Stratix II MAC WYSIWYG Description

Version 2.3
January 4, 2007

By
Altera Corporation

Table of Contents:

1 MAC WYSIWYG primitive

This document describes the WYSIWYG primitive for the Stratix II DSP (or MAC) block. The Stratix II MAC is a superset of the Stratix MAC: all legal Stratix MAC configurations are also legal Stratix II MAC configurations. See the "Stratix MAX WYSIWYG Description" document more information on the Stratix MAC WYSIWYG.

The following is a summary of the new features in the Stratix II MAC, compared to the Stratix MAC:

1. Rounding and Saturation support for Q1.15 format numbers
2. Accumulator is fully loadable with separate load data
3. Loadable input shift registers, by dynamically selecting input source between data and scanin
4. Dynamically switching between three of the modes (18-bit multipliers, 18-bit multiply-accumulators, and 36-bit multipliers).

1.1 The Stratix II MAC Multiplier

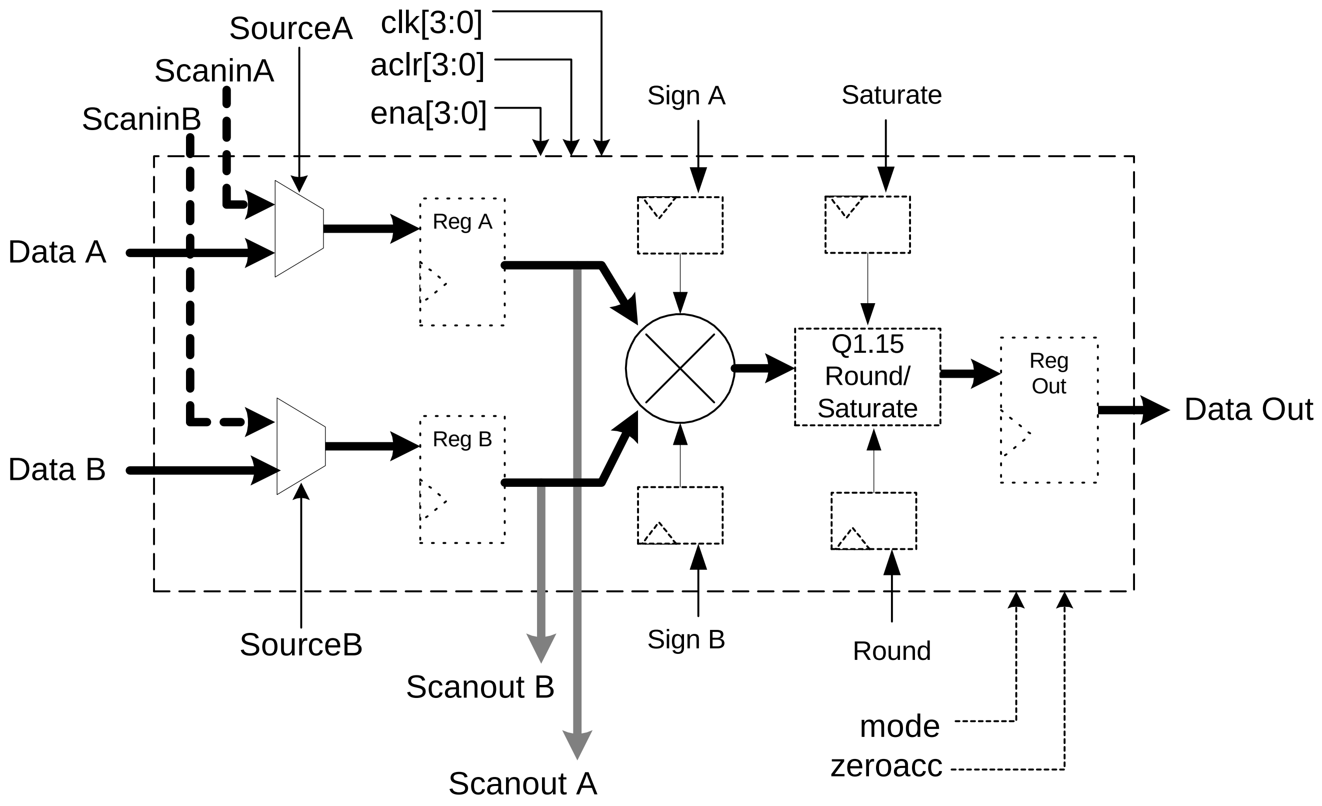

Figure 1: The Stratix II MAC Multiplier

1.1.1 MAC Multiplier Cell Primitives

```
stratixii_mac_mult <mac_mult_name>
(
      .dataa(<data_a source bus>),
      .datab(<data_b source bus>),
      .scanina(<scan input a bus>),
      .scaninb(<scan input b bus>),
      .sourcea(<source selection for data a>),
      .sourceb(<source selection for data b>),
      .signa(<signed/unsigned a source>),
      .signb(<signed/unsigned b source>),
      .round(<enable rounding for 1.15 format>),
      .saturate(<enable saturation for 1.15 format>),
      .clk(<clock source bus>),
      .aclr(<asynchronous clear source bus>),
      .ena(<clock enable source bus>),

      .mode(<dynamic mode control>),
      .zeroacc(<dynamic mode control>),

      .dataout(<data output bus>),
      .scanouta(<scan output bus for data a>),
      .scanoutb(<scan output bus for data b>),

      .observabledataa_regout(<observable bus for data a>),
      .observabledatab_regout(<observable bus for data b>),

      .observablesigna_regout(<observable port for sign a>),
      .observablesignb_regout(<observable port for sign b>),
      .observablesaturate_regout(<observable port for saturate>),
      .observableround_regout(<observable port for round>),
      .observablemode_regout(<observable port for mode>),
      .observablezeroacc_regout(<observable port for zeroacc>),
);
defparam <mac_mult_name>.dataa_width = <dataa width>;
defparam <mac_mult_name>.datab_width = <datab width>;
defparam <mac_mult_name>.dataa_clock = <clk for data_a>;
defparam <mac_mult_name>.datab_clock = <clk for data_b>;
defparam <mac_mult_name>.signa_clock = <clk for sign_a>;
defparam <mac_mult_name>.signb_clock = <clk for sign_b>;
defparam <mac_mult_name>.round_clock = <clk for round>;
defparam <mac_mult_name>.saturate_clock = <clk for saturate>;
defparam <mac_mult_name>.output_clock = <clk for output>;
defparam <mac_mult_name>.dataa_clear = <aclr for data_a>;
defparam <mac_mult_name>.datab_clear = <aclr for data_b>;
defparam <mac_mult_name>.signa_clear = <aclr for sign_a>;
defparam <mac_mult_name>.signb_clear = <aclr for sign_b>;
defparam <mac_mult_name>.round_clear = <aclr for round>;
defparam <mac_mult_name>.saturate_clear = <aclr for saturate>;
defparam <mac_mult_name>.output_clear = <aclr for output>;
defparam <mac_mult_name>.bypass_multiplier = <bypass multiplier>;

defparam <mac_mult_name>.mode_clock = <clk for mode>;
defparam <mac_mult_name>.zeroacc_clock = <clk for zeroacc>;
defparam <mac_mult_name>.mode_clear = <aclr for mode>;
defparam <mac_mult_name>.zeroacc_clear = <aclr for zeroacc>;
```

```
// simulation-only parameters
defparam <mac_mult_name>.signa_internally_grounded = <signa internal GND>;
defparam <mac_mult_name>.signb_internally_grounded = <signb internal GND>;
defparam <mac_mult_name>.dynamic_mode = <dynamic mode indicator>;
defparam <mac_mult_name>.dataout_width = <width of dataout port>;
```

1.1.2 MAC Multiplier Cell Input Ports

<*mac_mult_name*>**:** is the unique identifier for the MAC multiplier. This is any identifier name which is legal for the given description language (e.g. Verilog, VHDL, AHDL, etc.). This field is required.

.**dataa(**<*data_a source bus*>**), .datab(**<*data_b source bus*>**):** are the two input buses for the multiplier. The inputs can be registered by specifying the dataa/b_clock option. Each input can be of a different width, however internally this is implemented by 0 padding the low-order unused bits. The inputs can be driven by the scanouta/b bus of the previous multiplier through dedicated internal routing when not dynamically selecting the source (i.e. sourcea/b is unconnected or GND). However, regular routing can also be used in certain cases as is outlined in the description of the scanouts. If dynamically selecting the source, then this port cannot be driven by scanouta/b since in that case, scanouta/b must drive scanina/b. This port is required.

.scanina(<*scan input a bus*>**), .scaninb(**<*scan input b bus*>**):** are the two input buses along the scan-chain path when doing dynamic switching between a scan-input and a data-input (i.e. implementing a loadable input shift-register). These inputs can only be driven by scanouta or scanoutb of the respectively of the multiplier above it in the scan-chain, and should only be connected when performing dynamic switching of the source between scanina/b and dataa/b. This input has the same width as the dataa/b port. The **sourcea** and **sourceb** signals control which data source to use. This port is required if sourcea/b is used, and is ignored otherwise.

.sourcea(<source selection for *data a*>**), .sourceb(**<source selection for *data b*>**):** are the two source control signals that control the selection between dataa/b and scanina/b when performing dynamic switching between the two inputs (i.e. implementing a loadable input shift register). A value of GND selects the dataa/b input, and a value of VCC selects the scanina/b input. This signal is not required and defaults to GND (i.e. source is dataa/b input) if left unconnected.

.signa(<*signed/unsigned a source*>***)*****, .signb(**<*signed/unsigned b source*>***)*****:** designates the two sign signals for each input of the multiplier. High implies the input is a signed integer (i.e. MSB bit is used as sign bit), low implies the input is an unsigned integer. Each input can have a different signed/unsigned value since the multiplier will take this into account. This can be set to a constant value by tying the input to VCC or GND. The signal can also be registered by specifying the signa/b_clock option. This port is not required and defaults to VCC if left unconnected.

.round(<*round signal source*>***)*****:** designates the signal to enable/disable rounding in the multiplier assuming Q1.15 format numbers are being used. Only the upper 16-bits of either operand can be used when this signal is asserted since only Q1.15 format numbers are supported (i.e. data width must be < 16-bits). This signal can be registered by specifying the round_clock option. VCC implies that rounding is enabled. This port is not required and defaults to GND (i.e. disabled) if left unconnected.

.saturate(<*saturation signal source*>***)*****:** designates the signal to enable/disable saturation in the multiplier assuming Q1.15 format numbers are being used. Only the upper 16-bits of either operand can be used when this signal is asserted since only Q1.15 format numbers are supported (i.e. data width must be < 16-bits). This signal can be registered by specifying the saturate_clock option. VCC implies that saturation is enabled. This port is not required and defaults to GND (i.e. disabled) if left unconnected.

.clk(<*clock source bus*>**):** designates the bus of four clock inputs that drive the multiplier. Every register bank in the multiplier can independently choose which one of these four clocks can drive the register. However, every clock can only be enabled by the corresponding clock enable signal of the same index – so there can only be a maximum of four clk/ena pairs. This port must be connected if it is specified to be used by one of the parameters, otherwise it must not be connected.

