Stratix EDA and Academic Developer Functional Description

Version 1.35

December 15, 2005

By

Altera Corporation

Table of Contents:

1. Overview

This document is intended for developers working with the Stratix™ and Cyclone™ architectures. It is a companion to the WYSIWYG Device Primitive User Guide for Stratix and should be used in conjunction with that document. This document provides detailed information about the Stratix and Cyclone architectures that enables CAD tool developers to synthesize designs to the architecture which will not have fitting problems. It also enables CAD tool developers to create placements for Stratix and Cyclone which will be routable and will not violate any constraints on what constitutes a legal LAB or DSP block.

2. Coordinate System and Location Assignments

The diagram below shows the coordinate system used in Stratix and Cyclone. Note that some of the function blocks shown (M512 RAMs, M-RAMs and DSP blocks) do not exist in Cyclone.

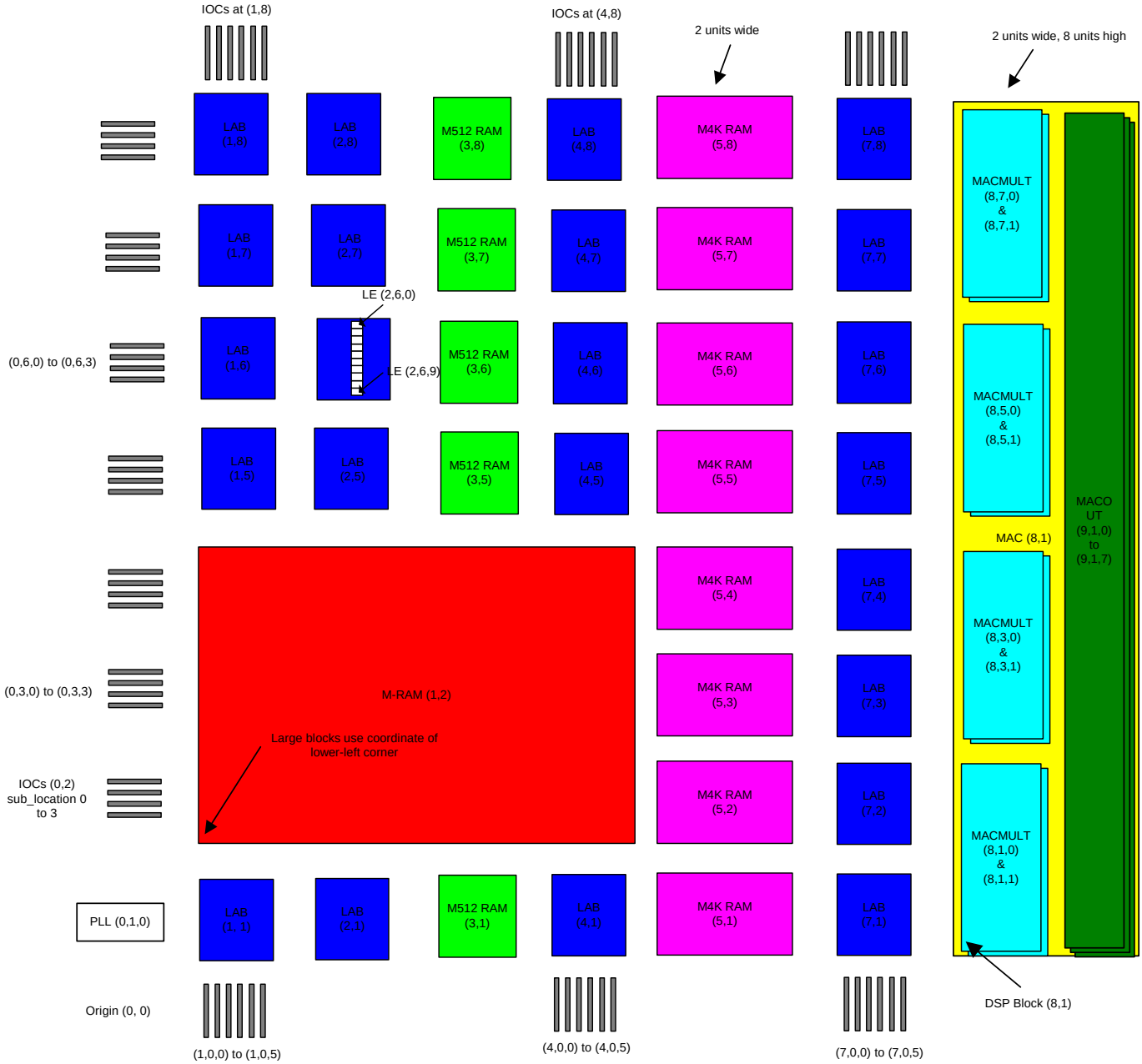


Figure 1: Stratix coordinate system.

Stratix and Cyclone use a flat coordinate system, with the origin (0, 0) in the **lower-left** corner. The routing channels define an (x,y) grid, and every block is identified by its (x,y) location. Each row is represented by a y-value, e.g. Y6, starting at 0 for the bottommost row. Each column is represented by an x-value, e.g. X4, starting at 0 for the leftmost row. Large blocks that span multiple grid points are referred to by the coordinate of their lower-left corner.

To the fitter, all location assignments are 2D region assignments. E.g. custom_region_X4_Y3_X8_Y7 constrains a cell to the region between coordinates (4, 3) and (8,

7), inclusive. Assignments to specific LABs and ESBs (e.g. LAB_X7_Y3) are simply a shorthand for 2D regions that are the same size as the block to which the assignment is being made.

You can make assignments to individual logic cells. For example, LE_X4_Y6_N0 assigns a circuit element to the logic cell at x = 4, y = 6 and number = 0. Number is simply an index that is used to differentiate between cells when there are multiple cells at a certain (x, y) location. Note that generally it is a bad idea to constrain logic cells all the way down to the number, or LE-level. Constraining cells all the way down to the LE-level doesn't provide any additional timing predictability vs. constraining to the LAB level, and it constrains the router more, causing routing to be slightly more difficult.

If a region assignment includes any portion of a large block within it, the entire large block is considered to be a legal placement location. For example, a region assignment of custom_region_X4_Y3_X7_Y3 would be considered to include the M-Ram at (1, 1), the M4K RAM at (5, 3) and the LAB at (7, 3).

3. Routing Delay vs. Distance & Routing Rules

In Stratix and Cyclone, there is no rigid routing hierarchy, so the routing delay increases roughly linearly with Manhattan distance.

