_LOGIC_VECTOR[7 .. 0]	Yes	Variety code
flags_input	STD_LOGIC_VECTOR[15 .. 0]	No	Input flag bits
data_input_1	STD_LOGIC_VECTOR [32 * data_words - 1 .. 0]	No	First input data record
data_input_2	STD_LOGIC_VECTOR [32 * data_words - 1 .. 0]	No	Second input data record
data_input_3	STD_LOGIC_VECTOR [32 * data_words - 1 .. 0]	No	Third input data record
flags_output_reg_input	STD_LOGIC_VECTOR[7 .. 0]	No	Index of flag register receiving the output flag bits
data_output_reg_1	STD_LOGIC_VECTOR[7 .. 0]	No	Index of data register receiving the first output data record
data_output_reg_2	STD_LOGIC_VECTOR[7 .. 0]	No	Index of data register receiving the second output data record
flags_acknowledge	STD_LOGIC	No	Gets asserted by the write arbiter for one clock cycle to signal the module that the flag output signals got written to the flag register file
data_acknowledge	STD_LOGIC	No	Gets asserted by the write arbiter for one clock cycle to signal the module that the data output signals got written to the register file
Output port	**Type**	**Required**	**Comments**
idle	STD_LOGIC	Yes	Has to be asserted when it is in idle state to signal the dispatcher that it is able to receive data
flags_ready	STD_LOGIC	No	Has to be asserted to signal the write arbiter that the flag output ports are ready
flags_abort	STD_LOGIC	No	Has to be asserted to signal the write arbiter that the acquired destination register shall be unlocked leaving its contents unchanged
flags_output	STD_LOGIC_VECTOR[15 .. 0]	No	Output flag bits to be written to the flag register file
flags_output_reg	STD_LOGIC_VECTOR[7 .. 0]	No	Index of flag register receiving the output flag bits
data_ready	STD_LOGIC	No	Has to be asserted to signal the write arbiter that the data output ports are ready
data_abort	STD_LOGIC	No	Has to be asserted to signal the write arbiter that the acquired destination register shall be unlocked leaving its contents unchanged
data_output	STD_LOGIC_VECTOR [32 * data_words - 1 .. 0]	No	Output data to be written to the data register file
data_output_reg	STD_LOGIC_VECTOR[7 .. 0]	No	Index of data register receiving the output data record
Parameter	**Type**	**Required**	**Comments**
data_words	Integer	Yes	Length of data records in 32-bit words

Table 3.1: Functional unit interface

4 Large Integer Package

4.1 Overview

The Large Integer Package is mainly intended as reference implementation for demonstration and research purposes. The package provides functional units acting as wrappers around Altera provided library components. The functional units support generic integer lengths. However, since this package is developed to be used on comparatively small devices not having dedicated hardware multipliers, the performance of these functional units compared to a high-performance implementation is comparatively poor.

4.2 Flag semantics

All flags input or output by the functional units of the Large Integer Package follow the same semantics. There are several flags specifiying arithmetic conditions as well as general numeric conditions. Unless otherwise stated, all functional units set their flag output (if any) according to these semantics.

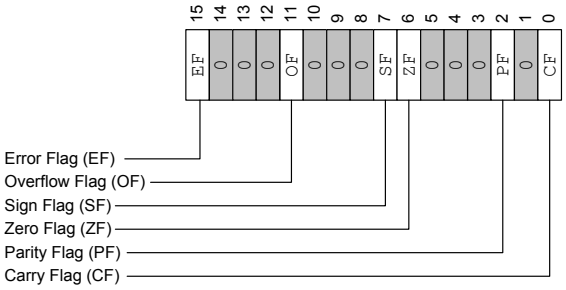

Figure 4.1: Large Integer Package flag semantics

The *Carry Flag (CF)* is filled with the resulting carry or borrow-in bit, respectively, resulting of unsigned addition and subtraction operations and also serves as input to the multi-word arithmetic functions. The *Overflow Flag (OF)* signals an overflow of the signed result of an arithmetic operation. The *Sign Flag (SF)* is set to 1 if the signed result of an operation is negative and to 0 otherwise.