.aclr(<*aclr source bus*>**):** designates the bus of four asynchronous clear inputs that drive the multiplier. Every register bank in the multiplier can independently choose which one of these four signals can drive the register. Unlike the clock enable signal, the aclr signal is not tied to any particular clock (e.g. can mix aclr[0] with clk[3]). This port must be connected if it is specified to be used by one of the parameters, otherwise it must not be connected.

.ena(<*clock enable source bus*>**):** designates the bus of four clock enable inputs that drive the multiplier. Each one of the four clock signals is enabled by the corresponding ena signal. This means that there can be a maximum of four pairs of clk/ce signals and not the sixteen combinations possible if they were completely independent. The reason for this is because the clock enable signal enables and disables the clock signal at the entry to the MAC block, and not at the register. This port must be connected if it is specified to be used by one of the parameters, otherwise it must not be connected.

.mode(<*dynamic mode control*>***)*****, .zeroacc(**<*dynamic mode control*>***)*****:** designates the signals used to indicate the mode of operation when the MAC block is in dynamic mode. See later for details on dynamic mode and the use of these signals. This signal cannot be connected if the corresponding MAC_OUT is not in "dynamic" mode. If left unconnected in dynamic mode, it defaults to GND.

1.1.3 MAC Multiplier Cell Output Ports

.dataout(<*data output bus*>**)**: the output bus of the multiplier (could be registered if using output registers). Note that this bus can only feed the data inputs of a mac_out atom through dedicated internal routing – it is not visible external to the MAC. When saturation is enabled, bit 0 of the 36-bit output bus is the saturation overflow signal – if the width is less than 36-bits, then the saturation overflow signal is not available. This output must be connected – it cannot be left unconnected. The outputs can be registered by specifying the output_clock option.

.scanouta(<*scan output a bus*>**), .scanoutb(**<*scan output b bus*>**):** the scan outputs from the corresponding inputs (basically the output of the input registers if the inputs are registered, or just the inputs themselves if they are not). This bus can use internal routing to drive the dataa/b bus or the scanina/b bus of the next adjacent multiplier, or it can drive external routing directly. Note that only the last multiplier in a MAC can drive external routing, which places some restrictions on the MAC when used as will be described later. Note also that this output is useless if configuring a 36-bit multiplier. This output can be left unconnected if not used.

.observabledataa_regout(<*observable bus*>***)*****, .observabledatab_regout(**<*observable bus*>***)*****:** assigns names for the observable bus for the data inputs. The observable bus will only be created if its corresponding data input bus is registered. These cannot be connected to other routing and are used for simulation and timing purposes only. By connecting wires to the observable bus, the user is able to rename the observable bus to the name of the connected wires.

.observablesigna_regout(<*observable port*>***)*****, .observablesignb_regout(**<*observable port*>**),**
.**observablesaturate_regout(**<*observable port*>**) , .observableround_regout(**<*observable port*>**),**

.observablemode_regout(<*observable port*>**) , .observablezeroacc_regout(**<*observable port*>**):** assigns a name for the observable port for the input signal. The observable port will only be created if its corresponding port is registered. These cannot be connected to other routing and are used for simulation and timing purposes only. By connecting wires to the observable ports, the user is able to rename the observable ports to the name of the connected wires. **This feature has not been implemented yet for these ports.**

1.1.4 MAC Multiplier Cell Modes

<*dataa width*>, <*datab width*> is an integer in the range *{1, 2, …, 18}*. It specifies the width of each operand of the multiplier, as well as the width of the corresponding scanout bus. The combination of the dataa width and the datab width implies the output width: the output width is the sum of both input widths (e.g. for dataa_width = 13 and datab_width = 15, dataout_width = 28). Internally there are only 9-bit and 18-bit multipliers, so multipliers of smaller width are implemented by tying the unused low-order bits to 0 (not the high-order bits since the MSB bit is still needed when doing signed multiplication). To implement a 36-bit multiplier, four 18-bit multipliers are used as will be described later. *This field is required.*

<*clk for …*> is one of *{none, 0, 1, 2, 3}*. Specifies which clock signal, if any, drives the given register, where the number indicates which clk signal (from clk[0] to clk[3]) should be used. If "none" is specified, then the corresponding signal is not registered. Otherwise, the corresponding signal will be registered and will be driven by the indicated clock, which must be connected. Additionally, this parameter also specifies which clock enable signal will be used. This forces the use of ena[2] if clk[2] is used for example. If the clock enable signal is not used, it must be tied to VCC. This field is not required and defaults to "none" if unspecified.

<*aclr for …*> is one of *{none, 0, 1, 2, 3}*. Specifies which asynchronous clear signal, if any, drives the given register, where the number indicates which aclr signal (from aclr[0] to aclr[3]) should be used. If the option is not "none", then the register's clk parameter cannot be "none", and the corresponding signal's register will be cleared by the given aclr signal. However, if the option is "none", then the register's clk parameter must also be "none", and the corresponding signal is not registered. To disable the aclr, this signal should be tied to GND. This field is required if a clk is specified, and must be "none" if a clk is not specified.

<*bypass multiplier*> is one of *{yes, no}*. It specifies whether the multiplier portion of the logic should be bypassed. For the majority of cases, *no* is used and simply represents the case where the MAC_MULT is performing regular multiplication. A value of *yes* is required for the MAC_MULT used to load data for the accumulator, which feeds Data A on the MAC_OUT in accumulator mode, and will not be performing multiplication. If in dynamic mode, then the multiplier operation changes depending on the value of the control signals "mode" and "zeroacc" since the multiplier will only be bypassed when in accumulator mode – but this parameter should still be set to *yes* for the MAC_MULT being used to load the accumulator even in dynamic mode. An error will be issued if the parameter is specified to a value that is not compatible with the operation mode. This parameter defaults to *no* if not specified.

Simulation Only Parameters:

The following parameters are only used for simulation purposes, and will be ignored by the compiler.

<*signa internal GND*>, <*signb internal GND*> is one of *{true, false}*. It specifies whether the sign control signal is internally grounded for this multiplier, effectively overriding the original sign control

signals. Normally, this parameter has a value of “false”, meaning the sign control signals determine the sign representation for the given operand. However, for 36-bit multiplier mode, the lower 18-bits of each 36-bit operand is always considered to be unsigned regardless of the sign specified for the entire 36-bit operand. As a result, the lower 18-bit operands will have this parameter set to true. If in dynamic mode, then the internal grounding changes depending on the value of the control signals “mode” and “zeroacc” since the multiplier can only be internally grounded when in 36-bit multiplier mode – but this parameter should still be set on each MAC_MULT assuming 36-bit multiplier mode, and will be ignored by the simulation models when the dynamic mode is not 36-bit multiplier mode.

<*dynamic mode indicator*> is one of *{yes, no}*. It specifies whether the MAC that the multiplier is part of is in dynamic mode – a value of *yes* indicates that it is in dynamic mode. This parameter has a default value of *no*, and must be specified to *yes* if the MAC is in dynamic mode. In other modes, this parameter can be left unspecified, or can be specified to *no*.

<*width of dataout port*> specifies the width of the dataout port. This is redundant information since the width is derived from the input port widths, but is needed for sim models that cannot do the calculation.

1.1.5 MAC Multiplier Cell Polarities and Default Values

All signals are active high. All inputs have programmable inverts.