The speed of connections in Stratix and Cyclone, from fastest to slowest is:

- Source and destination in the same LAB
    - A connection from the combinational output of a logic cell to the **.datad** input of the logic cell immediately below it (and in the same LAB) is extra-fast. So there is a very fast path from **.combout** of LE #0 to the **.datad** of LE #1 and so on. The fitter will automatically try to exploit this extra-fast connection.
- Destination in the LAB immediately to the left or right of the source. Such connections can be routed by an LE directly driving the LAB lines of its neighbouring LAB.
- Delay increases with Manhattan distance. There is a slight delay advantage (for equal Manhattan distances) when the source and destination are in the same row or column, since no "elbow" switching from vertical to horizontal or vice versa is required in this case. Delay for long-distance connections increases more slowly in Stratix than in Cyclone, since Cyclone has only length 4 routing (R4 and C4 wires), while Stratix has longer wires (R8, V8, R24 and C16 wires) as well as length 4 wires.

There is approximately twice as much horizontal routing as vertical routing in Stratix, so designs should be floorplanned such that most connections run horizontally. Cyclone has the same amount of horizontal and vertical routing (80 R4 and 80 C4 wires per channel).

4. Netlist Recommendations

Most input ports on WYSIWYG primitives have programmable inversion built into their hardware. To take advantage of this programmable inversion, *netlists should directly connect the complement of a signal to an input port if that is the circuit behavior desired*. For example, if it was desired to make a negative-edge triggered register in a logic cell, the netlist should connect **!clock** to the .clk port on the stratix_lcell primitive. The programmable inverter hardware on the logic cell will be used to invert the clock in this case. Consider what would have happened if instead the netlist had used a logic cell to invert the clock and create a new signal, nclock, which was then connected to the .clk port of a logic cell to make a negative-edge triggered register.

This netlist will result in one extra logic cell being created, and lead to a larger circuit with much worse clock skew.

```
stratix_lcell good_cell {
      .clk(!clock),  // good way to make an inverted clock
      ...

stratix_lcell unneeded_inverter {
      .dataa(clock),
      .combout(nclock)  // half of bad way to make an inverted clock
}

stratix_lcell bad_cell {
      .clk(nclock),  // bad way to make an inverted clock
      ...
```

A few input ports on some WYSIWYG primitives do not have programmable inversion. For such ports, a complemented signal (e.g. !clock) cannot be connected, and instead an explicit inversion using a logic cell must be used. See the WYSIWYG Device User Primitives Guide for Stratix for a listing of such ports.

Most input ports on WYSIWYG primitives can also be directly connected to GND and VCC. For such ports it is always best to make such direct connections, rather than creating a logic cell whose output is always 0 or 1 and connecting this signal to ports. A few ports cannot be directly connected to VCC and/or GND; in such cases a logic cell whose output is always 0 or 1 must be created and that signal connected to the desired input ports.

```
stratix_lcell make_preset_using_aload {
      .aload(aloadsig),
      .datac(VCC), // OK, but can't connect .datac to GND directly
      ...
```

5. LE and LAB Fitting Rules

The following rules should be obeyed by synthesis tools in order to ensure that (i) all logic cells are electrically valid, and (ii) when the logic cells are grouped into LABs, the resulting LABs are legal and routable.

5.1 Rules for Individual LEs

1. If the **clk** port is connected, either (i) the **regout** port must be connected or (ii) **sum_lutc_input = qfbk**.
2. The **clk** port must be connected when the **regout** port is connected.
3. The **clk** port must be connected when the **aclr** port is connected.
4. The **clk** port must be connected when the **sclr** port is connected.
5. The **clk** port must be connected when the **sload** port is connected.
6. The **clk** port must be connected when the **aload** port is connected.
7. The **clk** port must be connected when **sum_lutc_mode = qfbk**.
8. The **datac** port must be connected when the **sload** port is connected.
9. The **datac** port must be connected when the **aload** port is connected.

10. The **datac** port cannot be inverted when it is used as syncronous load data, or asynchronous load data.
11. The **datac** port cannot be set to GND.
12. The **clk** port must be connected when **ena** port is connected.
13. The **cin** port
    - can only be connected to a signal sourced by the **cout** port of another logic cell.
    - must not be set to VCC or GND.
    - cannot be connected to an inverted signal (e.g. !my_carry)
14. The **cout** port must either be connected to exactly 1 **cin** port of another logic cell or be unconnected.
15. The **regcascin** port
    - Can only be connected when the register_cascade_mode = on.
    - can only be connected to a signal sourced by the **regout** port of another logic cell
    - must not be set to VCC or GND
    - cannot be connected to an inverted signal
16. The **cout** port can only be connected on logic cells with **operation_mode = arithmetic**, and when **operation_mode = arithmetic** the **cout** port must be connected.
17. The **sload** port can only be connected on logic cells with **synch_mode = on**
18. The **sclr** port can only be connected on logic cells with **synch_mode = on**
19. The **inverta** port can only be connected on logic cells with one or both of **cin** and **cout** connected. This is a software restriction enforced by the Quartus® II development tool to prevent extremely poor fitting.

5.2 From Chains to LABs

A “chain” is a group of logic cells where either (i) the .cout port of one logic cell feeds the .cin port of the next logic cell (carry chain) or (ii) the .regout port of one logic cell feeds the .regcascin port of the next cell (register cascade chain).

In the Stratix and Cyclone architectures, logic cells are grouped into logic array blocks (LABs). Each LAB contains 10 logic cells. The logic cells used in the two architectures are identical.

The logic cells in a chain must go into adjacent logic cells. For example, if the first cell in a carry chain is placed at LE #1 in a LAB, the second cell in the chain **must** be placed in LE #2 of the same LAB. The eleventh logic cell in this chain would have to be placed in LE #0 of the LAB immediately below the LAB that contained the first 10 cells in the chain. The figure below shows how the logic cells in a chain must be placed.

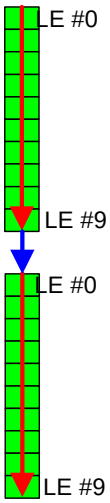


Figure 1: Carry chain connections.

The net effect is that groups of 10 adjacent cells in a chain **must** be placed in the same LAB if the chain starts at LE #0. Note also that there is no register cascade connection between LABs, so register cascade chains can be at most 10 logic cells long, or they will cause a deterministic no-fit.

