Modlib: A Flexible Module Library for High-Level Language Compilation

Technical Report
Version: 1.0

Benjamin Thielmann, Hagen Gädke-Lütjens

Integrated Circuit Design (E.I.S.)
Technische Universität Braunschweig
email: {thielmann|gaedke}@eis.cs.tu-bs.de

August 25, 2010

Abstract

In this technical report we describe the highly parameterizable, technology-independent module library Modlib for use by high-level language to hardware compiler back-ends for synchronous logic. Modlib not only supports all high-level language operators (including memory accesses), but also provides a wide spectrum of usage modes: covering static and dynamic scheduling, speculative predicated execution, pipeline balancing, and explicit cancelling of mis-speculated computations. Modlib has been presented on the FPL 2010 conference [GäTK10] and is available for download [GäTK].

1 Introduction

When compiling from a high-level language such as C to a synchronous hardware design in a hardware description language (HDL), every compiler somehow has to find a way to convert the high-level operators into a hardware representation. These include arithmetic (such as addition, multiplication, division), casts, and memory accesses. A direct conversion is often only feasible for *simple* arithmetic such as integer addition and subtraction, due to limitations of synthesis tools. For more complex operators (e.g., division), a simple HDL statement such as `a / b` probably does not include enough information to represent the programmer's (or compiler's) needs: Some synthesis tools would simply create a combinational circuit for the complete division, which may have a considerable negative impact on the clock frequency[1].

Therefore, we have developed Modlib, a module library containing hardware descriptions for all operators needed for compilation from high-level language descriptions, including 32 and 64 bit floating point. Modlib focuses C as source language, but is applicable to other languages, too, because operators have similar semantics in most high-level languages. Modlib is written in synthesizable VERILOG HDL (version 2001) on the register transfer level (RTL) and thus is technology-independent, which renders it suitable for a broad range of HW compilers. Complex modules such as floating point are embedded as EDIF netlists[2] and equipped with an interface which is consistent among all Modlib modules.

Modlib is highly parametrized, e.g., it supports both static as well as dynamic scheduling. In static scheduling, operators are controlled via a clock enable signal (CE): If CE == 1, the operator executes a computation cycle; otherwise, it stalls during the current cycle. In dynamic scheduling, the operator receives a START signal and then computes during every consecutive cycle, until it finishes its computation and outputs a RESULT_READY. In dynamic scheduling mode, the Modlib operators support predicated speculative execution. They can handle cancel tokens (CT)

[1]Note that Modlib is essentially designed for synchronous hardware, i.e., every operator is controlled by a clock signal. Some compilers such as CASH [Budi03] create asynchronous hardware, unsuited for Modlib.

[2]These EDIFs have to be created using an IP core generation tool such as Xilinx CoreGen. Modlib includes CoreGen scripts which can be simply executed to create the EDIFs needed for Modlib.

[GäKo08, GäKo07, GäSK08, GäTK10], also called anti tokens [BrGa03], down tokens [Kasp05, KoKa05], or kill tokens [JúCK06], to be able to suppress unnecessary speculative operations. CTs are even supported in two different modes: dynamic and static (sometimes referred to as active and passive, respectively). In dynamic mode, the CT is propagated to predecessor operators. However, this may require more complex control circuits in the logic which connects the operator instances[3]. The static CT model makes the CTs wait inside the operator. When the associated speculative input enters the operator, it is then discarded.

This report is structured as follows. Section 2 lists the Modlib operators and gives a short behavioral description for each operator. The generic module layout is described in Section 3. Section 4 details the module parameters, and Section 5 the I/O semantics. Modules differ from the generic implementation are addressed in Section 6. Finally, Section 7 illustrates how to create the EDIFs needed for the complex operators.

2 Operator Overview

Modlib consists of a bunch of VERILOG modules, each module packed in a separate file. A module `module_name` is contained in the file `module_name.v`. Most of the modules include a common file containing handshaking logic. There are three different handshaking logic files: `handshaking_assigns_blackbox_include.v` (used by modules which embed an EDIF netlist), `handshaking_assigns_mux.v` (only used by the mux module), and `handshaking_assigns_verilog_only.v` (used by the remaining modules). The following Subsections describe the semantics of each module.

Note that the overflow behavior is irrelevant for compilation from ANSI C, because ANSI C does not define it either. In fact, what happens on an overflow depends on the simulation or synthesis tool used. However, a typical behavior for integer arithmetic is that the bits exceeding the bitwidth of the result output are just discarded, keeping the result modulo the maximum value $+1$. E.g., the result bitwidth of the unsigned integer addition $255 + 2$ is 8 bits, the result will be $1 = 257 mod 256$.

2.1 Integer Arithmetic

2.1.1 Addition

- Module name: `add`
- Corresponding C operator: `+`
- Data inputs: `A`, `B`

[3]This logic, which we call *sequencer*, is *not* part of Modlib and is thus not described in this report.

- Data output: R

The add operator computes $R = A + B$.

2.1.2 Subtraction

- Module name: sub
- Corresponding C operator: -
- Data inputs: A, B
- Data output: R

The sub operator computes $R = A - B$.

2.1.3 Multiplication

- Module name: mul
- Corresponding C operator: $*$
- Data inputs: A, B
- Data output: R

The mul operator computes $R = A * B$.

2.1.4 Division

- Module name: divint
- Corresponding C operator: /
- Data inputs: A, B
- Data output: R

The divint operator computes $R = A/B$; R is assigned the integral part of the quotient.

2.1.5 Modulus

- Module name: mod
- Corresponding C operator: %
- Data inputs: A, B
- Data output: R

The mod operator computes $R = AmodB$, i.e., R is assigned the remainder of the division A/B.

2.2 Floating Point Arithmetic

2.2.1 Addition

- Module name: addfloat
- Corresponding C operator: +
- Data inputs: A, B
- Data output: R

The addfloat operator computes $R = A + B$.

2.2.2 Subtraction

- Module name: subfloat
- Corresponding C operator: -
- Data inputs: A, B
- Data output: R

The subfloat operator computes $R = A - B$.

2.2.3 Multiplication

- Module name: mulfloat
- Corresponding C operator: $*$
- Data inputs: A, B

- Data output: R

The mulfloat operator computes $R = A * B$.

2.2.4 Division

- Module name: divfloat
- Corresponding C operator: /
- Data inputs: A, B
- Data output: R

The divfloat operator computes $R = A/B$.

2.3 Logical Operators

2.3.1 Negation

- Module name: lognot
- Corresponding C operator: !
- Data inputs: A
- Data output: R

The lognot operator computes $R =!A$, i.e., $R = 1$ if $A = 0$, and $R = 0$ if $A \neq 0$.

2.3.2 Logical And

- Module name: logand
- Corresponding C operator: &&
- Data inputs: A, B
- Data output: R

The logand operator assigns $R = 1$ if $A \neq 0$ and $B \neq 0$, $R = 0$ otherwise.

2.3.3 Logical Or

- Module name: logor
- Corresponding C operator: ||
- Data inputs: A, B
- Data output: R

The logor operator assigns $R = 1$ if $A \neq 0$ or $B \neq 0$, $R = 0$ otherwise.