1.2 Stratix II MAC Output

1.2.1 MAC Output Cell Primitives

```
stratixii_mac_out <mac_out_name>
(
      .dataa(<data_a source bus>),
      .datab(<data_b source bus>),
      .datac(<data_b source bus>),
      .datad(<data_b source bus>),
      .zeroacc(<zero accumulator source>),
      .addnsub0(<add or subtract 0 source>),
      .addnsub1(<add or subtract 1 source>),
      .round0(<enable rounding for 1.15 format (add0/accum)>),
      .round1(<enable rounding for 1.15 format (add1)>),
      .saturate(<enable saturation for 1.15 format>),
      .multabsaturate(<enable mult a & b saturation for 1.15 format>),
      .multcdsaturate(<enable mult c & d saturation for 1.15 format>),
      .signa(<signed/unsigned a source>),
      .signb(<signed/unsigned b source>),
      .clk(<clock source bus>),
      .aclr(<asynchronous clear source bus>),
      .ena(<clock enable source bus>),

      .mode0(<dynamic mode 0 source>),
      .mode1(<dynamic mode 1 source>),
      .zeroacc1(<2nd zeroacc source in dynamic mode>),
      .saturate1(<2nd saturate source in dynamic mode>),
      .dataout(<data output bus>),
      .accoverflow(<accumulator overflow signal>),

      .observablezeroacc_regout(<observable port for zeroacc>),
      .observableaddnsub0_regout(<observable port for addnsub0>),
      .observableaddnsub1_regout(<observable port for addnsub1>),
      .observableround0_regout(<observable port for round0>),
      .observableround1_regout(<observable port for round1>),
      .observablesaturate_regout(<observable port for saturate>),
      .observablemultabsaturate_regout(<observable port for multabsaturate>),
      .observablemultcdsaturate_regout(<observable port for multcdsaturate>),
      .observablesigna_regout(<observable port for signa>),
      .observablesignb_regout(<observable port for signb>),

      .observablezeroacc_pipeline_regout
                  (<observable pipeline port for zeroacc>),
      .observableaddnsub0_pipeline_regout
                  (<observable pipeline port for addnsub0>),
      .observableaddnsub1_pipeline_regout
                  (<observable pipeline port for addnsub1>),
      .observableround0_pipeline_regout
                  (<observable pipeline port for round0>),
      .observableround1_pipeline_regout
                  (<observable pipeline port for round1>),
      .observablesaturate_pipeline_regout
                  (<observable pipeline port for saturate>),
      .observablemultabsaturate_pipeline_regout
                  (<observable port for multabsaturate>),
      .observablemultcdsaturate_pipeline_regout
                  (<observable port for multcdsaturate>),
      .observablesigna_pipeline_regout(<observable pipeline port for signa>),
      .observablesignb_pipeline_regout(<observable pipeline port for signb>),

      .observablemode0_regout(<observable port for mode0>),
      .observablemode1_regout(<observable port for mode1>),
      .observablezeroacc1_regout(<observable port for zeroacc1>),
      .observablesaturate1_regout(<observable port for saturate1>),

      .observablemode0_pipeline_regout(<observable pipeline port for mode0>),
      .observablemode1_pipeline_regout(<observable pipeline port for mode1>),
      .observablezeroacc1_pipeline_regout
                  (<observable pipeline port for zeroacc1>),
      .observablesaturate1_pipeline_regout
                  (<observable pipeline port for saturate1>),

);
defparam <mac_out_name>.operation_mode = <operation mode>;
defparam <mac_out_name>.dataa_width = <dataa width>;
defparam <mac_out_name>.datab_width = <datab width>;
defparam <mac_out_name>.datac_width = <datac width>;
defparam <mac_out_name>.datad_width = <datad width>;
```

```
defparam <mac_out_name>.zeroacc_clock = <clk for zeroacc>;
defparam <mac_out_name>.addnsub0_clock = <clk for addnsub0>;
defparam <mac_out_name>.addnsub1_clock = <clk for addnsub1>;
defparam <mac_mult_name>.round0_clock = <clk for round0>;
defparam <mac_mult_name>.round1_clock = <clk for round1>;
defparam <mac_mult_name>.saturate_clock = <clk for saturate>;
defparam <mac_mult_name>.multabsaturate_clock = <clk for multabsaturate>;
defparam <mac_mult_name>.multcdsaturate_clock = <clk for multcdsaturate>;
defparam <mac_out_name>.signa_clock = <clk for sign_a>;
defparam <mac_out_name>.signb_clock = <clk for sign_b>;
defparam <mac_out_name>.output_clock = <clk for output>;
defparam <mac_out_name>.zeroacc_clear = <aclr for zeroacc>;
defparam <mac_out_name>.addnsub0_clear = <aclr for addnsub0>;
defparam <mac_out_name>.addnsub1_clear = <aclr for addnsub1>;
defparam <mac_mult_name>.round0_clear = <aclr for round0>;
defparam <mac_mult_name>.round1_clear = <aclr for round1>;
defparam <mac_mult_name>.saturate_clear = <aclr for saturate>;
defparam <mac_mult_name>.multabsaturate_clear = <aclr for multabsaturate>;
defparam <mac_mult_name>.multcdsaturate_clear = <aclr for multcdsaturate>;
defparam <mac_out_name>.signa_clear = <aclr for sign_a>;
defparam <mac_out_name>.signb_clear = <aclr for sign_b>;
defparam <mac_out_name>.output_clear = <aclr for output>;
defparam <mac_out_name>.addnsub0_pipeline_clock = <clk for addnsub0 pipeline>;
defparam <mac_out_name>.addnsub1_pipeline_clock = <clk for addnsub1 pipeline>;
defparam <mac_mult_name>.round0_pipeline_clock = <clk for round0 pipeline>;
defparam <mac_mult_name>.round1_pipeline_clock = <clk for round1 pipeline>;
defparam <mac_mult_name>.saturate_pipeline_clock = <clk for saturate pipeline>;
defparam <mac_mult_name>.multabsaturate_pipeline_clock = <clk for
multabsaturate pipeline>;
defparam <mac_mult_name>.multcdsaturate_pipeline_clock = <clk for
multcdsaturate pipeline>;
defparam <mac_out_name>.zeroacc_pipeline_clock = <clk for zeroacc pipeline>;
defparam <mac_out_name>.signa_pipeline_clock = <clk for sign_a pipeline>;
defparam <mac_out_name>.signb_pipeline_clock = <clk for sign_b pipeline>;
defparam <mac_out_name>.addnsub0_pipeline_clear = <aclr for addnsub0 pipeline>;
defparam <mac_out_name>.addnsub1_pipeline_clear = <aclr for addnsub1 pipeline>;
defparam <mac_mult_name>.round0_pipeline_clear = <aclr for round0 pipeline>;
defparam <mac_mult_name>.round1_pipeline_clear = <aclr for round1 pipeline>;
defparam <mac_mult_name>.saturate_pipeline_clear = <aclr for saturate
pipeline>;
defparam <mac_mult_name>.multabsaturate_pipeline_clear = <aclr for
multabsaturate pipeline>;
defparam <mac_mult_name>.multcdsaturate_pipeline_clear = <aclr for
multcdsaturate pipeline>;
defparam <mac_out_name>.zeroacc_pipeline_clear = <aclr for zeroacc pipeline>;
defparam <mac_out_name>.signa_pipeline_clear = <aclr for sign_a pipeline>;
defparam <mac_out_name>.signb_pipeline_clear = <aclr for sign_b pipeline>;

defparam <mac_out_name>.mode0_clock = <clk for mode0>;
defparam <mac_out_name>.mode1_clock = <clk for mode1>;
defparam <mac_out_name>.zeroacc1_clock = <clk for zeroacc1>;
defparam <mac_mult_name>.saturate1_clock = <clk for saturate1>;
defparam <mac_mult_name>.output1_clock = <clk for output 1>;
defparam <mac_mult_name>.output2_clock = <clk for output 2>;
defparam <mac_mult_name>.output3_clock = <clk for output 3>;
defparam <mac_mult_name>.output4_clock = <clk for output 4>;
defparam <mac_mult_name>.output5_clock = <clk for output 5>;
defparam <mac_mult_name>.output6_clock = <clk for output 6>;
defparam <mac_mult_name>.output7_clock = <clk for output 7>;
defparam <mac_out_name>.mode0_clear = <aclr for mode0>;
defparam <mac_out_name>.mode1_clear = <aclr for mode1>;
defparam <mac_out_name>.zeroacc1_clear = <aclr for zeroacc1>;
defparam <mac_mult_name>.saturate1_clear = <aclr for saturate1>;
defparam <mac_mult_name>.output1_clear = <aclr for output 1>;
defparam <mac_mult_name>.output2_clear = <aclr for output 2>;
defparam <mac_mult_name>.output3_clear = <aclr for output 3>;
defparam <mac_mult_name>.output4_clear = <aclr for output 4>;
defparam <mac_mult_name>.output5_clear = <aclr for output 5>;
defparam <mac_mult_name>.output6_clear = <aclr for output 6>;
defparam <mac_mult_name>.output7_clear = <aclr for output 7>;
defparam <mac_out_name>.mode0_pipeline_clock = <clk for mode0 pipeline>;
defparam <mac_out_name>.mode1_pipeline_clock = <clk for mode1 pipeline>;
defparam <mac_out_name>.zeroacc1_pipeline_clock = <clk for zeroacc1 pipeline>;
defparam <mac_mult_name>.saturate1_pipeline_clock = <clk for saturate1
pipeline>;
defparam <mac_out_name>.mode0_pipeline_clear = <aclr for mode0 pipeline>;
defparam <mac_out_name>.mode1_pipeline_clear = <aclr for mode1 pipeline>;
defparam <mac_out_name>.zeroacc1_pipeline_clear = <aclr for zeroacc1 pipeline>;
defparam <mac_mult_name>.saturate1_pipeline_clear = <aclr for saturate1
pipeline>;
defparam <mac_out_name>.dataa_forced_to_zero = <dataa forced to zero>;
defparam <mac_out_name>.datac_forced_to_zero = <datac forced to zero>;

// simulation-only parameters
defparam <mac_mult_name>.dataout_width = <width of dataout port>;
```

1.2.2 MAC Output Cell Input Ports

<*mac_out_name*>**:** is the unique identifier for the MAC output. This is any identifier name which is legal for the given description language (e.g. Verilog, VHDL, AHDL, etc.). This field is required.

.**dataa(**<*data_a source bus*>**), … .datad(**<*data_d source bus*>**):** are the four input buses for the output block. Depending on the mode, these inputs can take on different meanings. This input can only be driven by the output of a mac_mult atom through dedicated internal routing – it is not visible external to the MAC. This input can only be connected if it is used in the specified operation mode.

.signa(<*signed/unsigned a source*>***)*, .signb(**<*signed/unsigned b source*>***)*:** designates the two sign signals that drive the multiplier. NOTE: they do not correspond to the dataa and datab ports directly – instead they specify the sign of all inputs to the MAC_OUT. If either signal is high, then this implies that all inputs are signed integers. If both signals are low, then all the inputs are unsigned integers. This can be set to a constant value by tying the input to VCC or GND. This signal can be registered by specifying the signa/b_clock option. This signal can also be pipelined (i.e. double registered) by specifying the signa/b_pipeline_clock option. This signal is not required, and defaults to VCC if left unconnected.

.zeroacc(<*zero accumulator source*>**):** the signal which resets the accumulator to zero and also loads the accumulator's upper 36-bits. Only available in accumulator mode or dynamic mode. Note that this is different from clearing the accumulator registers (i.e. output registers) since only the feedback input into the accumulator gets cleared (so that no clock cycles are lost while clearing the accumulator). A value of VCC clears the accumulator feedback, or if the accumulator has the load data port connected (i.e. DATAB), then a value of VCC loads the data into the upper 36-bits of the accumulator. When in dynamic mode, this signal also serves to select the mode – see later for further details. This signal can be registered by specifying the zeroacc_clock option. This signal can also be pipelined (i.e. double registered) by specifying the zeroacc_pipeline_clock option. This signal is not required and defaults to GND if left unconnected.

.addnsub0(<*add or subtract 0 source*>**), .addnsub1(**<*add or subtract 1 source*>**):** specifies whether the first stage adders should be configured as subtractors or adders. addnsub0 controls the top adder, and addnsub1 controls the bottom adder. High means that an addition should be performed, and low means that a subtraction should be performed. This can be set to a constant value by tying the input to VCC or GND. This signal can be registered by specifying the addnsub0/1_clock option. This signal can also be pipelined (i.e. double registered) by specifying the addnsub0/1_pipeline_clock option. This signal is not required and defaults to VCC if left unconnected.

.round0(<*round signal 0 source*>*)*, **.round1(**<*round signal 1 source*>*)***:** designates the signal to enable/disable rounding in the corresponding add/sub or accumulator assuming Q1.15 format numbers are being used. The round0 port controls the top add/sub and the accumulator, and round1 controls the bottom add/sub. Only the upper 16-bits of either operand can be used when this signal is asserted since only Q1.15 format numbers are supported (i.e. data width must be < 32-bits). This signal can be registered by specifying the round0/1_clock option. VCC implies that rounding is enabled. This port is not required and defaults to GND (i.e. disabled) if left unconnected. Note that the signal driving this port can be different from the signal driving the round port on the MAC_MULT WYSIWYG.

.saturate(<*saturation signal source*>*)***:** designates the signal to enable/disable saturation in the accumulator assuming Q1.15 format numbers are being used. This port can only be connected in the accumulator mode. Only the upper 16-bits of either operand can be used when this signal is asserted since only Q1.15 format numbers are supported (i.e. data input width must be < 32-bits). When asserted, the unused lower bits of the dataout[] port function as the saturation overflow signals, as described later. This signal can be registered by specifying the saturate_clock option. VCC implies that saturation is enabled. This port is not required and defaults to GND (i.e. disabled) if left unconnected. Note that the signal driving this port can be different from the signal driving the round port on the MAC_MULT WYSIWYG.

.multabsaturate(<*mult a & b saturation signal source*>**), .multcdsaturate(**<*mult c & d saturation signal source*>**):** designates the signal to enable/disable Q1.15 saturation in the given multipliers, and indicates if the multiplier's saturation overflow signal needs to be sent to the output. This port is physically the same port as the saturate port on the corresponding multiplier, and so must be connected identically as those ports (i.e. same signal source, and same input register configuration). When asserted, the unused lower bits of the dataout[] port function as the saturation overflow signals, as described later. This port is not required and defaults to GND if left unconnected.