Also note that any carry chain that uses the inverta signal and is dependent by the lutmask on that signal (at the cin position) **must** start in LE #0 of some LAB. Hence there is exactly one way to divide such a chain into LABs – the first 10 logic cells must go in one LAB, the next 10 logic cells in another LAB and so on.

5.3 LAB-wide Signal Limits

Many of the signals connected to logic cells are “LAB-wide” signals. Such signals are generated once per LAB and shared by all the logic cells placed within the LAB. Hence, having logic cells that require too many distinct LAB-wide signals placed within one LAB is illegal. Since chains of logic cells constrain placement to keep groups of 10 adjacent logic cells in the same LAB, synthesis must be careful not to generate chains of logic cells that cannot be divided into legal LABs. Chains that cannot fit into LABs will be repaired by Quartus by inserting feedthrough logic cells in the chain until it can fit into legal LABs. However, inserting feedthroughs in this way can slow down and increase the size of a circuit, so it is better if synthesis avoids generating such illegal chains. An overly conservative (but simple) rule that can be used by synthesis is to assume that all the logic cells in a chain together must use no more than the legal number of LAB-wide signals for one LAB. This guarantees the fitter will be able to divide the chain into legal LABs without inserting feedthroughs.

If a placement is sent into Quartus from a 3rd party placement or floorplanning tool via location constraints, and the placement of the logic violates LAB-wide signal limits, Quartus will give a no-fit (i.e. Quartus will not attempt to repair the placement). If a placement is sent in via “preferred locations”, Quartus will attempt to repair an illegal placement.

In Stratix and Cyclone the signals connected to the .clk, .ena, .aclr, .aload, .sclr, .sload and .inverta ports on logic cells are all LAB wide.

The method by which the number of LAB-wide signals of a given type required by a group of logic cells in a LAB is determined is a key concept. A logic cell port connected to a “regular” signal consumes one LAB-wide signal of that type. The same signal used in inverted form counts as a

separate LAB-wide signal. So, if we have clock used by one logic cell in a LAB and !clock used by another logic cell, we require two LAB-wide clock lines. VCC and GND connected to a logic cell port also require a LAB-wide signal line (set to GND or VCC). There are situations where some ports on a logic cell are left unconnected. If this is the case, then the unconnect may or may not count as a use of a LAB-wide signal line, depending on the port and the exact logic cell configuration. Table 1 below shows how unconnected ports on a logic cell are handled.

Table 1: Whether or not logic cell ports are considered to be "used" when unconnected, and the signal to which such "considered used" ports are connected.

Unconnected Signal	Register Used (.regout connected)	Register Unused (.regout unconnected)
.aload	Not Used	Not Used
.aclr	Used (GND)	Not Used
.sload	If (synch_mode = on), used (GND)	Not Used
.sclr	If (synch_mode = on) used (GND)	Not Used
.ena	Used (VCC)	Not Used
.clk	Used (GND)	Not Used
.inverta	Used (GND)	Used (GND)

In all the rules below, it is important to count unconnected signals that are considered "used" LE ports, as defined by Table 1.

1. All **.clk** and .**ena** signals on a logic cell are paired to form an LE clock (see Figure 1). In any LAB, there can be no more than 2 distinct LE clock pairs.
    - The same **.clk** signal with 2 different **.ena** signals counts as 2 LE clock pairs.
    - The same **.ena** signal with 2 different **.clk** signals counts as 2 LE clock pairs.

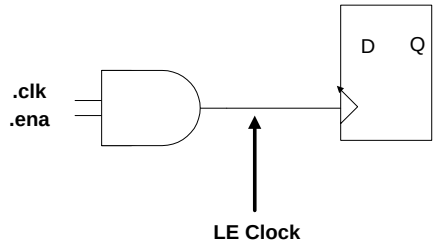


Figure 2: Grouping of clock and clock_enable pairs to form LE clocks.

Table 2 provides an example of how clock + clock enable pairs are created. The example shows 2 sets of clocks and 2 sets of clock enables corresponding to any 2 logic cells that use their registers. clk (#1) and ena (#1) belong to logic cell #1. clk (#2) and ena (#2) belong to logic cell #2. Unconnected **.ena** ports are set to VCC, as Table 1 specified. A, B, C and D correspond to signal nets.

Table 2: : Example of how the clock and clock enable of a logic cell are paired to form LE clocks. Both logic cell #1 and #2 use their regisers (**.regout** is connected).

.clk (#1)	**.ena (#1)**	**.clk (#2)**	**.ena (#2)**	**Number of LE Clock pairs**	**Pairs**
A	B	A	C	2	(A, B) (A, C)
A	C	B	C	2	(A, C) (B, C)
A	B	A	B	1	(A, B)
A	B	C	D	2	(A, B) (C, D)
A	Unconnected	B	Unconnected	2	(A, VCC) (B, VCC)
A	Unconnected	A	Unconnected	1	(A, VCC)
A	Unconnected	A	B	2	(A, VCC) (A, B)
A	B	Unconnected	Unconnected	1	(A, B) (GND, VCC)

2. A maximum of 2 distinct signals can be connected to the **.aclr** ports. (Recall from Table 1 that an unconnected **.aclr** on a logic cell counts as a GND signal if a logic cell uses its register.)
3. A maximum of 1 distinct signal can be connected to the **.aload** ports.
4. If any LE in a LAB uses **.**aload:
   - All logic cells in that LAB that use **.aload** must use the same **.aclr** signal as the first cell
   - All logic cells in that LAB that do not use **.aload** must use the same **.aclr** signal.
5. A maximum of 1 distinct signal can be connected to the collection of **.inverta** ports. As seen from Table 1, the .inverta signal affects all Lcells (since it is always connected). No 4-input combinational cell can be placed in a LAB that uses inverta unless it uses the exact same inverta.
6. A maximum of 1 distinct signal can be connected to the collection of **.sload** ports and a maximum of 1 distinct signal to the collection of **.sclr** ports.
   - A logic cell that has *synch_mode* = *off* may be placed in the same LAB as another logic cell which does make use of its **.sload** or **.sclr** ports
   - A logic cell that has *synch_mode* = *on* can only be placed in a LAB that uses the same sload or sclr as that cell or in a LAB that does not use its sclr and sload.

Table 3: Some examples of when .sload and .sclr are considered used/free.