2.4 Bitwise Operators

2.4.1 Complement

- Module name: bitnot
- Corresponding C operator: ~
- Data inputs: A
- Data output: R

For each bit A_i of A, the lognot operator assigns the associated bit $R_i = 1$ if $A_i = 0$, and sets $R_i = 0$ if $A_i = 1$.

2.4.2 Bitwise And

- Module name: bitand
- Corresponding C operator: &
- Data inputs: A, B
- Data output: R

For each two corresponding bits A_i of A and B_i of B, the bitand operator assigns the resulting bit $R_i = 1$, if $A_i = 1$ and $B_i = 1$; R_i is assigned 0 otherwise.

2.4.3 Bitwise Or

- Module name: bitor
- Corresponding C operator: |
- Data inputs: A, B
- Data output: R

For each two corresponding bits A_i of A and B_i of B, the bitor operator assigns the resulting bit $R_i = 1$, if $A_i = 1$ or $B_i = 1$; R_i is assigned 0 otherwise.

2.4.4 Bitwise Xor

- Module name: bitxor
- Corresponding C operator: ^
- Data inputs: A, B
- Data output: R

For each two corresponding bits A_i of A and B_i of B, the bitxor operator assigns the resulting bit $R_i = 1$, if $A_i = 1$ and $B_i = 0$, or $A_i = 0$ and $B_i = 1$; R_i is assigned 0 otherwise.

2.5 Shifts

2.5.1 Constant Left Shift

- Module name: lsl
- Corresponding C operator: <<
- Data inputs: A, B, with B = const
- Data output: R

The lsl operator computes $R = A * 2^B$. We destinguish between constant and variable shifts, because the difference in area consumption in the hardware implementation is significant: a constant shift is just a re-wiring, but a variable shift (see below) needs a barrel shifter instance (or sth. equivalent).

2.5.2 Constant Right Shift

- Module name: lsr
- Corresponding C operator: >>
- Data inputs: A, B, with B = const
- Data output: R

The lsr operator computes $R = A/2^B$, assigning R the integer part of the division result.

2.5.3 Variable Left Shift

- Module name: vsl
- Corresponding C operator: <<
- Data inputs: A, B
- Data output: R

The vsl operator computes $R = A * 2^B$, with B not necessarily being a constant.

2.5.4 Variable Right Shift

- Module name: vsr
- Corresponding C operator: >>
- Data inputs: A, B
- Data output: R

The vsr operator computes $R = A/2^B$, assigning R the integer part of the division result, with B not necessarily being a constant.

2.6 Integer Comparison

2.6.1 Less Than

- Module name: cmplt
- Corresponding C operator: <

- Data inputs: A, B
- Data output: R

The cmplt operator computes $R = A < B$, i.e., it sets $R = 1$ if $A < B$; $R = 0$ otherwise.

2.6.2 Greater Than

- Module name: cmpgt
- Corresponding C operator: $>$
- Data inputs: A, B
- Data output: R

The cmpgt operator computes $R = A > B$, i.e., it sets $R = 1$ if $A > B$; $R = 0$ otherwise.

2.6.3 Less Than Or Equal

- Module name: cmplteq
- Corresponding C operator: $\leq$
- Data inputs: A, B
- Data output: R

The cmplteq operator computes $R = A \leq B$, i.e., it sets $R = 1$ if $A \leq B$; $R = 0$ otherwise.

2.6.4 Greater Than Or Equal

- Module name: cmpgteq
- Corresponding C operator: $\geq$
- Data inputs: A, B
- Data output: R

The cmpgteq operator computes $R = A \geq B$, i.e., it sets $R = 1$ if $A \geq B$; $R = 0$ otherwise.

2.6.5 Equality

- Module name: cmpeq
- Corresponding C operator: $==$
- Data inputs: A, B
- Data output: R

The cmpeq operator computes $R = A == B$, i.e., it sets $R = 1$ if $A = B$; $R = 0$ otherwise.

2.6.6 Inequality

- Module name: cmpneq
- Corresponding C operator: $! =$
- Data inputs: A, B
- Data output: R

The cmpneq operator computes $R = A! = B$, i.e., it sets $R = 1$ if $A \neq B$; $R = 0$ otherwise.

2.7 Floating Point Comparison

2.7.1 Less Than

- Module name: cmplt_float
- Corresponding C operator: $<$
- Data inputs: A, B
- Data output: R

The cmplt_float operator computes $R = A < B$, i.e., it sets $R = 1$ if $A < B$; $R = 0$ otherwise.

2.7.2 Greater Than

- Module name: cmpgt_float
- Corresponding C operator: $>$
- Data inputs: A, B
- Data output: R

The cmpgt_float operator computes $R = A > B$, i.e., it sets $R = 1$ if $A > B$; $R = 0$ otherwise.

2.7.3 Less Than Or Equal

- Module name: cmplteq_float
- Corresponding C operator: $\leq$
- Data inputs: A, B
- Data output: R

The cmplteq_float operator computes $R = A \leq B$, i.e., it sets $R = 1$ if $A \leq B$; $R = 0$ otherwise.

2.7.4 Greater Than Or Equal

- Module name: cmpgteq_float
- Corresponding C operator: $\geq$
- Data inputs: A, B
- Data output: R

The cmpgteq_float operator computes $R = A \geq B$, i.e., it sets $R = 1$ if $A \geq B$; $R = 0$ otherwise.

2.7.5 Equality

- Module name: cmpeq_float
- Corresponding C operator: $==$
- Data inputs: A, B

- Data output: R

The cmpeq_float operator computes $R = A == B$, i.e., it sets $R = 1$ if $A = B$; $R = 0$ otherwise.

2.7.6 Inequality

- Module name: cmpneq_float
- Corresponding C operator: $! =$
- Data inputs: A, B
- Data output: R

The cmpneq_float operator computes $R = A! = B$, i.e., it sets $R = 1$ if $A \neq B$; $R = 0$ otherwise.

2.8 Casts

2.8.1 Bit Selection

- Module name: bitsel
- Corresponding C operator: `(data_type)`
- Data inputs: A
- Data output: R

The bitsel operator assigns R a bit range of A, padding R's MSBs if necessary. Signedness is also handled correctly, i.e., preserving the sign, or conversions between signed and unsigned.

Thus, bitsel can be used to implement all integer-related casts, e.g., from `unsigned char` to `signed int`.

2.8.2 Integer to Floating Point

- Module name: int_to_float
- Corresponding C operator: `(data_type)`
- Data inputs: A
- Data output: R

The int_to_float operator consumes an integer value A and converts it to the IEEE 754 floating point representation which is output via the R port.

2.8.3 Floating Point to Integer

- Module name: float_to_int
- Corresponding C operator: `(data_type)`
- Data inputs: A
- Data output: R

The float_to_int operator consumes an IEEE 754 floating point value A and converts it to the integer representation which is output via the R port.