.clk(<*clock source bus*>**):** designates the bus of four clock inputs that drive the MAC_OUT cell. Every register bank in the MAC_OUT cell can independently choose which one of these four clocks can drive the register. However, every clock can only be enabled by the corresponding clock enable signal of the same index – so there can only be a maximum of four clk/ena pairs. This signal must be connected if it is specified as used by a parameter.

.aclr(<*aclr source bus*>**):** designates the bus of four asynchronous clear inputs that drive the MAC_OUT cell. Every register bank in the MAC_OUT cell can independently choose which one of these four signals can drive the register. Unlike the clock enable signal, the aclr signal is not tied to any particular clock (e.g. can mix aclr[0] with clk[3]). This signal must be connected if it is specified as used by a parameter.

.ena(<*clock enable source bus*>**):** designates the bus of four clock enable inputs that drive the MAC_OUT cell. Each one of the four clk signals is enabled by the corresponding ena signal. This means that there can be a maximum of four pairs of clk/ce signals and not the sixteen combinations possible if they were completely independent. The reason for this is because the clock enable signal enables and disables the clock signal at the entry to the MAC block, and not at the register. This signal must be connected if it is specified as used by a parameter.

.mode0(<*mode0 signal source*>***),*** **.mode1(**<*mode1 signal source*>***)*** **:** designates the signals that control the mode when operating in dynamic mode. This port can only be connected in the dynamic mode. In conjunction with the zeroacc signals, these signals allow the user to dynamically switch between independent 18-bit multipliers, 18-bit multiply-accumulators, and a 36-bit multiplier. See later for details on how to switch between various modes. This signal can be registered and pipelined as with all control signals. This port is not required and defaults to GND if left unconnected.

.zeroacc1(<2nd *zeroacc source in dynamic mode*>**), .saturate1(**<2nd *saturate source in dynamic mode*>**):** designates the additional signals available when in dynamic mode, and can only be connected when in dynamic mode. These signals have the same functionality as their equivalently named signals that are available for all modes. However, in dynamic mode, these signals are used to control the bottom half of the MAC block, while the original signals are used to control the top half of the MAC block. If left unconnected, these signals default to GND.

.observablezeroacc_regout(<*observable port for zeroacc*>**),**
.observableaddnsub0_regout(<*observable port for addnsub0*>**),**
.observableaddnsub1_regout(<*observable port for addnsub1*>**),**
.observableround0_regout(<*observable port for round0*>**),**
.observableround1_regout(<*observable port for round1*>**),**
.observablesaturate_regout(<*observable port for saturate*>**),**
.observablemultabsaturate_regout(<*observable port for multabsaturate*>**),**
.observablemultcdsaturate_regout(<*observable port for multcdsaturate*>**),**
.observablesigna_regout(<*observable port for signa*>**),**
.observablesignb_regout(<*observable port for signb*>**):** assigns a name for the observable port for the input signal. The observable port will only be created if its corresponding port is registered. These cannot be connected to other routing and are used for simulation and timing purposes only. By connecting wires to the observable ports, the user is able to rename the observable ports to the name of the connected wires. **This feature has not been implemented yet for these ports.**

.observablezeroacc_pipeline_regout(<*observable pipeline port for zeroacc*>**),**
.observableaddnsub0_pipeline_regout(<*observable port for pipeline addnsub0*>**),**
.observableaddnsub1_pipeline_regout(<*observable port for pipeline addnsub1*>**),**
.observableround0_pipeline_regout(<*observable pipeline port for round0*>**),**
.observableround1_pipeline_regout(<*observable pipeline port for round1*>**),**

.observablesaturate_pipeline_regout(<*observable pipeline port for saturate*>**),**
.observablemultabsaturate_pipeline_regout(<*observable pipeline port for multabsaturate*>**),**
.observablemultcdsaturate_pipeline_regout(<*observable pipeline port for multcdsaturate*>**),**
.observablesigna_pipeline_regout(<*observable pipeline port for signa*>**),**
.observablesignb_pipeline_regout(<*observable pipeline port for signb*>**):** assigns a name for the observable pipeline port for the input signal. The observable port will only be created if its corresponding port is registered in the pipeline. These cannot be connected to other routing and are used for simulation and timing purposes only. By connecting wires to the observable pipeline ports, the user is able to rename the observable pipeline ports to the name of the connected wires. **This feature has not been implemented yet for these ports.**
.observablemode0_regout(<*observable port for mode0*>**),**
.observablemode1_regout(<*observable port for mode1*>**),**
.observablezeroacc1_regout(<*observable port for zeroacc1*>**),**
.observablesaturate1_regout(<*observable port for saturate1*>**):** assigns a name for the observable port for the input signal for dynamic mode. These cannot be connected to other routing and are used for simulation and timing purposes only. By connecting wires to the observable ports, the user is able to rename the observable ports to the name of the connected wires. **This feature has not been implemented yet for these ports.**
.observablemode0_pipeline_regout(<*observable pipeline port for mode0*>**),**
.observablemode1_pipeline_regout(<*observable pipeline port for mode1*>**),**
.observablezeroacc1_pipeline_regout(<*observable pipeline port for zeroacc1*>**),**
.observablesaturate1_pipeline_regout(<*observable pipeline port for saturate1*>**):** assigns a name for the observable pipeline port for the input signal for dynamic mode. These cannot be connected to other routing and are used for simulation and timing purposes only. By connecting wires to the observable pipeline ports, the user is able to rename the observable ports to the name of the connected wires. **This feature has not been implemented yet for these ports.**

1.2.3 MAC Output Cell Output Ports

.dataout(<*data output bus*>**)**: the output bus of the MAC output cell (could be registered if output_clock option is specified). When saturation is enabled for any of the multipliers or for the accumulator, then the unused lower bits of the output function as the saturation overflow signals as described later.
.accoverflow(<*accumulator overflow signal*>**)**: in accumulator mode, indicates when the accumulator has overflowed (or underflowed if dealing with very negative numbers); in two-level-adder mode, indicates when the bottom multiplier's saturation overflow has been asserted. This signal passes through the output register bank and therefore has the same register usage (i.e. bypassed/used) as the data output, and the same register controls. In accumulator mode, this means the register is always used, since the accumulator must always be output registered. Note that this signal does not latch – i.e. it only indicates whether the last accumulation operation overflowed (or multiplier saturated), and gets cleared for the next cycle. This signal can be latched by adding external logic to implement a latch.

1.2.4 MAC Output Cell Modes

<*clk for …*> is one of *{none, 0, 1, 2, 3}*. Specifies which clock signal, if any, drives the given register, where the number indicates which clk signal (from clk[0] to clk[3]) should be used. If "none" is

specified, then the corresponding signal is not registered. Otherwise, the corresponding signal will be registered and will be driven by the indicated clock, which must be connected. Additionally, this parameter also specifies which clock enable signal will be used. This forces the use of ena[2] if clk[2] is used for example. If the clock enable signal is not used, it must be tied to VCC.

<*aclr for …*> is one of *{none, 0, 1, 2, 3}*. Specifies which asynchronous clear signal, if any, drives the given register, where the number indicates which aclr signal (from aclr[0] to aclr[3]) should be used. If the option is not "none", then the register's clk parameter cannot be "none", and the corresponding signal's register will be cleared by the given aclr signal. However, if the option is "none", then the register's clk parameter must also be "none", and the corresponding signal is not registered. To disable the aclr, this signal should be tied to VCC.

<*clk for … pipeline*> is one of *{none, 0, 1, 2, 3}*. These behave identically to the clk parameters for the other registers. However, these signals control the pipeline register for the given signal (i.e. second stage register) instead.

<*aclr for … pipeline*> is one of *{none, 0, 1, 2, 3}*. These behave identically to the aclr parameters for the other registers. However, these signals control the pipeline register for the given signal (i.e. second stage register) instead.

<*dataa width*>, <*datab width*>, <*datac width*>, <*datad width*> is an integer in the range {0, 1, 2, …, 36}. It specifies the width of the input ports for this output cell. If a MAC_MULT is feeding the input port, then this width must be the same as the multiplier's output width. This parameter also implies the output width as described in Table 1. In most modes, all the widths must be the same and have a minimum value of 2. In 36_bit_multiply mode, width_a + width_b must be equal to width_c + width_d. In accumulator mode, width_b can be between 0 (unconnected) and 36 (or the output width if that is less). *This field is required.*

Table 1: Output Widths for MAC Output Cell

Operation Mode	**Output Width for dataout port**
Output Only	width_a
Accumulator	width_b + 16
One Level Adder	width_a + 1
Two Level Adder	width_a + 2
36_Bit_Multiply	width_a + width_b
Dynamic	width_a + width_b + width_c + width_d = 144

<*width of dataout port*> specifies the width of the dataout port. This is redundant information since the width is derived from the input port widths, but is needed for sim models that cannot do the calculation. This is only a simulation parameter and is ignored by the compiler.

<*operation mode*> is one of *{output_only, accumulator, one_level_adder, two_level_adder, 36_bit_multiply, dynamic}*. It specifies how the output block behaves and which optional components within the block are used. See later for details on the operation of the individual modes. *This field is required*

<*dataa forced to zero*>, <*datac forced to zero*> is a parameter used to force the inputs for dataa or datac to ground. Valid values are "yes" and "no", where "yes" implies that the given port will be forced to ground. This is only usable in dynamic mode, and can be used to allow the scan-chains to still be usable when switching to accumulator mode since the scan-path will not affect the accumulator's

datab input. For other modes, this parameter is ignored. This parameter is optional and defaults to "no" if left unspecified.

1.2.5 MAC Output Cell Polarities and Default Values

All control signals are active high. All control signal inputs have programmable inverts. MAC_MULT data inputs have programmable inverts, while most MAC_OUT data inputs do not have programmable inverts. The exception is in accumulator mode, where the MAC_OUT has programmable inverts on the Data B input, provided that that the MAC_MULT is not using its Scanout port.

1.3 Modes of Operation

This section describes each of the five possible modes of operation for a MAC slice. In each mode, certain signals are allowed, and certain restrictions are required.
The following are conditions that apply in all modes of operation:

1. In all modes, the "Sign A" and "Sign B" signals on the multiplier can be used, but are not required. They default to VCC (i.e. signed integer) when left unconnected. Since there are only two sign signals for the entire MAC, there can only be two sign signals that drive all the cells in a MAC slice (e.g. signa must be the same signal for the MAC_MULT and the MAC_OUT cell it drives). Similarly, the register control parameters must be identical for each of the sign signals.
2. In all modes, all dynamic control signals (i.e. Sign A/B, AddnSub 0/1, Zero Acc) can be independently registered and/or pipelined (i.e. double registered) if the signal is connected. Additionally, the operand and load data inputs, MAC_MULT outputs, and MAC_OUT outputs can be independently registered.
3. In most modes, any register can be used or bypassed, though a register can only be used if its corresponding signal is connected. The only exception is the multiply-accumulator mode where the output registers of the MAC_OUT are required. If a register is used, then it must have a valid CLK and ACLR selection, and its specified CLK, ENA, and ACLR ports must be connected.
4. If the Scanout signals of a multiplier are used and they drive another multiplier's Data inputs, then both multipliers must be of the same internal base width (i.e. 9-bits or 18-bits). Also, they both must be in the same operation mode.
5. Each Scanout signal can be used or bypassed independently (i.e. Scanout A can be used, while Scanout B is not used).
6. The Scanout signal of the bottom multiplier in a MAC can either drive the next MAC through dedicate internal routing, or it can drive regular routing (and possibly the next MAC through regular routing). However, if it drives regular routing, then some multipliers may become unusable, depending on the mode.