	.sload	.sclr	.sload used	.sclr used
Case #1 (synch_mode = off)	Unconnected	Unconnected	Not Used	Not Used
Case #2 (synch_mode = off or on)	GND	GND	Not Used	Not Used
Case #3 (synch_mode = on)	Net_A	Unconnected	Used	Used (set to GND)
Case #4 (synch_mode = on)	Unconnected	Net_A	Used (set to GND)	Used
Case #5 (synch_mode = on)	Net_A	Net_B	Used	Used
Case #6 (synch_mode = on)	VCC	Unconnected	Used	Used (set to GND)

Table 4: Examples of legal-LAB and illegal combinations of two logic cells using .sclr and .sload.

.sload (LE #1)	.sclr (LE #1)	.sload (LE #2)	.sclr (LE #2)	Form legal LAB?
Net_A	Net_B	Net_A	Net_B	Legal
Net_A	Net_B	Net_A	Net_C	Illegal
Net_A	Net_B	Net_B	Net_A	Illegal
Net_A	Net_B	Unconnect or GND	Unconnect or GND	Legal
Net_A	Net_B	Net_A	Unconnect or GND	Illegal
Net_A	Net_B	Unconnect or GND	Net_B	Illegal

5.4 LAB Control Routability Constraints

The following rules ensure that all the control signals required within a LAB can be routed into the LAB. Any logic cell port that is connected to a signal net (non-VCC, non-GND) and is not routed on a dedicated global routing resource requires that signal to be routed into the LAB, to the appropriate LAB-wide input port. If a signal is required in true and complemented form in a LAB, it must be routed in twice. If one of the logic cell ports is connected (or considered connected by Table 1) to either VCC or GND then it may or may not require a VCC or GND signal to be routed into the LAB to the appropriate LAB-wide input port. This depends on whether or not there exists a LAB-wide tie-off for that signal. Finally, signals connected to the **.clk** and **.aclr** ports do not require a LAB-wide input port if they are routed on a dedicated global network, since they have a dedicated port into the LAB in this case. Table 5 below shows when a LAB-wide input port is

required for each logic cell port when the port is connected to a signal, and when it is connected (or considered to be connected by Table 1) to either VCC or GND.

Table 5: When LAB-wide input ports are required.

LE Port	Connected to Signal	Connected to VCC	Connected to GND
.aclr	Needs LAB-wide input **if** not on global routing	Needs LAB-wide input	Doesn't need input
.aload	Needs LAB-wide input	Needs LAB-wide input	Doesn't need input
.clk	Needs LAB-wide input **if** not on global routing	Needs LAB-wide input	Needs LAB-wide input
.ena	Needs LAB-wide input	Doesn't need input	Needs LAB-wide input
.sload	Needs LAB-wide input	Doesn't need input	Doesn't need input
.sclr	Needs LAB-wide input	Needs LAB-wide input	Doesn't need input
.inverta	Needs LAB-wide input	Needs LAB-wide input	Doesn't need input

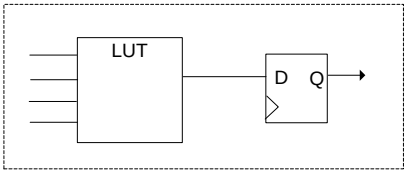
The rules below apply to instances of the signal only if they need a LAB-wide input according to Table 5. For example, a global .aclr signal shouldn't be counted when evaluating the rules below.

1. The sum of the signals requiring LAB-wide inputs ports must be no more than 6. The following LE ports can require LAB-wide input ports, as shown in Table 5.
    - **.clk**
    - **.ena**
    - **.aload**
    - **.sload**
    - **.aclr**
    - **.sclr**
    - **.inverta**
2. If a **.sload** signal requires a LAB-wide input port, then only 1 distinct **.ena** signal can be used in that LAB.
3. If an **.aload** signal requires into a LAB, then only 1 distinct **.clk** signal can use a LAB-wide input port in that LAB.
4. The number of **.aclr** plus **.sclr** signals that can be routed into a LAB is 2.
5. The total number of signals that can be routed into a LAB (not including cin or signals that are generated by LEs inside the LAB) is 30 for Stratix, and 26 for Cyclone. For good routability, however, it is best not to go above 26 distinct signals for Stratix, and 23 for Cyclone, that need to be routed into a LAB (not including clk and aclr, which will usually be routed on special, global interconnect).

5.5 Recommendations for Good Fitting Results

1. The **.inverta** port is very restrictive since it hurts placement of other combinational cells in the same LAB where it is used. It also forces the carry chain to start at LE #0 for most chains that use the **.inverta** port. It is best to use the **.inverta** port only on large chains. Using it on many small carry chains of size less than 1 LAB may make it difficult to fit circuits with very high logic utilization.
2. The **.sclr** port on a logic cell should be used with discretion. The Stratix architecture supports a maximum of 1 **.sclr** per LAB. Improper use of this port could result in a severe degradation in logic utilization, fitting, and circuit speed. For example, in the past **.sclr** has been used to generate a 5-input function in one logic cell. This reduces the number of logic cells needed to implement a circuit, and hence may appear to be a good idea from a synthesis perspective. However, once a logic cell that uses **.sclr** is placed into a LAB, that LAB typically cannot have any other registered logic cells packed into it unless they use the same **.sclr** value or have synch_mode = off. This can result in a 90% reduction in logic density in the worst case, since only one logic cell can be placed in a LAB, rather than 10. Hence undisciplined use of **.sclr** will lead to very bad fitting and speed, as the fitter will have little flexibility on which logic cells can be grouped together into LABs.

   .sclr should be used only to implement real synchronous clear signals in a user circuit, since there tend to be a small number of high-fanout synchronous clear signals in this case. Hence the fitter will not be unduly constrained by the synchronous clear signals. Using sclr on all the logic cells in a chain is also a reasonable use of this port.
3. The same restrictions for **sclr** as described above exist for **sload**. **Sload** should only be used for high fanout signals.
4. Clock enables (**.ena**) can be used somewhat liberally, since two distinct **.ena** values are allowed per LAB, and this greatly improves the fitter's ability to cope with a large number of **.ena** values vs. having one per LAB. Nonetheless, an excessive number of clock enables (**.ena**) can make it difficult to achieve logic utilization above 90%. We find that circuits that use thousands of distinct **.ena** signals typically can often achieve only 90% or so logic utilization before the fitter can no longer divide the logic cells into legal LABs. Circuits that use **.ena** more sparingly routinely achieve logic utilizations over 99%. Hence some care to limit the number of distinct **.ena** signals (particularly low-fanout signals) is merited.
5. Don't use the **register_cascade_mode = on** unless you have developed a detailed placement algorithm. These modes are used to group a look-up table and an unrelated register together to save logic cells. It is best to let the fitter perform this grouping, since it has a much better idea of where different registers and look-up tables should naturally be placed, what the relative wiring congestion around a region is, and so on. Grouping registers and look-up tables together using these modes can make it much more difficult to successfully fit a circuit, and reduce the achievable circuit speed.
6. **sum_lutc_input = qfbk** can now be used to pack a register with its fan-in look-up table, while the other operation modes can be used to pack a look-up with its fan-out register. The figure below illustrates the choice.