2.9 CPU I/O

2.9.1 Input Register

- Module name: inreg
- Corresponding C operator: n/a
- Data inputs: A
- Data output: R
- Proprietary inputs: IOA, IOSEL

The inreg operator implements a proprietary, but very simple interface between hardware accelerator and a CPU, which we use in our COMRADE compiler. For details, refer to Section 5.

2.9.2 Output Register

- Module name: outreg
- Corresponding C operator: n/a
- Data inputs: A
- Data output: R
- Proprietary inputs: IOA, IOSEL

- Proprietary outputs: IODOUT

Analogously to inreg, the outreg module stores an input value and provides it to the CPU. For details, refer to 5.

2.9.3 IRQ Output Register

- Module name: irqreg
- Corresponding C operator: n/a
- Data inputs: A
- Data output: R
- Proprietary inputs: IOA, IOSEL
- Proprietary outputs: IODOUT, IRQ

An irqreg has the functionality of an outreg, but as soon as the module is activated, an additional IRQ output is set. In COMRADE, we use the IRQ to signal the CPU that the current hardware kernel has finished its computation. Details: see Section 5.

2.10 Memory Accesses

2.10.1 Memory Read

- Module name: memread
- Corresponding C operator: $*$, []
- Data inputs: A
- Data output: R
- Proprietary inputs: CACHE_READ, CACHE_STALL
- Proprietary outputs: CACHE_ADDR, CACHE_WIDTH, CACHE_OE

The memread operator implements single word read accesses to memories. The address in consumed via the A port, and the result (i.e., the data word read from memory) output via R. There is a simple, proprietary memory interface which is very similar to the Xilinx BRAM interface, extended by a stall signal for cached accesses. See Section 5 for details.

2.10.2 Memory Write

- Module name: memwrite
- Corresponding C operator: $*$, []
- Data inputs: A, B
- Data output: -
- Proprietary inputs: CACHE_STALL
- Proprietary outputs: CACHE_WRITE, CACHE_ADDR, CACHE_WIDTH, CACHE_WE

In analogy to memread, the memwrite operator implements single word write accesses to memories. The address in consumed via the A port, data via B. Note that there is no data output port. The memory interface works in analogy to memread (see Section 5 for details).

2.11 Other Data Flow Modules

The following modules are used for data flow networks, without having a direct correspondence in C.

2.11.1 Multiplexer

- Module name: mux
- Corresponding C operator: n/a
- Data inputs: A
- Select input: B
- Data output: R

The mux operator reads several concatenated signals via A and outputs one of them via R. The select input B determines which signal (i.e., which bit range of input A) is propagated to R.

2.11.2 No Operation

- Module name: nop
- Corresponding C operator: n/a

- Data inputs: A
- Data output: R

The nop operator does not perform any operation; it just forwards A to R.

2.11.3 Constant

- Module name: const
- Corresponding C operator: n/a
- Data inputs: -
- Data output: R

The const operator outputs a constant value via R.

3 Generic Module Layout

Fig. 1 gives an overview over some of the internal structures of a Modlib module wrapper. The intrinsic operation (e.g., an RTL operator or embedded IP block) consumes the incoming operands. Its output can optionally be buffered in a transparent output queue to decouple the execution of the module from stalled data successors. Also optionally, the output can be delayed for a fixed number of cycles in a configurable shift register. This can be used both for balancing pipeline paths with unequal latency as well as enabling higher clock-frequency operation by retiming in the logic synthesis tool. The token queue is required in a dynamic scheduling scheme only. It buffers activate and cancel tokens in the order they were received. Thereby, the data flow is independent of the token flow and token delivering modules can proceed to provide the next token without having to wait for all receivers to have them processed. Again, if these features are not requested by the user (the compiler back-end), they will be optimized away in synthesis.

4 Parameters

The parameters supported by the Modlib modules can roughly be divided into three groups. The first affects the operator function directly and encompasses the bit-width, or signedness of data or the number of inputs and select scheme (one-hot or encoded) of a multiplexer. The second group deals with the buffering of data, allowing the insertion of transparent queues or fixed-depth shift registers. The third group controls token handling and can indicate static or dynamic CTs and predication. In all cases,

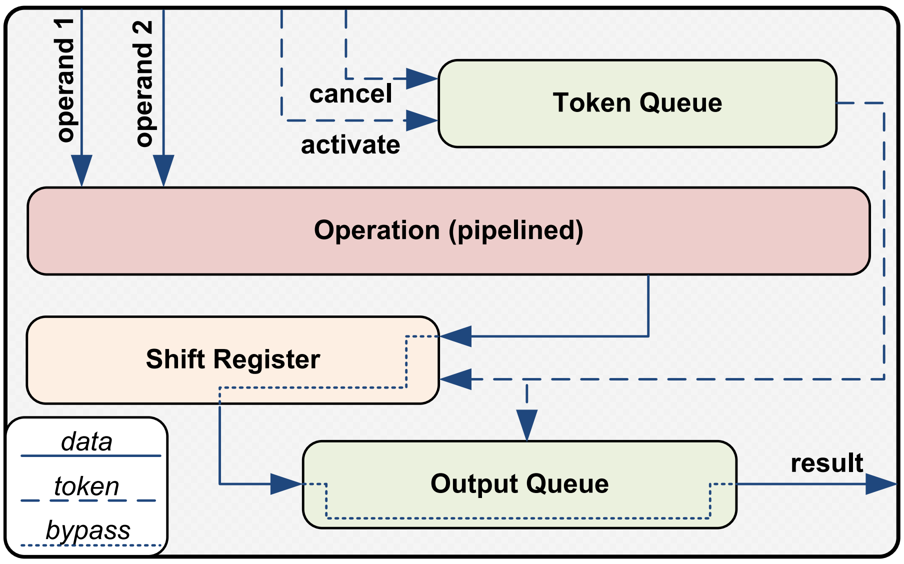


Figure 1: Generic Module Layout.

if a feature is not required, it will not be generated in HW. Thus, Modlib is also applicable for building simple statically scheduled datapaths.

4.1 Operator function

This subsection contains parameters that affect the operator function directly.

4.1.1 General parameters

bit-width These parameters define the bit-width of all data inputs and data outputs.

- WA: bit-width of data input A
- WB: bit-width of data input B
- WR: bit-width of data output R

SIGN This parameter defines the signedness of the operation.

- 0: the operation is a unsigned operation
- 1: the operation is a signed operation

AreaNSpeed This parmeter optimizes a module for timing or area respectively.

- 0: timing optimization, instantiate IP with small initiation interval
- 1: area optimization, instantiate IP with low area consumption

Multiplexer parameters The following parameters are solely required for the multiplexer module.

NIN This parameter defines the number of the multiplexer′s inputs.

- 2: the multiplexer has 2 inputs
- 3: the multiplexer has 3 inputs
- n: the multiplexer has n inputs

ONEHOT This parameter defines the multiplexer select-scheme.

- 0: select is binary coded
- 1: select is one-hot coded

4.1.2 Register parameters

The following parameters are solely required for the Input Register (inreg), the Output Register (outreg) and the IRQ Register (irqreg) modules.