1.3.1 Multiplier Only Mode

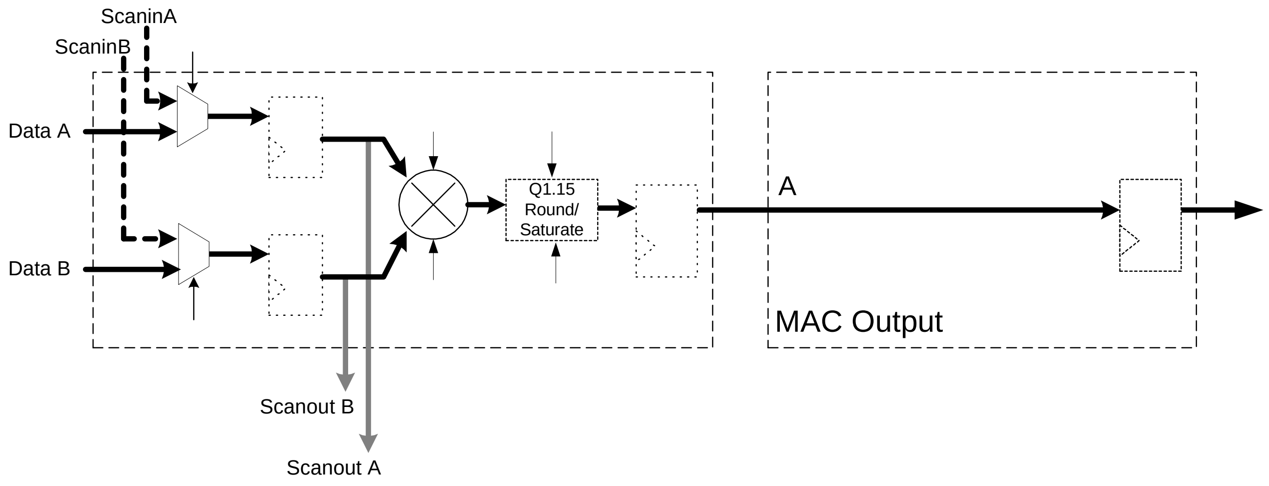

Figure 2: Multiplier Only Mode

In this mode, an independent multiplier is implemented. The following are conditions associated with this mode:

1. Up to four 18-bit multipliers or eight 9-bit multipliers can be implemented per MAC.
2. MAC_MULT conditions:
    a. The width of each input can be 1 to 18 bits.
    b. The width of the output is equal to width_a + width_b
    c. The Scanout signals can be used
3. MAC_OUT conditions:
    a. The width of the input must be the same as the output width of the multiplier feeding it (i.e. in the range 2 to 36 bits).
    b. The width of the output is equal to the width of the input (i.e. width_a)
    c. Data A must be connected – all other data ports cannot be used.
    d. The dynamic control signals AddnSub 0/1 and ZeroAcc are not used and cannot be connected.
4. Scan Chain conditions:
    a. The bottom multiplier(s) becomes unusable if the scanout signals are driven to regular routing since there are not enough outputs in the MAC for both the scanout signals and the bottom multiplier's results. In 18-bit mode, the bottom multiplier in the MAC becomes unusable and only 3 multipliers can be used in the MAC. In 9-bit mode, two multipliers can have their scanout signals drive regular routing (the ones in LAB rows 4 and 5). Therefore, depending on whether the multiplier above it is set to use regular routing, the two multipliers in rows 6 and 7 may become unusable. Only one of them may become unusable, or both of them may become unusable since choosing the scanout path is independent for both.
5. Q1.15 Saturation overflow conditions:
    a. When saturation is enabled in the multiplier, the saturation overflow for the multiplier is output on MAC_OUT.dataout[0] ONLY if dataout[] is a 36-bit bus. If the output width is less than 36-bits, then the saturation overflow signal cannot be accessed.

1.3.2 Multiply-Accumulator Mode

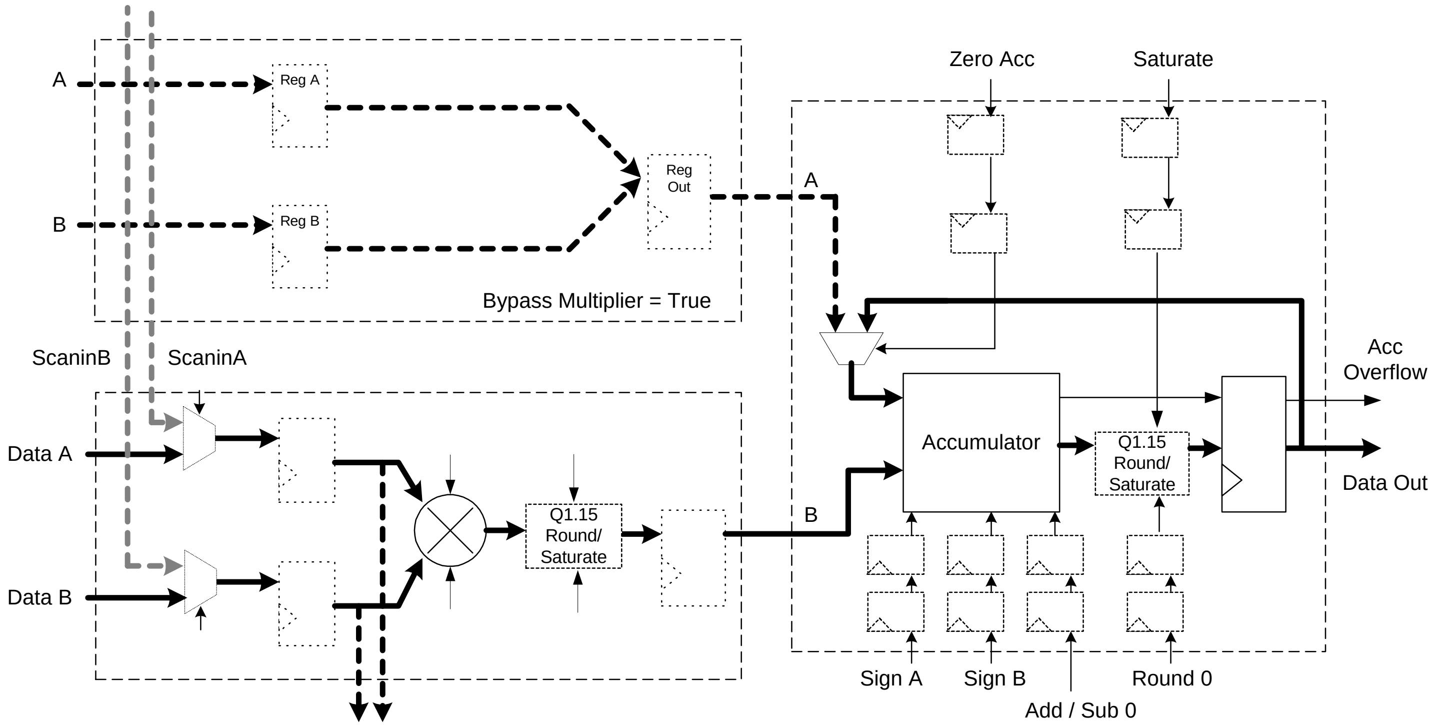

Figure 3: Multiply-Accumulator Mode

In this mode, a multiplier feeding a loadable accumulator is implemented. The following are conditions associated with this mode:

1. Up to two 18-bit multiply-accumulators can be implemented per MAC. Accumulators with 9-bit based multipliers will not be implemented differently than 18-bit multipliers, so still only two 9-bit based multipliers can be implemented per MAC.
2. MAC_MULT conditions:
   a. The width of each input can be 1 to 18 bits (except for the MAC_MULT in bypass mode where a width of 0 is permitted).
   b. The width of the output is equal to width_a + width_b
   c. The Scanout signals can be used, except if the accumulator below this in the scan-chain is being loaded externally (i.e. if Data A of MAC_OUT is connected for that mult-accum).
   d. The top MAC_MULT is optional and does not have to be connected. However if it is connected and does not have the bypass flag set, and the bottom multiplier is not connected, then it will be assumed that this was intended to be the multiplier and it will be moved to the bottom multiplier spot.
   e. The top MAC_MULT bypasses the multiplier if it exists since it is used to load the accumulator. This is used for loading the upper bits of the accumulator: dataa[] specifies the upper width_a bits, and datab[] specifies the next width_b bits (i.e. the MSB of dataa[] corresponds to the MSB of the load_data bus, and the LSB of datab[] corresponds to the LSB of the load_data bus). width_a must be the maximum width of 18 if width_b is not zero.
3. MAC_OUT conditions:
   a. The width of the Data B input must be the same as the output width of the multiplier feeding it (i.e. in the range 2 to 36 bits).

   b. The width of the output is equal to width_a + 16 (i.e. 16-bits extra to do the accumulation to a maximum of 52-bits)
   c. The output registers of the MAC_OUT are required in this mode.
   d. Data B must be connected and should be fed by a MAC_MULT for the multiplication.
   e. Data A may optionally be connected, and should be fed by a MAC_MULT in bypass mode if it is connected. When Zero-Acc is asserted, the accumulator is fed the value of Data A for the upper bits and 0 for the lower bits, instead of the feedback from the output registers. If Data A is not connected, then a default of 0 for this port is assumed (which is implemented on chip using the pipeline register's 5th GND selection bit). If Data A is connected, then the dedicated scan-chain path cannot be used to feed the multiplier.
   f. The dynamic control signal AddnSub 0 can be used if desired to perform decimation (i.e. subtract multiplier result from total instead of adding), and defaults to VCC if only addition is desired. The dynamic control signal AddnSub 1 is not used and cannot be connected.
   g. The dynamic control signal "Zero Acc" can be used, and defaults to GND if left unconnected. This signal, if asserted, clears the feedback input from the output registers feeding into the Accumulator, or if Data A is connected, it sets the feedback of the upper 36-bits to Data A. This signal can be registered and pipelined.
   h. The output signal "Acc Overflow" indicates when the accumulator has overflowed, or underflowed (i.e. result is less than lowest negative number allowed). This signal is automatically cleared in the next clock cycle and is registered by the same output register bank used by the accumulator.
4. Scan Chain conditions:
   a. The bottom multiply-accumulator becomes unusable if the scanout signals are driven to regular routing instead of using the dedicated inter-MAC scanout routing. Therefore, only one multiply-accumulator can be implemented in a MAC that uses its scanout signal and drives regular routing.
   b. If the bypass MAC_MULT is connected for loadable accumulation, then the dedicated scan-chain path cannot be used to feed the multiplier, unless it is being fed by GND (i.e. doing zero accumulation).
6. Q1.15 Saturation overflow conditions:
   a. When saturation is enabled in the multiplier, the saturation overflow for the multiplier is output on MAC_OUT.dataout[1] and for the accumulator is output on MAC_OUT.dataout[2] ONLY if dataout[] is a 52-bit bus. If the output width is less than 52-bits, then the saturation overflow signal cannot be accessed.