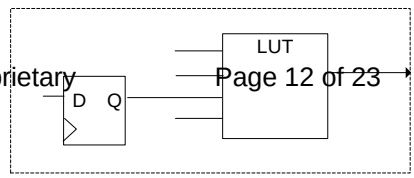


Figure 3: Register & look-up table grouping into logic cell options.

The decision of which way to pack should be made based on which connection appears to be most timing critical, and routability concerns. Bear in mind the following points:

- The connection directly from a look-up table to a register in the same logic cell (or vice versa) is the fastest connection in a Stratix device.
- Dual-output cells are more difficult to route than single-output cells. Hence you should try to group registers and look-up tables together such that only one output is created from the logic cell. This means that if there is a connection from a look-up table to a register, where the register is the only fanout of the look-up table, sum_lutc_input = datac is a good choice to combine the pair. Similarly, if there is a connection from a register to a look-up table, where the look-up table is the only fanout of the register, sum_lutc_input = qfbk is a good choice to combine the pair.
- There is a slight routability advantage to using sum_lutc_input = datac instead of sum_lutc_input = qfbk. As well, usually logic cells with sum_lutc_input = qfbk will also use sload = VCC to load the register from .datac. Such cells are slightly harder to fit because of their use of the LAB-wide synchronous load signal. For both these reasons, it is best to use a sum_lutc_input = datac logic cell when there is a tie on the other points.
- The Quartus fitter will automatically make a decision as to whether or not a register and a look-up table should be merged using qfbk mode or not, so leaving registers separate from look-up tables is a good policy if you're not very sure of your algorithm and its timing estimates.

6. DSP Block

The DSP Block (also referred to as the MAC Block) is a new block that implements basic DSP functions. These functions are multipliers, multiply-accumulators, multiply-adders, as well as a special input shift register. This block is not implemented using LEs - instead a dedicated block approximately 8 LABs by 2 LABs in size implements these functions.

Each DSP Block is composed of four 18-bit multipliers that can feed adders. Some of these adders can be configured to implement an accumulator. It should be noted that these adders cannot be used independent of the multipliers – the multipliers must always be used. The four 18-bit multipliers can be configured to instead implement eight 9-bit multipliers, or a single 36-bit multiplier without any external logic. The Stratix MAC WYSIWYG document describes the various configurations and their options in more detail.

There are two columns of DSP blocks in every Stratix device, with the number of blocks per column increasing for the larger devices (the smallest device has a total of 6 DSP blocks, while the current largest device has a total of 22 DSP blocks). Cyclone does not contain DSP blocks.

Each column of DSP blocks sits between columns of LABs, and is between a column of SEABs and a column of MEABs. The DSP columns are 3 LABs away from the column of MEABs so that a single H4 line can be used to connect a RAM to a DSP block, which is expected to be a common application.

6.1 Instantiating a DSP Block

DSP blocks can be instantiated by either instantiating the MAC WYSIWYGs directly, or by black boxing one of three megafunctions that implement DSP blocks. Instantiating the MAC WYSIWYGs directly provides more control and information since it does not have to be treated as a black box. However, instantiating the megafunctions will be less complex and provides greater user control in Quartus (for example, user can choose to implement a megafunction in LEs instead of DSP blocks).

The megafunctions are as follows:

1. The **LPM_MULT** megafunction that implements a single multiplier.
    - Has a special parameter to specify that the multiplier should be implemented in a Stratix DSP block ("*DEDICATED_MULTIPLIER_CIRCUITRY*"). Parameter can also be set to "AUTO" which indicates Quartus will determine whether to implement the logic in LEs or DSPs.
    - Does not provide full control of implementation since there are no parameters to specify exactly which registers can be used, or to allow dynamic sign control signals, or different sign representations for each operand. For these cases, it is better to instead use an ALTMULT_ADD with NUMBER_OF_MULTIPLIERS=1.
2. The **ALTMULT_ACCUM** megafunction that implements a multiplier feeding an accumulator.
    - This megafunction can implement a multiplier and an accumulator of any width by adding any necessary external logic.
    - Implementation of this megafunction in LEs is supported. As with LPM_MULT, the *DEDICATED_MULTIPLIER_CIRCUITRY* parameter controls the implementation and has options *YES*, *NO*, and *AUTO*. The default is *AUTO* if left unspecified.

3. The **ALTMULT_ADD** megafunction that implements one or more multipliers feeding an adder.
    - If NUMBER_OF_MULTIPLIERS=1, then this megafunction will implement a single multiplier and will bypass the adder. Unlike lpm_mult, this provides full control of all registers and dynamic controls.
    - This megafunction now supports multiply-adders that do not fit in a single DSP block. Multiple DSP blocks will be stitched together with logic in LEs as necessary to implement any functions that will not fit in a single DSP block. However, as before, this megafunction is still restricted to a maximum of 4 multipliers feeding an adder.
    - Implementation of this megafunction in LEs is now supported. As with LPM_MULT, the *DEDICATED_MULTIPLIER_CIRCUITRY* parameter controls the implementation and has options *YES*, *NO*, and *AUTO*. The default is *AUTO* if left unspecified.

6.2 Estimating DSP Block Usage

This section describes how to estimate the eventual usage of DSP Blocks in a particular design. In most cases, this estimate will be the same as the actual usage and only differs in cases where signal conflicts may result in higher usage.

A DSP block 9-bit element (DSP element) is used to describe the smallest component in a DSP block. A DSP element is essentially a 9-bit multiplier, where one of these elements is needed to build a 9-bit multiplier, and two of these are needed to build an 18-bit multiplier. Four 9x9 multipliers are not needed to build an 18x18 since the hardware itself supports 18x18 multipliers, and 9x9 multipliers are derived by splitting each of the 18x18 multipliers in half.

By counting the number of these elements used for every mode and width, it is possible to determine the number of DSP blocks required since 8 DSP elements of the same mode and width can fit in a single DSP block. Since multipliers of different mode or width cannot be combined in the same DSP block, it is important to count their usage separately.

The following table shows the DSP element usage for every possible mode and width combination.