IOADDR This parameter defines the unique ID, which is used to address the particular register.

- n: CPU access to this register via the following address HW_KERNEL_ADDR_OFFFSET+(n«2), where the shift by 2 is required for word access.

IOAWIDTH This parameter defines the bit-width of the I/O address.

- n: IOAddr has n bits

4.1.3 Memory Access parameters

The following parameters apply for Memory Read (memread) and Memory Write (memwrite) modules only.

ADDRWIDTH This parameter defines the width of address bus.

DATAWIDTH This parameter defines the width of data bus.

WIDTH This parameter defines the data width of a single access.

- 0: 32-bit access
- 1: 8-bit access
- 2: 16-bit access

PORTNUM This parameter defines the number of the memory port starting from zero.

4.2 Data Buffering Parameters

These parameters affect the latency and the buffer capability of a module.

4.2.1 DEPTH

This parameter defines the depth of the intrinsic shift register.

- Register Retiming: insertion enables the synthesis tool to optimize register to register delay for complex computations
- Pipeline-Balancing: insertion can delay branches to match with other data pathes

4.2.2 QDEPTH

This parameter defines the amount of computational result, that can be buffered in the output queue at a time. In a dynamic scheduling scheme the buffer avoids computation stalls when data is not allowed to enter the next stage and that way enables pipeline-balancing for operators with variable latency.

4.3 Token Handling Parameters

These parameters affect the scheduling model and how tokens are treated.

4.3.1 START_CTRL_IN

This parameter defines whether activate (AT) and cancel tokens (CT) may enter a particular module. ATs are provided via StartCtrl/Ack, CTs are provided via Cancel/Ack (discussed in the next section).

- `1` predicated operation (token queue insertion)
- `0` non-predicated operation

4.3.2 STATIC_CT

This parameter defines whether cancel tokens are passed to data predecessors if the particular computation has not reached the module yet.

- `1` static CTs, CTs do not leave the module
- `0` dynamic CTs, CTs are passed backwards along the data path

4.4 Unrequired Parameters

The following parameters are required in the COMRADE architecture only.

- PSEUDO_AT: set to zero
- NoCT: set to zero

5 I/O Signals

Fig. 2 shows the generic I/O signals available for each Modlib module[4]. These include customary, data i/o, handshaking, and target architecture specific signals, detailed in the following[5].

[4] As an exception, the memread module has no second data input port B, and the memwrite module has no data output port R.

[5] There are some other signals (not described here), which are used internally for current research. However, in Section 5.7, we provide information how to connect these signals in order to get Modlib working correctly.

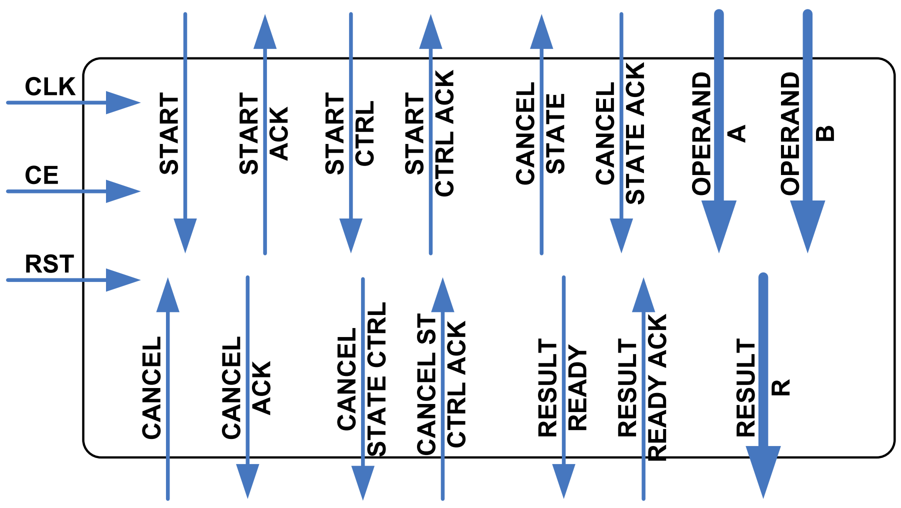


Figure 2: Generic I/O signals.

5.1 Customary Signals

The customary clock, reset and clock enable signals are used by each Modlib module. In static scheduling mode, these are the only control signals for the Modlib modules (besides specific signals such as select for multiplexers or memory access signals for memread/memwrite).

5.1.1 CLK

- Direction: input
- Width: 1

CLK is the central clock signal which must be connected to each Modlib module. Internal registers latch new values on the rising edge of CLK (assuming CE=1, see below).

5.1.2 RESET

- Direction: input
- Width: 1

RESET is the central reset signal which must be connected to each Modlib module. Internal registers are asynchronously reset on a positive edge of RESET.

5.1.3 CE

- Direction: input
- Width: 1

CE is the central clock enable signal which must be connected to each Modlib module. Internal registers latch new values on the rising edge of CLK only if CE=1. Thus, CE has to be set to 1 during operation, independent of the scheduling mode.

5.2 Data I/Os

The customary clock, reset and clock enable signals are used by each Modlib module. In static scheduling mode, these are the only control signals for the Modlib modules (besides specific signals such as select for multiplexers or memory access signals for memread/memwrite).

5.2.1 A

- Direction: input
- Width (standard case): WA
- Width (mux module): WA * NIN

A is the first data input of a Modlib module and has bitwidth WA. Only for the mux module, A reads more than one input (NIN inputs, see Section sec:parameters) and thus has an extended bitwidth.

In static scheduling mode, A is consumed if CE is 1. In dynamic scheduling mode, A is consumed if CE, START and START_RFD are 1.

5.2.2 B

- Direction: input
- Width (standard case): WB
- Width (mux module): NIN

B is the second data input of a Modlib module and has bitwidth WB. Only for the mux module, B represents the select signal instead of a data input. Depending on the parameter ONE_HOT, B has the semantics of an encoded select (ONE_HOT = 0) or a one-hot select (ONE_HOT = 1). In encoded select mode, the mux propagates $A[(B+1)*WA-$

$1 : B * WA]$ to the output. In one-hot mode, it propagates $A[(i+1)*WA-1 : i*WA]$ if $B[i] = 1$. Note that the encoded select mode is currently not supported in dynamic scheduling mode.

In static scheduling mode, B is consumed if CE is 1. In dynamic scheduling mode, B is consumed if CE, START and START_RFD are 1.

5.2.3 R

- Direction: output
- Width: WR

R outputs the result of the module's computation.

5.3 Handshaking Signals

The signals described in this Section are used for handshaking in dynamic scheduling mode. For static scheduling, each input needs to be set to a specific constant value (see below). Each signal comes with an acknowledgement partner signal, e.g., START and START_ACK.

All handshaking signals work combinatorially, i.e., START is acknowledged, if START and START_ACK are both 1 on a rising clock edge. An example is given in Fig. 3: only on edges 2 and 3, the module consumes the inputs. The START on edge 1 marks the same input data valid as the START on edge 2, because there is no acknowledgement on edge 1. In data flow networks consisting of multiple connected Modlib modules, this model achieves the maximum possible throughput (assuming ideal conditions, i.e., balanced pipelines or adequatly sized buffers) for both the static as well as the dynamic scheduling mode.