1.3.3 Multiply-One-Level-Adder Mode

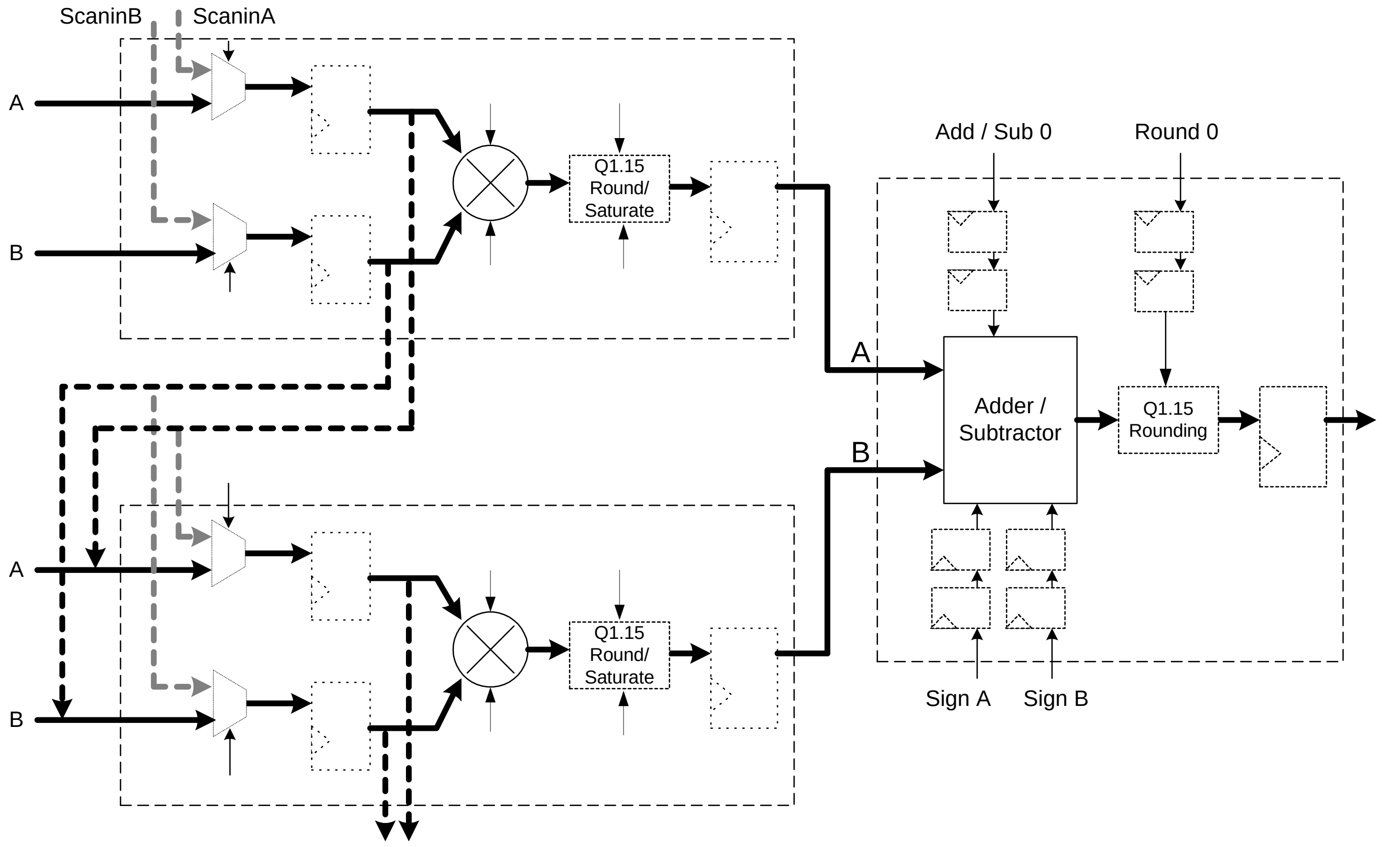

Figure 4: Multiply-One-Level-Adder Mode

In this mode, two multipliers feeding an adder/subtractor is implemented. The following are conditions associated with this mode:

1. Up to two 18-bit multiply-one-level-adders or four 9-bit multiply-one-level-adders can be implemented per MAC.
2. MAC_MULT conditions:
   a. The width of each input can be 1 to 18 bits.
   b. The width of the output is equal to width_a + width_b. The output width of both multipliers must be the same (though all four operand input widths can be different).
   c. Either multiplier can use their Scanout signals. If the top multiplier uses its Scanout signal then it must feed the bottom multiplier's data inputs – it cannot drive anything else. If the bottom multiplier's Scanout signals are used, then it can drive another multiplier of the same base width as described earlier, or it can drive regular routing (and thus any other cell). However, if two multiply-one-level-adders are in the same MAC then the top one can only drive the bottom one, since only the bottom multiplier in a MAC can drive regular routing.
3. MAC_OUT conditions:
   a. The width of each input must be the same as the output width of the multiplier feeding it (i.e. in the range 2 to 36 bits). Additionally, both inputs must have the same width.
   b. The width of the output is equal to width_a + 1

   c. The Data A and Data B ports must be connected – all other input ports cannot be used.
   d. The dynamic control signals AddnSub 1 and ZeroAcc are not used and cannot be connected.
   e. The dynamic control signal AddnSub 0 can be used and defaults to VCC (add) if left unconnected. This signal controls whether addition or subtraction is performed.
4. Scan Chain conditions:
   a. In this mode no multipliers are lost if the scanout signal drives regular routing. Unlike the previous modes, in this modes there are enough MAC outputs to accommodate both the scanout signals and the adder outputs.
   b. When scanouta (not scanoutb) drives regular routing, and when mixing modes with the one-level adder in the lower half, then the upper half will not be able to use Mult B if the upper half implements independent multipliers, though Mult A is still usable. Mixing with other modes does not cause any change.
5. Q1.15 Saturation overflow conditions:
   a. When saturation is enabled in a multiplier, the saturation overflow for the multiplier feeding dataA is output on MAC_OUT.dataout[0] and for the multiplier feeding dataB is output on MAC_OUT.dataout[1] ONLY if dataout[] is a 37-bit bus. If the output width is less than 37-bits, then the saturation overflow signals cannot be accessed.

1.3.4 Multiply-Two-Level-Adder Mode

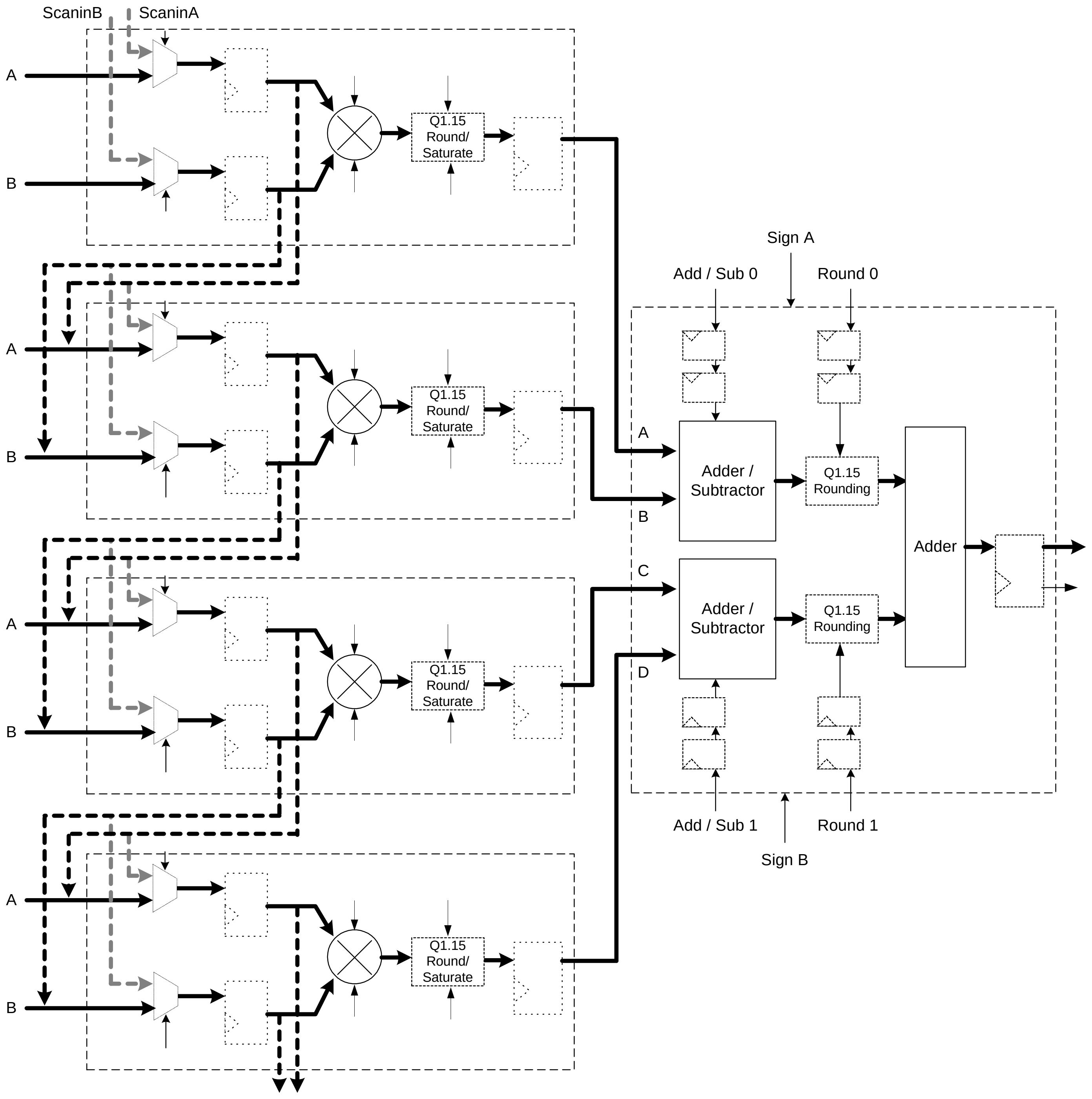

Figure 5: Multiply-Two-Level-Adder Mode

In this mode, four multipliers feeding a two-level adder is implemented. The following are conditions associated with this mode:

1. Up to one 18-bit or two 9-bit multiply-two-level-adders can be implemented per MAC.
2. MAC_MULT conditions:
    a. The width of each input can be 1 to 18 bits.

   b. The width of the output is equal to width_a + width_b. The output width of all multipliers must be the same (though their input widths can be different).
   c. Either multiplier can use their Scanout signals. As in the One-Level-Adder case, if any multiplier other than the last multiplier uses its Scanout signal, it can only drive its next adjacent multiplier. Only the last multiplier can drive regular routing or another multiplier of the same base width.