		Width	
Mode	**9x9**	**18x18**	**36x36**
Multiplier Only	1	2	8
Multiply-Accumulator	X(1)	4	X(2)
Two-Multipliers Adder	2	4	X(2)
Four-Multipliers Adder	4	8	X(2)

(1) The multiply-accumulator mode only supports 18-bit multipliers. All multipliers smaller than 18-bits (including those smaller than 9-bits) will be use up an 18-bit multiplier.

(2) Multipliers larger than 18x18 and up to 36x36 can only be implemented as independent multipliers since an entire DSP block is used to implement 36x36 multipliers. For all other modes, the accumulator or adder will be implemented in LEs while only the multiplier portion is implemented in a DSP block set to 36x36 mode.

To determine the number of DSP blocks required for a particular mode-width combination, count the number of DSP elements used, divide the number by 8, and then round *up* the number. Then

the total number of DSP blocks needed can be derived by adding the number needed for each mode-width combination.

The following is an example. Suppose a design has the following DSP functions:

a) Seven 7x5 simple multipliers (not feeding an adder or accumulator)

b) Ten 8x8 simple multipliers

c) Five 16x16 simple multipliers

d) Three sets of two 14x12 multipliers feeding an adder (i.e. 6 multipliers feeding 3 adders)

Each of these will lead to the following DSP element counts:

a) 7 DSP elements for 9x9 Multiplier Only Mode

b) 10 DSP elements for 9x9 Multiplier Only Mode

c) 10 DSP elements for 18x18 Multiplier Only Mode (= 5 x 2)

d) 12 DSP elements for 18x18 Two-Multipliers Adder Mode (= 3 x 4)

So for each mode-width combination, the following lists the number of DSP elements needed:

1) 9x9 Multiplier Only Mode: 17 DSP elements (= 7 + 10)

2) 18x18 Multiplier Only Mode: 10 DSP elements

3) 18x18 Two-Multipliers Adder Mode: 12 DSP elements

For each mode-width combination, the following lists the number of DSP blocks needed

1) 9x9 Multiplier Only Mode: ceil(17 / 8) = 3 DSP blocks needed

2) 18x18 Multiplier Only Mode: ceil(10 / 8) = 2 DSP blocks needed

3) 18x18 Two-Multipliers Adder Mode: ceil(12 / 8) = 2 DSP blocks needed

Adding the number of DSP blocks needed gives us 3+2+2 = 7 DSP blocks needed for this design.

It is important to note that 3 of these DSP blocks are only partially filled and so it is possible for example to add another 9x9 simple multiplier without requiring any more DSP blocks. This should be taken into consideration when doing resource balancing between DSP blocks and LEs. For example, in this case, it may be better to convert one of the 9x9 multipliers into LEs since this will reduce the number of DSP blocks required to 6 since one of the DSP blocks contains only one 9x9 multiplier.

Providing the detailed DSP block usage (i.e. per mode and width combination) to the user should greatly improve the usability of the DSP blocks.

6.3 Input Shift Register (Scan Chain)

The scan chain of the DSP block is basically a shift register implemented out of the input registers of the multipliers. The output of each multiplier's input register can be configured to feed the input of the next multiplier through dedicated routing. The last multiplier in a DSP block can feed the first multiplier in the DSP block immediately below it in the same column. However, the last DSP block in a column cannot not use its scanout port, and the first DSP block in a column cannot be driven by the scanout port of another multiplier through dedicated routing.

IMPORTANT restriction: *Due to hardware limitations, hold violations can occur if different clock or clock enable signals are used for registers that are part of the same scan chain. For this reason, only scan chains which have the same clock and clock enable signal for all registers should be inferred. Otherwise, Quartus II will give a compilation error.*

The scan chains can be used in all modes except the 36-bit multiplier mode. For this case, external shift registers will have to be implemented.

It is possible for the scanout port of a multiplier to drive regular routing. This is accomplished by "stealing" the outputs of the last multiplier in a DSP block. When implementing independent multipliers, this means that only three 18-bit multipliers will be usable in a given DSP block. Similarly, when implementing an 18-bit accumulator, only one 18-bit accumulator can be implemented per DSP block. However, for both adder modes, no multipliers are lost since they do not use their DSP block outputs (they drive the adders instead). So for these modes, there is no change if using regular routing or dedicated routing.

The decision to use regular routing versus dedicated routing will be made by the fitter based on placement constraints and other considerations.

Input shift registers are commonly used in DSP filter applications, such as a FIR filter. It can be shown that a 4-tap FIR Filter can be implemented in a single DSP block using this feature.