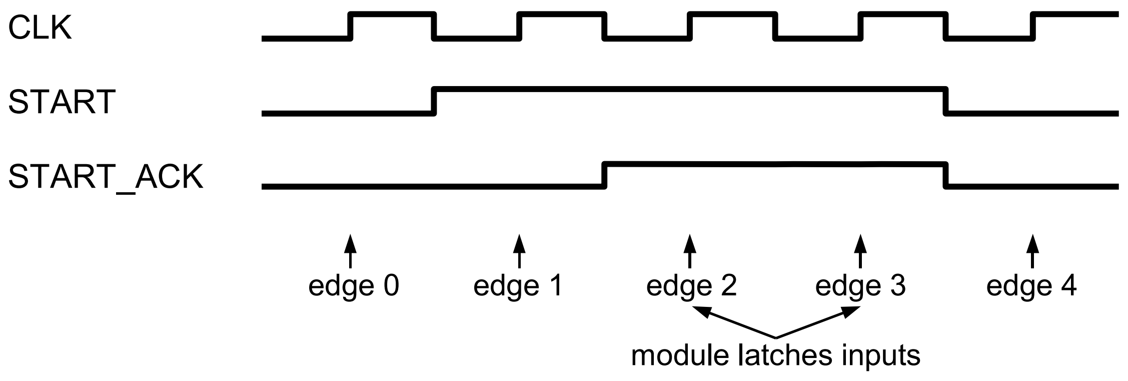


Figure 3: Timing of handshaking signals.

5.3.1 START

- Direction: input
- Width (standard case): 1
- Width (mux case): NIN
- Constant input for static scheduling: 1

If START is 1, the data inputs (A, B) are valid and can be latched by the module. If the module actually *does* latch them, it sets START_ACK to 1.

For multiplexers, START is a multi-bit signal, each bit indicating valid data at exactly one input signal of the A port. START(i) then corresponds to A((i+1)*WA-1:i*WA).

5.3.2 START_ACK

- Direction: output
- Width (standard case): 1
- Width (mux case): NIN

This is the acknowledgement partner for START.

5.3.3 START_CTRL

- Direction: input
- Width: 1
- Constant input for static scheduling: 1

START_CTRL indicates an incoming AT for predicated operators (i.e., parameter START_CTRL_IN is 1). A predicated operator does not output the computed data before the operator has received such an AT.

5.3.4 START_CTRL_ACK

- Direction: output
- Width: 1

This is the acknowledgement partner for START_CTRL.

5.3.5 RESULT_READY

- Direction: output
- Width: 1

If RESULT_READY is 1, the operator has finished the current operation. The value currently assigned to the data output port R represents the result of the computation.

5.3.6 RESULT_READY_ACK

- Direction: input
- Width: 1
- Constant input for static scheduling: 1

This is the acknowledgement partner for RESULT_READY.

5.3.7 CANCEL

- Direction: input
- Width: 1
- Constant input for static scheduling: 0

For predicated operators: If CANCEL is 1, a CT is coming in. If predication is disabled for this operator (i.e., START_CTRL_IN is 0), but the dynamic CT model is used, the CANCEL input is used to receive CTs from cancelled successor operators in the data flow. In both cases (predication or CT from successor), the node handles the CT internally in the same way, except the storage in the token queue, which only exists for predicated execution mode. A CT in a node will discard the next computed result, or (if no computation is running and the dynamic CT model is used) forward the cancel information by setting CANCEL_STATE and CANCEL_STATE_CTRL to 1 (see below).

5.3.8 CANCEL_ACK

- Direction: output
- Width: 1

This is the acknowledgement partner for CANCEL.

5.3.9 CANCEL_STATE

- Direction: output
- Width (standard case): 1
- Width (mux case): 1

Via CANCEL_STATE, CTs are propagated to the predecessor operators in the data flow. If CANCEL_STATE is 1, the module outputs a CT via this port, otherwise it doesn't. As a consequence, CANCEL_STATE is constant 0 if the static CT model is used (the node does never propagate CTs to predecessor modules in a data flow network).

For multiplexers, CANCEL_STATE is a multi-bit signal similar to START. Each bit of CANCEL_STATE is then used to address exactly one data input.

5.3.10 CANCEL_STATE_ACK

- Direction: input
- Width (standard case): 1
- Width (mux case): 1
- Constant input for static scheduling: 0

This is the acknowledgement partner for CANCEL_STATE.

5.3.11 CANCEL_STATE_CTRL

- Direction: output
- Width: 1

CANCEL_STATE_CTRL is used to forward CTs as false predicate to another, predicated, operator. This in used the COMRADE compiler to implement CT-forwarding from cancelled controller operators to their dependants [GäKo08]. If CT-forwarding is not needed (e.g., for implementing CT scheduling on structures without nested conditions), this output can just be ignored.

5.3.12 CANCEL_STATE_CTRL_ACK

- Direction: input
- Width: 1

- Constant input for static scheduling: 0

This is the acknowledgement partner for CANCEL_STATE_CTRL.

5.4 Specific Signals for Multiplexers

While the standard handshaking signals control the data flow, i.e., latching data from A and B for non-mux operators, latching data from A for mux operators, and outputting data via R, multiplexers need a signal to control the input along their select port B. This signal is START_SEL, following the same handshaking scheme anticipated in the previous Section.

A design option would be to require all mux inputs, i.e., the data inputs A and the select input B, at the same time. While this would render START_SEL obsolete (because the existing START handshaking can be used for data and select simultaneusly), a mux would not be able to receive data and select at different points in time, which we prefer for dynamic scheduling. However, in static scheduling mode, START_SEL is unused (as is START).

5.4.1 START_SEL

- Direction: input
- Width: 1
- Constant input for static scheduling: 1

For muxes, this signal notifies a valid value at the select input (port B).

5.4.2 START_SEL_ACK

- Direction: output
- Width: 1

This is the acknowledgement partner for START_SEL.

5.5 Specific Signals for CPU I/O

The inreg, outreg, and irqreg modules provide a simple, proprietary slave mode interface for communication with a CPU, used by our COMRADE compiler. The CPU is the master; the **hardware accelerator** is the slave, consisting of several **hardware kernels**, each of which containing several inregs, outregs, and one irqreg.

Trying to find a reasonable borderline for this report between Modlib and our specific hardware configuration for COMRADE, we give some information about the semantics of the Modlib I/O signals, but do not explain the full CPU/HW communication protocol in detail.

IOSEL is used to address a hardware kernel, while IOA specifies a single inreg, outreg, or irqreg. An inreg receives data via its A port; an outreg and irqreg provide data via the R and IODOUT ports. The IRQ output of the irqreg is connected to an interrupt controller or directly to the CPU to inform the CPU about a certain hardware accelerator state. In COMRADE, we use this to signal the CPU the ending of a hardware kernel computation.