3. MAC_OUT conditions:
   a. The width of each input must be the same as the output width of the multiplier feeding it (i.e. in the range 2 to 36 bits). Additionally, all inputs must have the same width.
   b. The width of the output is equal to width_a + 2
   c. All Data inputs must be connected. To implement 3 multipliers feeding an adder, simply set both operand inputs of one of the multipliers to 0.
   d. The dynamic control signal ZeroAcc is not used and cannot be connected.
   e. The dynamic control signals AddnSub 0 and AddnSub 1 can be used and default to VCC (add) if left unconnected. These signals control whether addition or subtraction is performed by each of the first level adder units – AddnSub 0 controls the unit that is fed by Data A and Data B, while AddnSub 1 controls the unit that is fed by Data C and Data D. The second stage adder can only perform addition.
   f. The accoverflow port is used in this mode: it is used to output the bottom multiplier's saturation overflow signal.

6. Scan Chain conditions:
   a. In this mode no multipliers are lost if the scanout signal drives regular routing. Unlike the previous modes, in this modes there are enough MAC outputs to accommodate both the scanout signals and the adder outputs.

7. Q1.15 Saturation overflow conditions:
   a. When saturation is enabled in a multiplier, the saturation overflow for the multiplier feeding dataA is output on MAC_OUT.dataout[0], for the multiplier feeding dataB is output on MAC_OUT.dataout[1], for the multiplier feeding dataC is output on MAC_OUT.dataout[2], and for the multiplier feeding dataD is output on MAC_OUT.overflow, ONLY if dataout[] is a 38-bit bus. **Note that multD's overflow is output on the accoverflow port.** If the output width is less than 38-bits, then the saturation overflow signals cannot be accessed.

1.3.5 36-bit Multiplier Mode

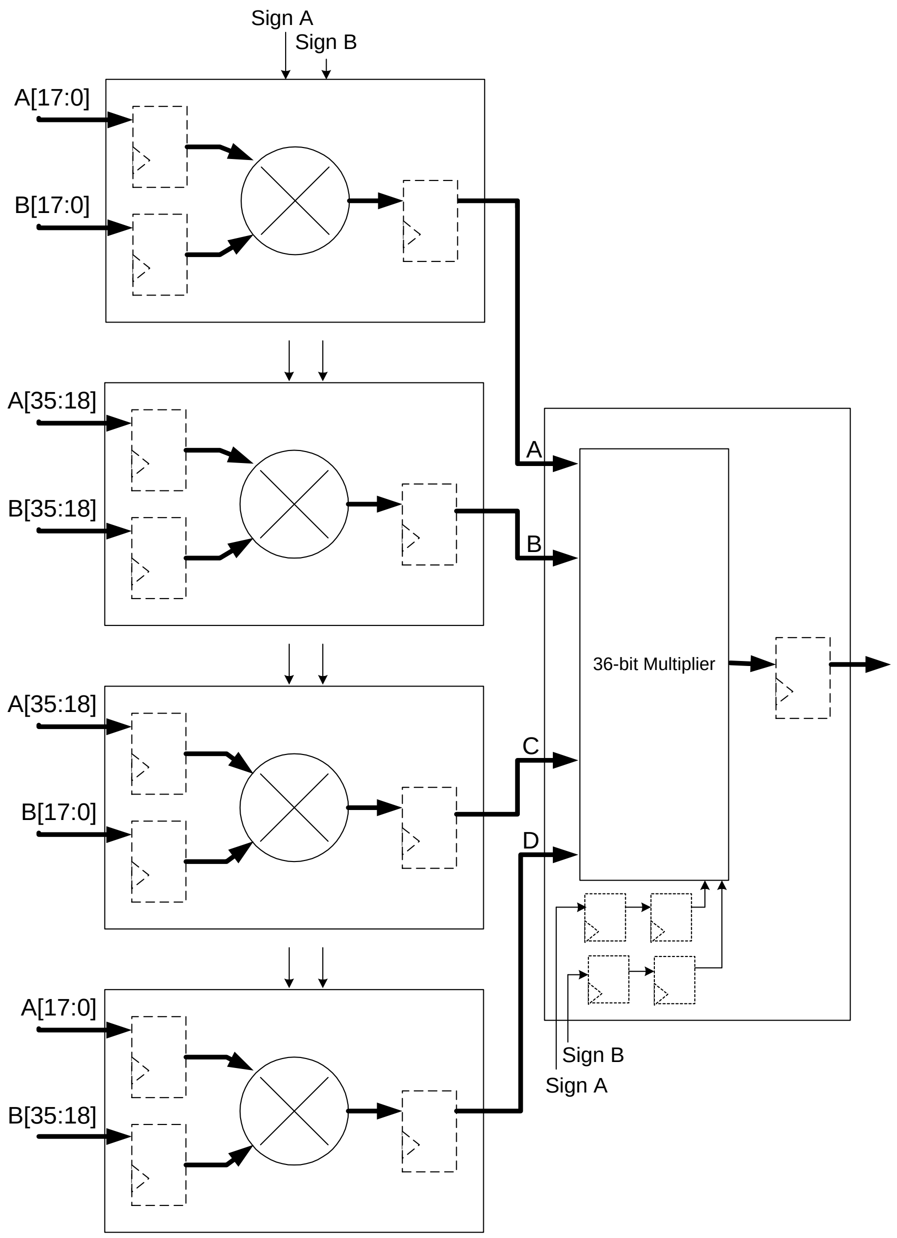

Figure 6: 36-bit Multiplier Mode

In this mode, an independent 36-bit multiplier is implemented. This is accomplished by using four 18-bit multipliers and special partial-adder circuitry to perform a 36-bit multiplication. The four 18-bit multipliers are fed part of each input as outlined in the diagram. The following are conditions associated with this mode:

1. Only one 36-bit Multiplier using four 18-bit multipliers can be implemented per MAC.
2. MAC_MULT conditions:
    a. The width of each input can be 1 to 18 bits. However, this is completely dependent on the width of the original 36-bit inputs. To implement smaller width multipliers, the low-order bits must be padded with GND. For example, if operand A is actually 27 bits wide, then all bits in A[35:18] and A[17:9] must be connected, while all bits in A[8:0] must be tied to

GND. Additionally, if operand B is actually 15 bits wide, then all bits in B[35:21] must be connected, and all bits in B[20:18] and B[17:0] must be tied to GND. This implies the MAC_OUT would have the 42-bit result at OUT[71:30], and OUT[29:0] should be ignored.

   b. The width of each output is equal to width_a + width_b
   c. The Scanout signals cannot be used

3. MAC_OUT conditions:
   a. The width of the input must be the same as the output width of the multiplier feeding it (i.e. in the range 2 to 36 bits).
   b. The width of the output is equal to width_a + width_b
   c. All Data input ports must be connected as outlined in the diagram.
   d. The dynamic control signals AddnSub 0/1 and ZeroAcc are not used and cannot be connected.
8. Scan Chain conditions:
   a. In this mode the scanout signals cannot be used. If scan chains are desired, they should be implemented by the user external to the MAC block using regular logic.
9. Rounding/Saturation conditions:
   a. Rounding and saturation cannot be used in this mode. This can be done externally in LEs if desired since this mode implements a single multiplier.

1.3.6 Dynamic Mode

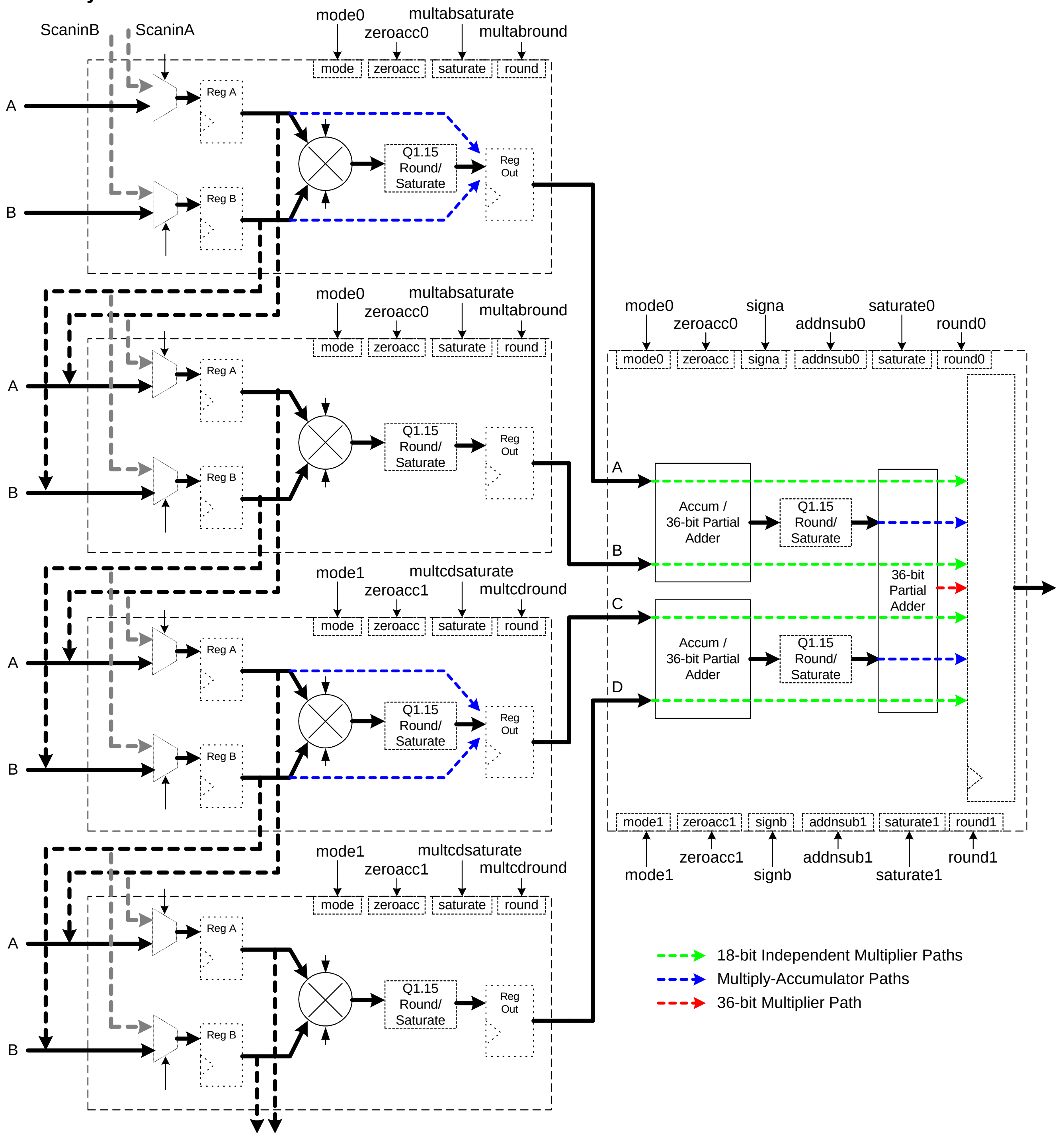

Figure 7: Dynamic Mode

In this mode, a dynamic MAC block is implemented that can switch dynamically between up to four 18-bit independent multipliers, up to two multiply-accumulators, and one 36-bit multiplier. In this mode, all the MAC_MULT WYSIWYGs must also be specified as being in dynamic mode in addition to the MAC_OUT WYSIWYG.