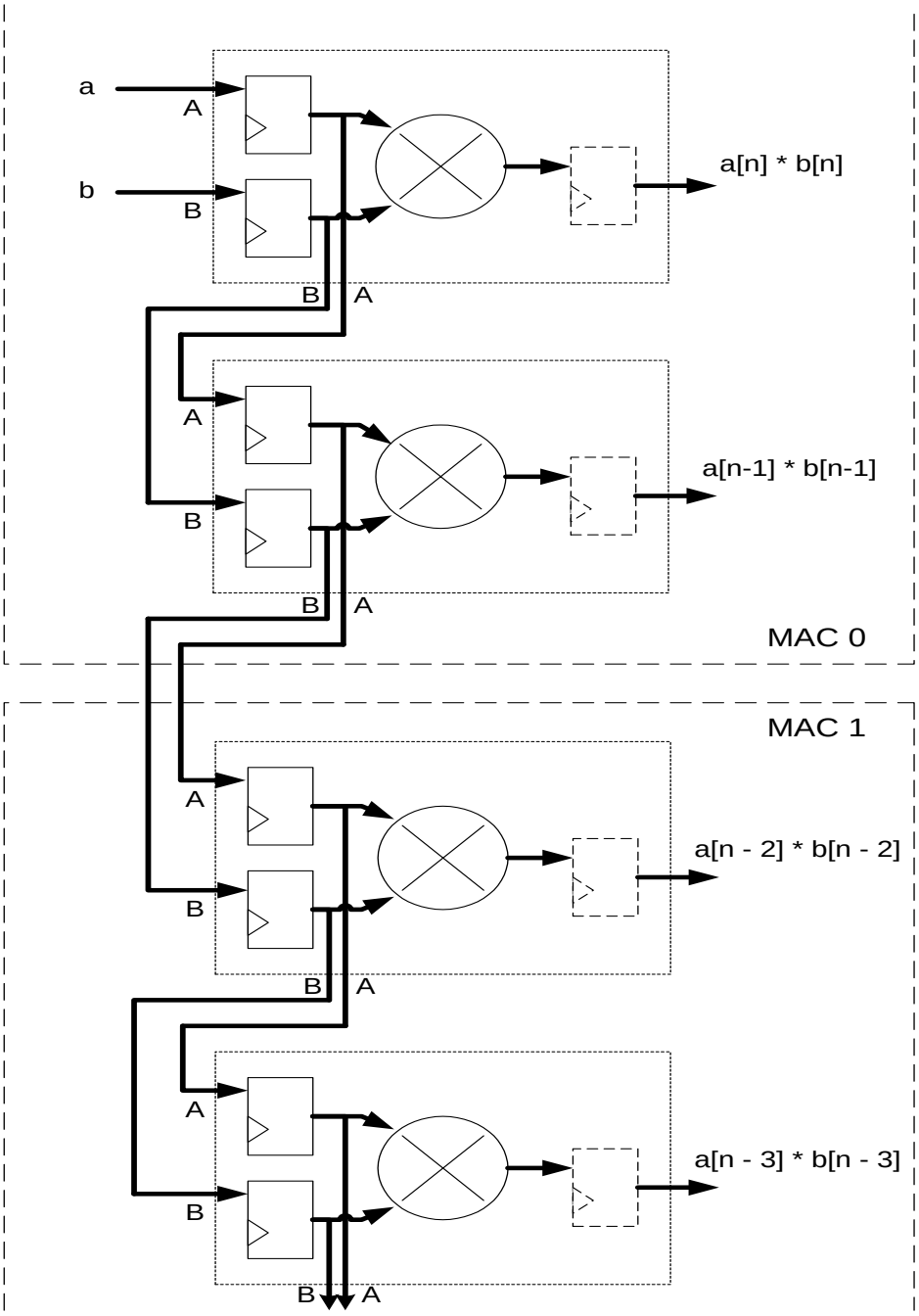


Figure 4: Multipliers Using Scan Outputs

6.4 Use of DSP Blocks versus LEs

Since the DSP blocks are essentially blocks of logic, they can be interchanged with LEs to implement the same function. The advantages for using a DSP block over LEs are the large size savings, and the high performance advantage. However, since there is a limited number of DSP blocks in any given device, the user may choose to implement some multipliers in the DSP blocks, and some multipliers using regular LEs.

Making the decision between using DSP blocks and LEs is currently not completely handled by Quartus. For this reason, the first and easiest step is to have options in the synthesis tool where the user can choose to have all DSP blocks to be inferred, or none of them, or individual DSP functions in the design (through signal-leval and/or module level parameters). This allows the user to then manually change which blocks should be implemented in DSP blocks, and which should be implemented in LEs at the synthesis tool level. However, the optimal solution is for the synthesis tool to pick whether to use a DSP block or LEs based on the number of available DSP blocks on the device, and the criticality of that particular portion of the design.

6.4.1 DSP Block Resource Balancing

Quartus now has support for DSP block Resource Balancing. The goal of this feature is to prevent no-fits, and to select a better device when AUTO device selection is enabled.

If the DSP blocks used in the design will not fit in the selected device, then Quartus will convert the smaller multipliers into LEs until the DSP usage is legal. If the DSP usage is legal, then this feature will have no effect. Therefore, if the synthesis tool does resource balancing itself and only infers a legal number of DSP blocks, then Quartus will not do any conversions.

Resource balancing will try to honor the DEDICATED_MULTIPLIER_CIRCUITRY parameter if it was specified. If the parameter was set to “YES” then the associated instances will not be converted by resource balancing and will be implemented in DSP blocks. Similarly, if set to “NO” then the associated instances will always be implemented in LEs. Only if the parameter is not specified, or set to “AUTO”, will Quartus try to do resource balancing with the instance.

A Quartus assignment, *DSP_BLOCK_BALANCING*, allows the user to control resource balancing. For example, the user can force a conversion to LEs by specifying the DSP_BLOCK_BALANCING assignment on the instance to “Logic Elements” – if the DEDICATED_MULTIPLIER_CIRCUIRTY option was specified differently, then it will be overridden in this case and a warning will be issued that the parameter is being ignored. Additionally, the user can disable resource balancing entirely by setting the option to “Off”. By default, resource balancing will be enabled.

6.5 Inferring DSP Blocks

There are some issues to consider when inferring multipliers from a user’s design:

1. Recognizing registers and scan chains from a user’s design
    - Since the DSP block has several registers at various stages, trying to absorb registers into the DSP block itself would in most cases be preferred over implementing the registers in LEs. However, since the DSP block can potentially operate at very high speeds and may not be part of the critical path of a design, it may be desirable to implement these registers externally to help other paths. Implementing this as a user option (i.e. option to turn on/off register absorption) would be one way of resolving this.
    - Not only would simple registers have to be recognized and absorbed into the DSP block, but also scan chains (i.e. shift registers on the multiplier inputs) will have to be recognized and absorbed. (See important restriction above).
    - Quartus now supports packing registers into the DSP block input, output and scan chain registers during fitting. However, the fitter will not automatically pack registers into the DSP block pipeline registers, so synthesis must infer those registers for them to be inferred.
2. Recognizing dynamic control signals

- Recognizing a dynamically changing sign in the user's design, which the DSP block supports, may prove to be difficult to implement. The basic case is to treat these signals as constants and recognize either signed or unsigned data. However, it is still important to be able to infer both signed and unsigned logic and set the representation appropriately in the megafunctions.
- Other dynamic control signals such as addnsub and zero_acc, though not as difficult to recognize as a dynamic sign, will also require some work. The basic case again is to treat these signals as constants.
- Currently there doesn't seem to be any easy way to infer the accumulator overflow signal from generic HDL, so this signal can be ignored.

6.6 Rules for DSP WYSIWYGs

The following rules apply to MAC slices (i.e. a MAC_OUT driven by one or more MAC_MULTs). NOTE: if you infer megafunctions instead of WYSIWYGs, then this section is not relevant.