Fig. 4 shows how the COMRADE compiler connects inreg, outreg, and irqreg modules to a CPU. As inregs can only read data, and outregs and irqregs can only write data, we here encode the direction of the access (read / write) in the address.

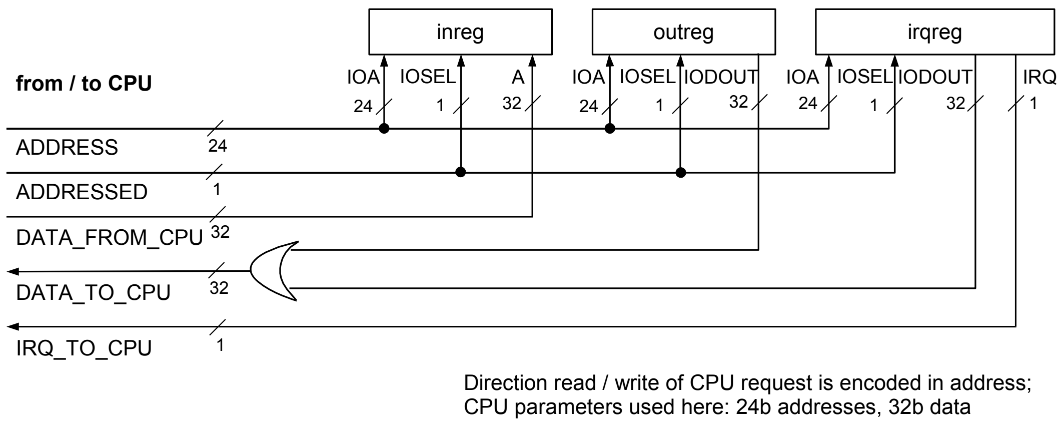


Figure 4: Connecting inreg, outreg, irqreg to a simple CPU interface.

5.5.1 IOSEL

- Direction: input
- Width: 1

If IOSEL is 1, the hardware kernel containing this module is addressed. Otherwise, it is not addressed.

5.5.2 IOA

- Direction: input
- Width: IOAWIDTH (see Section 4)

If IOA matches the module parameter IOADDR (see Section 4), and IOSEL is 1, this module is addressed (otherwise, it it not addressed).

If this module is an inreg, it is addressed, and START and START_ACK are 1, the module latches the value which is present at the A input.

5.5.3 IODOUT

- Direction: output
- Width: WR

outregs and irqregs have an IODOUT output, which they use to provide data for the master. The data currently stored in the module is always output via port R. If the module is addressed (see above), it is output via IODOUT at the same time. Otherwise, IODOUT is 0. This allows the simple or-ing of multiple IODOUTs (multiple outregs) to a common bus connected to the CPU.

5.5.4 IRQ

- Direction: output
- Width: 1

An irqreg has an IRQ output which it assigns 1 if START, START_ACK, and CE are all 1. In COMRADE, we use the IRQ to signal the CPU that the current hardware kernel has finished its computation. Such a technique of course only makes sense in dynamic scheduling mode.

5.6 Specific Signals for Memory Accesses

The memread and memwrite modules provide a simple, proprietary, but Xilinx BRAM-like interface for communication with a memory unit. In the COMRADE context, we use them for cached accesses to the main memory and for local BRAM accesses.

Similar to the CPU/HW communication addressed above, we give some information about the semantics of I/O signals, but do not explain the full memory communication protocol in detail. More details can be found in [LaKo00] and [Lang01].

In Modlib, we support a virtually unlimited number NUM_CACHEPORTS of memory ports, each port being either a cache port or a local BRAM port. Each memread and memwrite instance is actually connected to each port, but accesses only exactly one port (defined by parameter PORTNUM). This simplifies the signal connections for memread and memwrite instances, without imposing additional overhead: As only one port is actually used, the other connections are simply optimized away by the synthesis tool[6].

Note that the CACHE_ prefix in the signals is derived from the first access model (cached memory ports) we have implemented for our COMRADE compiler. Today, we use local BRAM accesses in addition, without altering the signal names in memread and memwrite.

Fig. 5 shows a network connecting a memread and memwrite module to the memory backend we use in COMRADE, configured for a single memory port; Fig. 6 shows the connections for two memory ports. If multiple memory ports are used, port number 0 always refers to the lower bus bits, i.e., port 0 connects to CACHE_OE(0) and CACHE_ADDR(23:0), while port 1 connects to CACHE_OE(1) and CACHE_ADDR(47:24), and so on.

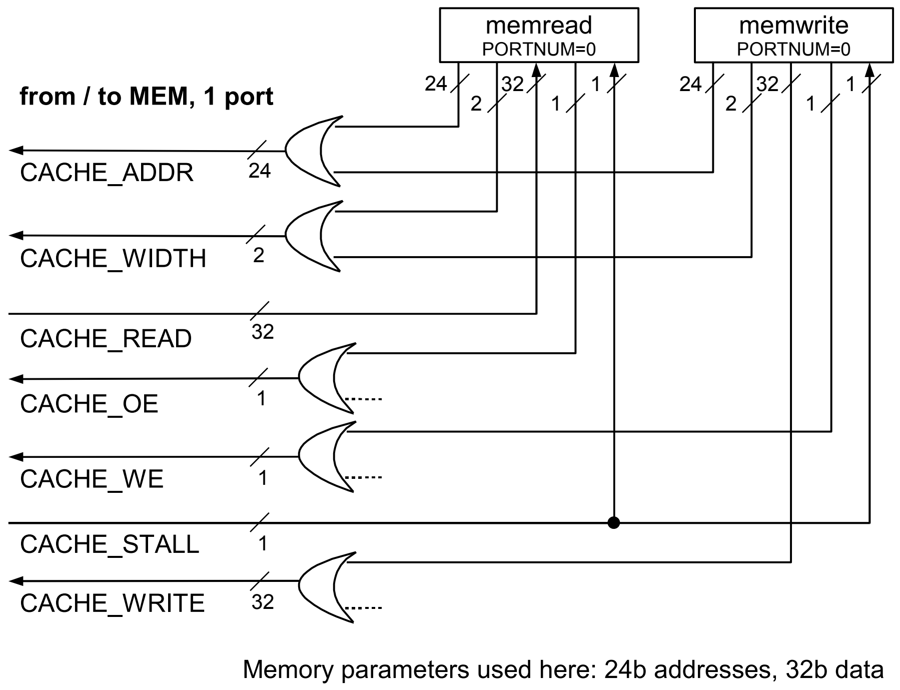
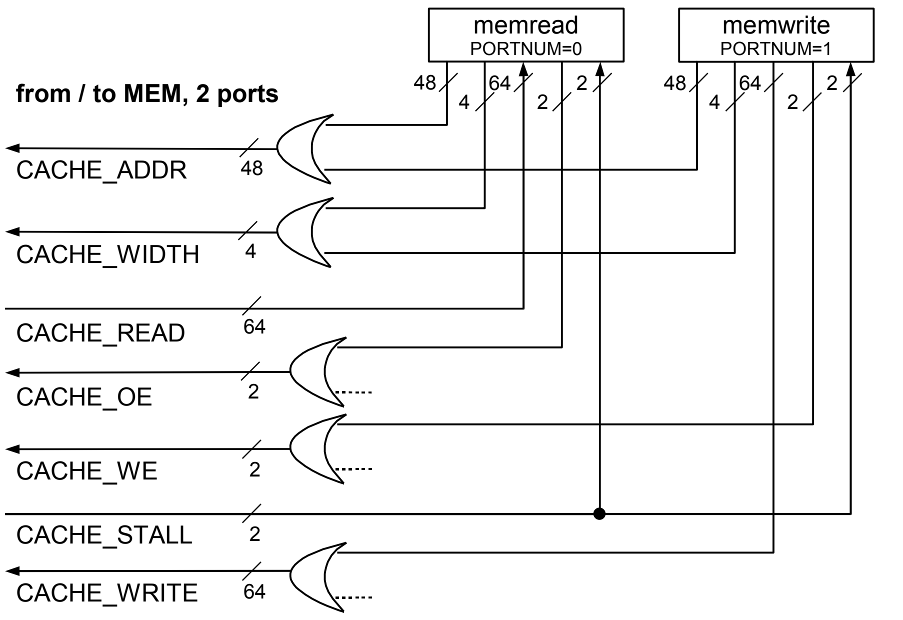