The *Zero Flag (ZF)* indicates that the result of an operation is equal to zero. The *Parity Flag (PF)* is set to the least significant bit of the result of a computation. If it is set to 1, the result of the operation is odd, otherwise it is even. The *Error Flag (EF)* indicates an arithmetic error condition like for example a division by zero. If this flag is set, the contents of the destination registers (if any) are undefined by specification.

4.3 Arithmetic unit

The arithmetic unit of the Large Integer Package is able to do binary as well as two's complement additions, subtractions as well as comparisions. Multi-word operation is supported through an

externally provided carry bit read from the input carry flag. All operations with the exception of the negation instruction are applied to the first and second source operand in the case of two input operands and to the first operand in the case one input operand. The negation instruction is applied to the second operand only, for reasons of logic compactness. Figure 4.2 on page 14 shows the encoding of the instructions supported by the arithmetic unit.

	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47–40	39–32	31–24	23–16	15–8	7–0
ADD	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	Destination Register #1	Source Register #1	Destination Flag Register			Source Register #2
ADC	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	Destination Register #1	Source Register #1	Destination Flag Register	Source Flag Register		Source Register #2
SUB	1	0	0	1	0	0	0	0	0	0	1	0	0	1	1	0	Destination Register #1	Source Register #1	Destination Flag Register			Source Register #2
SBB	1	0	0	1	0	0	0	0	0	0	1	0	0	1	1	1	Destination Register #1	Source Register #1	Destination Flag Register	Source Flag Register		Source Register #2
INC	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	Destination Register #1	Source Register #1	Destination Flag Register			
DEC	1	0	0	1	0	0	0	0	0	0	1	0	1	1	0	0	Destination Register #1	Source Register #1	Destination Flag Register			
NEG	1	0	0	1	0	0	0	0	0	0	1	1	0	1	1	0	Destination Register #1		Destination Flag Register			Source Register #2
CMP	1	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0		Source Register #1	Destination Flag Register			Source Register #2
CMPB	1	0	0	1	0	0	0	0	0	0	1	0	0	0	1	1		Source Register #1	Destination Flag Register	Source Flag Register		Source Register #2
Function code: 16											Complement second input	First input zero	Second input zero	Output data	Fixed carry	Use carray flag						

Figure 4.2: Encoding of Large Integer Package arithmetic instructions

4.4 Logic unit

The logic unit of the Large Integer Package is able to do a variety of basic bitwise logic operations. All operations are applied to the first and second source operand in the case of two input operands and to the first operand in the case one input operand. Figure 4.3 on page 15 shows the encoding of the instructions supported by the logic unit.

	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47–40	39–32	31–24	23–8	7–0
NOT	1	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1	Destination Register #1	Source Register #1	Destination Flag Register		
AND	1	0	0	1	0	0	0	1	0	1	0	0	0	0	1	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
OR	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
XOR	1	0	0	1	0	0	0	1	0	1	0	0	1	0	0	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
NAND	1	0	0	1	0	0	0	1	0	1	0	1	0	0	1	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
NOR	1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
NXOR	1	0	0	1	0	0	0	1	0	1	0	1	1	0	0	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
AND-NOT	1	0	0	1	0	0	0	1	0	1	1	0	0	0	1	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
OR-NOT	1	0	0	1	0	0	0	1	0	1	1	0	0	1	0	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
XOR-NOT	1	0	0	1	0	0	0	1	0	1	1	0	1	0	0	0	Destination Register #1	Source Register #1	Destination Flag Register		Source Register #2
TEST	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0		Source Register #1	Destination Flag Register		Source Register #2
										Output data	Complement second input	Complement result	XOR	OR	AND	Identity					

Function code: 17

Figure 4.3: Encoding of Large Integer Package logic instructions

4.5 Future expansion

The following function codes are reserved for future expansion of the Large Integer Package.

Functional unit	Function code
Multiplier	18
Divider	2
Barrel shifter	20
Bit scanner	21
Permuter	22
Exponentiation	23

Table 4.1: Reserved function codes of Large Integer Package