1. All signa signals must be driven by the exact same source, and all signb signals must be driven by the exact same source.
2. All signa clock and clear parameters must be set identically for all MAC_MULTs and MAC_OUTs in the same slice. Similarly for the signb clock and clear parameters.
3. The mac_mult.data_out[] port must be connected and can only drive a data port on a mac_out. Similarly, the dataa, datab, datac, and datad ports on a mac_out can only be driven by a mac_mult.data_out[].
4. The output registers are required for the MAC_OUT when in "accumulator" mode.
5. Only multipliers with the same resulting width can be added together.
6. The clk[], aclr[], and ena[] signals must be identically connected for all cells in the same MAC slice.
7. The mac_mult.scanout[] ports cannot be used when implementing a 36-bit multiplier: external registers implemented in LEs must be used if a scan chain is desired.
8. Only signals available in a particular mode can be connected. For example, in one-level-adder mode, the zero_acc and overflow signals cannot be connected since they are only available in the accumulator mode.
9. The pipeline register for a given dynamic control signal can only be used if its corresponding input register is also used (i.e. the pipeline register cannot be used as the only register on the input – use the input register in that case).

6.7 Recommendations for DSP Megafunctions

1. Use single altmult_add megafunction instead of an LPM_MULT and LPM_ADDSUB.
2. Use altmult_add megafunction instead of LPM_MULT if DSP specific features like scanout signals, dynamic sign control, etc are needed. Even for cases where registers are used, it may be a good idea to use the altmult_add megafunction anyway since it gives you greater control of the register locations (lpm_mult only allows you to specify a pipeline parameter, but not where those registers go). In this case, set the number_of_multipliers parameter to 1.
3. Do not try to use scanouts of multipliers to feed regular logic since this places severe restrictions on the placement of the multipliers (they can only be placed at the last

multiplier slot, and in some cases, a multiplier may become unusable). Only use if scanouts feed data input of another multiplier (though inversions are allowed).

4. Specify the actual width of a data port using the parameters instead of “faking” it by using a fixed width and grounding some bits. The megafunctions internally generate extra logic to handle larger multipliers.
5. Have option for user to specify whether to implement multipliers/adders/accumulators in DSP blocks or in LEs. Each megafunction has parameter to do the same.
6. Need some system where user can specify only some multipliers/adders/accumulators are implemented in DSP blocks, while others are implemented in LEs. Since a limited number of DSP blocks per chip, user may want to trade off between LEs and DSPs based on some criteria.
7. Absorb registers into DSP block where feasible. Also need an option here to disable this if user decides that registers are better left in LEs. Quartus will also try to absorb registers into DSP blocks based on user options.
8. Recognize dynamic control signals whenever possible (i.e. addnsub, accum_sload). Saves considerable logic if implemented in DSP block instead of LEs.
9. The signed/unsigned representation of the outputs is dependant on the signed/unsigned representation of the inputs (i.e. sign signals) and whether doing addition or subtraction. If either input is signed, then the multiplier result is considered signed. For the adders, if either the multiplier result is signed, or if doing subtraction, then the adder result is considered signed. This means that the output’s sign representation can change dynamically if the sign signals are dynamic, or if the addnsub signals are dynamic.
10. As with all signals that are constant, they should be set to GND or VCC directly instead of using an intermediate Lcell.
11. The ALTMULT_ADD supports from 2, 3, or 4 multipliers feeding an adder, or just 1 simple multiplier. This megafunction will stitch together multiple DSP blocks in order to implement functions that do not fit in a single DSP block. For all cases, the multiplier input widths must be the same for all multipliers feeding the same adder – however, note 12 explains how to get around this restriction for some cases.
12. For cases where the user does an addition of multipliers with different input sizes, these can still be supported by ALTMULT_ADD if the multiplier inputs are padded appropriately as follows:

    To pad an input with unsigned representation, add the extra bits as upper bits (i.e. after the original MSB bit) and tie them to GND. To pad an input with signed representation, add the extra bits as upper bits, and have each of them fed by the original MSB bit (i.e. sign bit) – this is necessary to maintain the sign of the signed number.

    Since all dataA multiplier inputs need to be of the same width, simply pad any dataA multiplier inputs that are less than the largest dataA width. The same needs to be done for dataB multiplier inputs.

    For example, if we have “result[31:0] = a[15:0] * b[7:0] + c[12:0] * d[14:0]” and all numbers are signed, then this can not directly be implemented in a DSP block since the input widths are all different. But we can map the multiplier input c[12:0] to have 3 extra upper bits c[15:13] all driven by c[12], and we can map b[7:0] to have 7 extra upper bits b[14:8] all driven b[7], then all multipliers are 16-bit x 15-bit and so can be implemented with ALTMULT_ADD. If the numbers were unsigned instead, then the extra upper bits would just be tied to GND instead.

6.8 Example

The following is an example of a simple design input by a user, its corresponding high-level netlist, and the resulting VQM file that can then be fed to Quartus. The design is written in Verilog and implements two multipliers feeding an adder, with a dynamical control to choose between addition and subtraction.

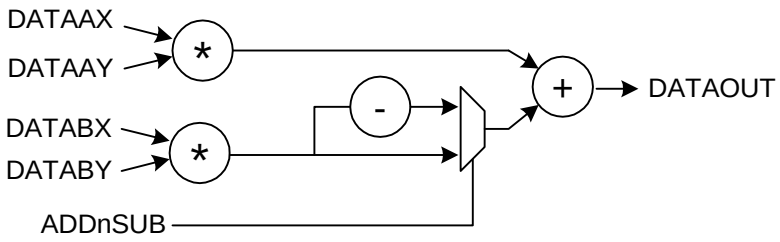
```
module mult_test (DATAOUT, DATAAX, DATAAY, DATABX,
      DATABY, ADDnSUB);

      output [16:0] DATAOUT;
      input [7:0] DATAAX, DATAAY, DATABX, DATABY;
      input ADDnSUB;

      wire [15:0] multa = DATAAX * DATAAY;
      wire [15:0] multb = DATABX * DATABY;
      assign DATAOUT = ADDnSUB ? (multa + multb) :
                                  (multa - multb);

endmodule
```

The resulting high-level netlist will approximately be as follows:



The VQM file generated will be as follows:

```
module mult_test (DATAOUT, DATAAX, DATAAY, DATABX, DATABY,
                  ADDnSUB);
      output [15:0] DATAOUT;
      input [7:0] DATAAX, DATAAY, DATABX, DATABY;
      input ADDnSUB;
            altmult_add mult_add_0 (
                  .dataa({DATAAX, DATABX}),
                  .datab({DATAAY, DATABY}),
                  .addnsub1(ADDnSUB),
                  .result(DATAOUT)
            );
      defparam mult_add_0.NUMBER_OF_MULTIPLIERS = 2;
      defparam mult_add_0.WIDTH_A = 8;
      defparam mult_add_0.WIDTH_B = 8;
      defparam mult_add_0.WIDTH_RESULT = 16;
endmodule /* mult_test */
```