Figure 5: Connecting memread and memwrite to the COMRADE memory backend, using 1 memory port.

5.6.1 CACHE_ADDR

- Direction: output
- Width: NUM_CACHEPORTS * ADDR_WIDTH

[6]This has been tested with current versions of Synopsys Synplify, however, it should work for other tools (such as Xilinx XST), too.



Memory parameters used here: 24b addresses, 32b data.

Multi-port buses are fully connected to each memread/memwrite module; during synthesis, all connections but the actually selected bitrange (PORTNUM) are optimized away.

Port 0 = lower bus bits, e.g.: CACHE_ADDR[23:0] is associated to port 0, while CACHE_ADDR[47:24] refers to port 1. This scales to an arbitrary number of ports.

Figure 6: Connecting memread and memwrite to the COMRADE memory backend, using 2 memory ports.

CACHE_ADDR contains the address for the current memory access.

5.6.2 CACHE_WIDTH

- Direction: output
- Width: NUM_CACHEPORTS * 2

CACHE_WIDTH controls the bitwidth of an access. E.g., if 0x0000FFFF is stored in memory, it really matters if we perform a 32 bit write (storing 0x0000FFFF) or a 16 bit write (storing 0xFFFF without overwriting the upper 2 bytes). If CACHE_WIDTH is 0, a 32 bit wide access is performed. If it is 1, an 8 bit wide access is performed; if it is 2, a 16 bit wide access is performed.

5.6.3 CACHE_READ

- Direction: input
- Width: NUM_CACHEPORTS * DATA_WIDTH

CACHE_READ contains the data read from memory.

5.6.4 CACHE_WRITE

- Direction: output
- Width: NUM_CACHEPORTS * DATA_WIDTH

CACHE_WRITE contains the data written to the memory.

5.6.5 CACHE_OE

- Direction: output
- Width: NUM_CACHEPORTS

If CACHE_OE (cache output enable) is 1, the memread module actually tries to perform a read access to the memory.

5.6.6 CACHE_WE

- Direction: output
- Width: NUM_CACHEPORTS

If CACHE_WE (cache write enable) is 1, the memwrite module actually tries to perform a write access to the memory.

5.6.7 CACHE_STALL

- Direction: output
- Width: NUM_CACHEPORTS

If CACHE_STALL is 1, the requested access can currently not be performed, because the memory system is busy. The reason for this can be a cache stall (i.e., a cache line is currently loaded from or written to the main memory), if the addressed memory port is cached. The signals for the access request have to be held by the module until CACHE_STALL switches back to 0.

5.7 Other Internally Used Signals

For our research, we are performing experiments exceeding the current Modlib state. In this context, there are some I/Os undocumented so far. However, to use Modlib, some of the undocumented inputs need to be connected to specific constants as shown below.

5.7.1 Undocumented Inputs

- START_CTRL_PSEUDO: 0
- LOOP_ENDED: 0
- MEM_READY_ACK: 1

5.7.2 Undocumented Outputs

- INITIALIZABLE
- TOKENS_EMPTY
- CURRENT_TOKEN

- CURRENT_TOKEN_PSEUDO
- MEM_READY

6 Special Module Layout

Multiplexers, CPU I/O, and memory access modules deviate from the generic module layout described in Section 3. The following descriptions address the differences.

6.1 Multiplexer

In contrast to all other modules, mux connects multiple input signals to the data input port A. We choose this implementation to support an arbitrary number of inputs: VERILOG requires a fixed number of I/O ports for modules. Thus, while the width of each single input data signal is WA, the A port has bitwidth WA * NIN, where NIN is the number of data input signals.

B is used as select input port and is NIN bits wide. The ONEHOT parameter decides if B is interpreted as one-hot select (ONEHOT=1) or encoded select (ONEHOT=0), cf. Section 5. While both one-hot and encoded are possible in static scheduling mode, the dynamic scheduling mode is limited to one-hot selects.

According to the multi-signal character of the input data port A, the START, START_ACK, CANCEL_STATE, and CANCEL_STATE_ACK handshaking ports are multi-bit ports, each bit controlling exactly one data input signal. In detail, bit i of a handshaking signal controls the data input signal $A[(i+1)*WA-1 : i*WA]$. This allows the module to selectively accept input signals, and to selectively send cancel tokens (CTs) to data predecessor modules in data flow networks, which actually makes sense when dynamic scheduling with probably imbalanced pipelines (due to variable-latency operators) is used.

In dynamic scheduling mode, muxes interpret the select input bits as AT (bit value 1) or CT (bit value 0), pushing each token into a token queue (Fig. 7). There is one token queue per data input signal. Multiple token queues are the basis for the independent acknowledgement (START_ACK[i]) or cancellation (CANCEL_STATE[i]) of the inputs, which again provides the required flexibility for dynamic scheduling. Note that the one-hot select contains at most one 1-bit; the remaining bits are 0, because only one input may be selected. It is also possible that select is 0 on all bits. This is used by our COMRADE compiler to cancel all inputs in a nested condition tree, because an outer condition already has been evaluated to false [GäKo08]. The mux even supports to suppress the sending of CTs to specific inputs. Therefore, its parameter NO_CT is used: If NO_CT[i] is 1, the mux will delete a CT instead of storing it in token queue i.

Note that apart from the token queues, there is a select queue. This ensures the correct order of data values propagated to the data output. Without the select queue, there could be more than one token queue holding an AT at the output (previous CTs already being sent to predecessor modules), without having a mechanism that denotes the *next* input that has to be routed through.