To switch between the various modes, the mode0/1 and zeroacc0/1 signals are used. The mode0 and zeroacc0 signals control the top half of the MAC block, while the mode1 and zeroacc1 signals control the bottom half of the MAC block. To implement 36-bit multiplier mode, the top half and bottom half of the MAC must have the same mode. But for 18-bit multipliers and multiply-accumulators, each half can be in either mode since they operate independently. The following table describes the values to select each mode.

Mode	**mode0/1**	**zeroacc0/1**
Independent 18-bit Multipliers	0	0
Multiply-Accumulators	1	X (used by accum)
36-bit Multiplier	0	1

As shown in the diagram, depending on the mode, a different path through the MAC block will be taken.

1. For independent 18-bit multipliers, all the adder units are bypassed and the multiplier results feed the output registers directly.
2. For the multiply-accumulators, the 2nd level adder is bypassed and the first-level accumulator units feed the output registers directly. Additionally, the multipliers feeding A and C of the MAC_OUT have the inputs bypassing the multiplier circuitry and feeding the multiplier output registers directly (in bypass mode), since they are used to load the accumulators.
3. For 36-bit multiplier mode, the adder units perform partial-product-summation and the output from the 2nd level adder feeds the output registers. Additionally, the multipliers internally ground the sign signals for some operands in order to correctly perform the partial multiplications.

As shown in the diagram, the signals to control the top and bottom half of the MAC block are connected to ports that don’t necessarily match the name on each of the WYSIWYGs. For example, the zeroacc0 signal is connected to the zeroacc ports (notice the missing ‘0’) of the top two multipliers as well as the MAC_OUT. But the zeroacc1 signal is connect to the zeroacc ports of the bottom two multipliers, and the zeroacc1 port of the MAC_OUT. As expected, the signals ending with ‘0’ control the top half of the MAC block, while the signals ending in ‘1’ control the bottom half of the MAC block.

Each half of the MAC block can operate in a different mode since each half has a separate mode control. However, in 36-bit multiplier mode, the top-half and the bottom half of the MAC should be the same.

All input operands are 18-bits – no other input widths are supported.

The output of the MAC_OUT is 144-bits, which is divided into eight 18-bit register banks. To enable independent configuration of the output registers, separate parameters are used to configure different sections of the output as specified below.

Bank	**Bank Range**	**Clock Selection Param**	**Clear Selection Param**
0	dataout[17:0]	output_clock	output_clear
1	dataout[35:18]	output1_clock	output1_clear
2	dataout[53:36]	output2_clock	output2_clear
3	dataout[71:54]	output3_clock	output3_clear
4	dataout[89:72]	output4_clock	output4_clear
5	dataout[107:90]	output5_clock	output5_clear

6	dataout[125:108]	output6_clock	output6_clear
7	dataout[143:126]	output7_clock	output7_clear

Depending on the mode, the output of the MAC_OUT should be interpreted differently, as specified below. Note that the outputs are not always contiguous and are not always in order. Also, since each half of the MAC block can operate in a different mode, each half of the dataout[] range can represent the output for a different mode, where banks 0 to 3 are in the top half, and banks 4 to 7 are in the bottom half (though both halves must be in the same mode for 36-bit multiplier operation).

Mode	**Result Range**	**Dataout Range**	**Output Bank**
18-bit Multipliers	MultA[35:0]	dataout[35:0]	1, 0
	MultB[35:0]	dataout[71:36]	3, 2
	MultC[35:0]	dataout[107:72]	5, 4
	MultD[35:0]	dataout[143:108]	7, 6
Multiply-Accumulators	MAcc0[51:36]	dataout[52:37]	2
	MAcc0[35:0]	dataout[35:0]	1, 0
	Overflow0	dataout[53]	2
	MAcc1[51:36]	dataout[124:109]	6
	MAcc1[35:0]	dataout[107:72]	5, 4
	Overflow1	dataout[125]	6
36-bit Multiplier	Mult36[71:54]	dataout[71:53]	3
	Mult36[53:36]	dataout[35:18]	1
	Mult36[35:18]	dataout[53:36]	2
	Mult36[17:0]	dataout[17:0]	0

The table above sorted by dataout[] in 18-bit segments is as follows:

Bank	**Bank Range**	**18-bit Multipliers**	**Multiply-Accumulators**	**36-bit Multiplier**
0	dataout[17:0]	MULTA[17:0]	MAcc0[17:0]	Mult36[17:0]
1	dataout[35:18]	MULTA[35:18]	MAcc0[35:18]	Mult36[53:36]
2	dataout[53:36]	MULTB[17:0]	Overflow0, MAcc0[51:36], X(1)	Mult36[35:18]
3	dataout[71:54]	MULTB[35:18]		Mult36[71:54]
4	dataout[89:72]	MULTC[17:0]	MAcc1[17:0]	
5	dataout[107:90]	MULTC[35:18]	MAcc1[35:18]	
6	dataout[125:108]	MULTD[17:0]	Overflow1, MAcc1[51:36], X(1)	
7	dataout[143:126]	MULTD[35:18]		

Notes: (1) 'X' designates the low order bit that is not driven when in Multiply-Accumulator mode
(2) For unused output ranges for a particular mode, the output is not driven and has an unknown value

For simulation purposes, the signa/b_internally_grounded parameter and the bypass_multiplier parameter on the MAC_MULT WYSIWYG must be set assuming the 36-bit multiplier mode and the accumulator mode, respectively, is being used. The parameter will be ignored if it is not applicable to the dynamic mode currently in operation.

1.4 Rounding and Saturation

The rounding and saturation features are available when using the Q1.15 number format. In this format, 16-bits are used to represent a fixed-point number, where the MSB represents the sign, and 15-bits are used to represent the value after the decimal place (the fractional value). This is equivalent to saying that the Q1.15 value as an integer number can be divided by 2^{15} to get the fixed-point number.

```
e.g. –1/2 = 1 100 0000 0000 0000 <==> -0x4000 / 2^15
      1/8 = 0 001 0000 0000 0000 <==>  0x1000 / 2^15
```

Consequently, if Q1.15 numbers are used, then the multiplier input widths should be 16-bits, and the output will be a Q2.30 number (32-bits). For a Q1.30 result, ignore the MSB of the multiplier result, and for a Q1.15 result, ignore the MSB and the lower 15-bits of the multiplier result.

Similarly, if two multipliers feed a single adder, then the result will be a Q3.30 number (33-bits). And the result of a two-level adder will be a Q4.30 number (34-bits). For the accumulator, the result will be a Q18.30 number (48-bits) since the extra 16-bits of accumulation provided are simply extra sign bits.

So the following generalization can be made about the number formats in each of the different modes. The parameter NUM_SIGN refers to the number of sign bits (i.e. number of bits to the left of the decimal point), and NUM_FRACTION refers to the number of fraction bits (i.e. number of bits to the right of the decimal point), for the inputs to the multipliers:

Multiplier Inputs are Q*<NUM_SIGN>.<NUM_FRACTION>* format numbers

Multiplier result is Q*<NUM_SIGN*2>.<NUM_FRACTION*2>*
One level adder result is Q*<NUM_SIGN*2+1>.<NUM_FRACTION*2>*
Two level adder result is Q*<NUM_SIGN*2+2>.<NUM_FRACTION*2>*
Accumulator result is Q*<NUM_SIGN*2+16>.<NUM_FRACTION*2>*

Note that the fraction width remains the same after the multiplication, while only the number of sign bits differs. NUM_SIGN (number of input sign bits) must be 1 to be supported by the MAC, but NUM_FRACTION (number of input fraction bits), can be any number from 1 to 17 (though a width of 15 is usually expected from the user, and a width of 17 is what it will be mapped to in the hardware).

The dynamic control signal "Round" when asserted will enable the rounding feature such that the result is rounded to the nearest Q1.15 value (instead of just truncating). The dynamic control signal "Saturate" when asserted will enable the saturation feature such that the result will be the maximum Q1.15 value if an overflow occurs or the minimum Q1.15 value if an underflow occurs. The flag to indicate when saturation occurs appears on the unused LSBs of the result[] bus depending on the mode (see description of each mode to get these details). When the rounding and saturation features are both enabled on a data path, rounding will always be performed before saturation. Both control signals are optional, and will default to GND (i.e. unasserted) if left unconnected.

The following generalization can be made regarding how rounding and saturation behave, regardless of where in the logic it is performed. The parameter WSIGN refers to the number of sign bits (i.e. number of bits to the left of the decimal point), WFRACTION refers to the number of fraction bits (i.e. number of bits to the right of the decimal point), and WFRACTION_ROUND refers to number of fraction bits to round to:

```
Input to rounding/saturation block:
        Q<WSIGN>.<WFRACTION>
        Total width WTOTAL = WSIGN + WFRACTION
        Input bus: datain[WTOTAL-1..0]
        Sign bits: datain[WTOTAL-1 .. WTOTAL-WSIGN] = datain[WTOTAL-
       1..WFRACTION]
        Fraction bits: datain[WTOTAL-WSIGN-1 .. 0] = datain[WFRACTION-1..0]

The rounding block performs the following:
        roundout[] = datain[] + round_adder_constant
Where when rounding is enabled, and WFRACTION > WFRACTION_ROUND,
        round_adder_constant = (1 << WFRACTION-WFRACTION_ROUND-1)
Otherwise,
        round_adder_constant = 0

The saturation block performs the following:
        satout[] = satoverflow ? satvalue[] : roundout[]
Where when saturation is enabled,
        satoverflow = 1 if any of the sign bits are different, otherwise 0 (i.e. XOR all sign bits)
        satvalue[WTOTAL-1..WFRACTION] = roundout[WTOTAL-1]
        satvalue[WFRACTION-1..0] = ! roundout[WTOTAL-1]
Otherwise,
        satoverflow = 0 (i.e. just copy roundout[] to satout[])

Finally, the output of this block is as follows:
        dataout[] = satout[]
Where if rounding is enabled, and WFRACTION > WFRACTION_ROUND,
        dataout[WFRACTION-WFRACTION_ROUND-1..0] = 0
```

Given this generalization, it is sufficient to specify these parameters to implement rounding and saturation for all cases:

Block	**WTOTAL**	**WSIGN**	**WFRACTION_ROUND**
After multiplier	Multiplier result width	2	15
After one-level adder	Adder result width	3	15
In Accumulator	Accumulator result width	18	15

And by definition, WFRACTION = WTOTAL – WSIGN