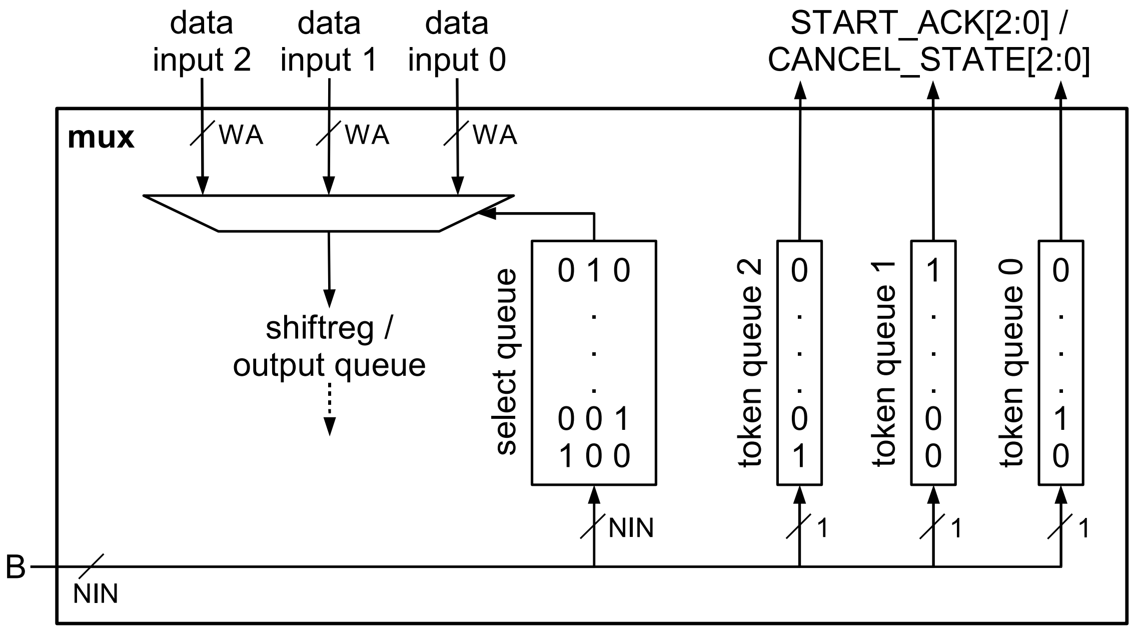


Figure 7: Layout of the mux module concerning select and CT queues.

6.2 CPU I/O

In our COMRADE compiler, we use inreg, outreg, and irqreg modules to establish a connection to a CPU. An inreg receives just one data word (such as an initial variable value or pointer address) before the hardware accelerator executes. Similarly, outregs and irqregs store only one data word and provide it for the CPU, the irqreg generating an IRQ in addition.

As these operations execute seldomly and need no speculation, their layout is much simpler than the ordinary module layout: They contain no token queue, no shift register, and no output queue.

6.3 Memory Accesses

memread and memwrite are designed as non-speculative modules[7]. Therefore, the module's computation (i.e., the memory access) is not executed before there is an

[7]We are currently implementing speculative versions.

activate token (AT) in the token queue. In other words, the input address (and, for memwrite, input data) must be marked valid before the memory access.

7 CoreGen Scripts

For obvious reasons we cannot publish the intrinsic Xilinx IP blocks we instantiate inside some of the modlib modules. However, to make it easier for the user we included .xco files in the Modlib package. The .xco files enable the user to create the IP cores for their own device in very few steps. Our target architecture is a Virtex-5 FX70T, with speedgrade -1 in ff1136 package. The ISE Software version is 11.4.

The target architecture can be changed easily by text substitution via the following sed scripts. The strings starting with 'my_device' need to be replaced with the appropriate device number etc.

```
# sed -i.bak -e 's/SET device = xc5vfx70t/SET device = my_device/'
*.xco

# sed -i.bak -e 's/SET devicefamily = virtex5/SET devicefamily =
my_devicefamily/' *.xco

# sed -i.bak -e 's/SET package = ff1136/SET package =
my_devicepackage/' *.xco

# sed -i.bak -e 's/SET speedgrade = -1/SET speedgrade =
my_devicespeedgrade/' *.xco
```

Afterwards, coregen needs to be invoked with the following command to create the IP Core's netlist and the corresponding verilog blackbox. This has to be done for all IP Cores. The string 'my_ip_core.xco' needs to be replaced by the particula IP core name.

```
# coregen -p 'my_ip_core.xco'
```

References

[BrGa03] BREJ, C. ; GARSIDE, J.: Early Output Logic using Anti-Tokens. In: *Proc. Int. Workshop on Logic Synthesis (IWLS)*, 2003

[Budi03] BUDIU, M.: *Spatial Computation*. Carnegie Mellon University, Pittsburgh, USA, School of Computer Science, Carnegie Mellon University, Diss., Dec. 2003

[GäKo07] GÄDKE, H. ; KOCH, A.: Comrade - A Compiler for Adaptive Computing Systems using a Novel Fast Speculation Technique. In: *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, 2007, S. 503–504

[GäKo08] GÄDKE, Hagen ; KOCH, Andreas: Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens. In: *Proceedings 4th Int. Workshop on Reconfigurable Computing (ARC)* Bd. 4943/2008. Berlin, Heidelberg : Springer, Aug. 2008. – ISBN 978–3–540–78609–2, S. 185–195

[GäSK08] GÄDKE, Hagen ; STOCK, Florian ; KOCH, Andreas: Memory access parallelisation in high-level language compilation for reconfigurable adaptive computers. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, S. 403–408

[GäTK10] GÄDKE-LÜTJENS, H. ; THIELMANN, B. ; KOCH, A.: A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation. In: *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, 2010

[GäTK] GÄDKE-LÜTJENS, Hagen ; THIELMANN, Benjamin ; KOCH, Andreas: *Modlib Download Page*. http://www.esa.cs.tu-darmstadt.de, Section Research/Publications,

[JúCK06] JÚLVEZ, Jorge ; CORTADELLA, Jordi ; KISHINEVSKY, Michael: Performance analysis of concurrent systems with early evaluation. In: *ICCAD'06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–389–1, S. 448–455

[Kasp05] KASPRZYK, Nico: *COMRADE - Ein Hochsprachen-Compiler für adaptive Computersysteme*. Mühlenpfordtstr. 23, 38106 Braunschweig, Abteilung Entwurf integrierter Schaltungen, Technische Universität Braunschweig, Diss., June 2005

[KoKa05] KOCH, Andreas ; KASPRZYK, Nico: High-Level-Language Compilation for Reconfigurable Computers. In: *Proc. Int. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC)*, 2005

[LaKo00] LANGE, Holger ; KOCH, Andreas: Memory Access Schemes for Configurable Processors. In: *Proceedings 10th Int. Workshop on Field-Programmable Logic and Applications (FPL)*. London, UK : Springer-Verlag, Aug. 2000. – ISBN 3–540–67899–9, S. 615–625

[Lang01] LANGE, Holger: *MARC: Ein parametrisiertes Speicherzugriffssystem für adaptive Rechner*, Integrated Circuit Design (E.I.S.), Tech. Univ. Braunschweig, Diplomarbeit, Apr. 2001