GreenPak4 HDL Place-And-Route User Guide

Andrew Zonenberg
azonenberg@drawersteak.com

May 22, 2017

Abstract

This document is the primary reference manual for *gp4par*, Andrew Zonenberg's place-and-route tool for *Silego GreenPak4* devices. As of this writing, the toolchain is **not** officially supported by *Silego*. It is under active development and should be considered alpha quality.

Contents

1 Revision History

- May 22, 2017: [in progress] Initial draft

2 Introduction

2.1 Architecture Support

This guide applies to all *Silego GreenPak4* devices, however not all are fully supported by the current software version.

- *SLG46620V*: All features described in this document are supported.
- *SLG46621V*: All features described in this document are supported, but not as well tested.
- *SLG46140V*: Preliminary support, many features are not yet implemented.

2.2 Coding Examples

The coding examples in this guide are accurate as of the date of publication. The most up-to-date version of this document, as well as source code for the place-and-route tool, may be found on GitHub at https://github.com/azonenberg/openfpga/.

2.3 Syntax Examples

The syntax examples in this guide show how to use constraints and options. The examples are comprehensive; only the described syntax for a particular constraint or option is guaranteed to work.

All Verilog attributes and values are case sensitive unless otherwise noted.

2.4 Acronyms

Acronym	Meaning
HDL	Hardware Description Language.
IOB	Input/Output Buffer.
PAR	Place And Route.
PTV	Process, Temperature, and Voltage.
RTL	Register Transfer Level.

2.5 Formatting

This guide uses several distinct text styles for:

- Proper names, such as *Silego GreenPak4*;
- Port and parameter names, such as `AUTO_PWRDN`;
- References, such as `GP_IBUF`;
- Signal names, such as `out`;
- Property values, such as `"YES"`;
- Numeric literals, such as `6'b101010`.

2.6 Support

If you have questions, comments, suggestions, or wish to contribute to the project please join our IRC channel (##openfpga on Freenode). This is the sole support forum available at this time.

3 Synthesizing a Netlist

3.1 Design Flow

gp4par is **not** a synthesis tool and cannot be run directly on HDL source code. Your HDL must be synthesized to a JSON netlist by a separate tool before *gp4par* is be invoked. The recommended synthesis tool is *Yosys*, which may be obtained from the *Yosys* website, http://www.clifford.at/yosys/. The flow of data between components is shown in Figure 1.

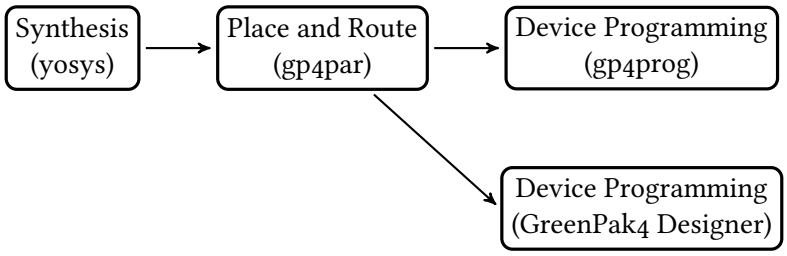

Figure 1: Data flow between toolchain components

gp4par is being developed in tandem with *GreenPak* support in *Yosys*. Most of the *GreenPak* features are only available in the latest development version of *Yosys* and have not made it into a stable release yet. We recommend use of the latest *Yosys* from GitHub for testing.

3.2 Synthesis Example

A simple synthesis script for *Yosys* is shown in figure 2. The script synthesizes a single Verilog source file `Blinky.v`, with a top-level module `BlinkyTop`, to the netlist `Blinky.json` and targets the *SLG46620V*. This script is only a starting point and may be customized as needed. This document does not cover synthesis commands; please see the online documentation for *Yosys* for command documentation.

```
1 #!/usr/bin/env yosys
2 read_verilog Blinky.v
3 synth_greenpak4 -top BlinkyTop -part SLG46620V -json Blinky.json
```

Figure 2: Example synthesis script

4 Toolchain Limitations

4.1 *gp4par* Device Limitations

The following device features are not supported by *gp4par* for any device as of this writing:

- Counters: Delay, edge detector, PWM, wake-sleep mode, counter cascading. All clock sources other than on-chip oscillators.
- ADC
- DAC: Inputs from DCMP.
- Wake-Sleep
- DCMP/PWM: PWM mode. Inputs from ADC.
- SPI: ADC buffer mode, parallel output to fabric, clock synchronization.

In addition, the following *SLG46140V* device features are not yet supported by *gp4par*:

- Comparators
- ADC
- DAC
- PGA
- DCMP/PWM
- Slave SPI
- Wake-Sleep
- Voltage reference
- Ring oscillator
- RC oscillator
- Dedicated DFF cells
- Counters
- Combination function macrocells
- Power detector

4.2 *Yosys* Verilog Inference Limitations for GreenPAK

The following device features cannot be inferred from behavioral Verilog, but are supported if primitives are directly instantiated.

- Latches with set/reset
- Counters with input divider, clock enable, or upward/adjustable direction
- Voltage reference blocks must be explicitly instantiated. (A future version of the toolchain will support inferring *GP_VREF* blocks when driving a comparator via an external reference voltage or DAC.)

5 *gp4par* HDL Constraints

Constraints may be entered by Verilog attributes or an external Physical Constraints File (.pcf) file. In the event of a conflict, the PCF takes precedence over constraints in the Verilog. If two constraints in the same PCF file conflict, the later constraint takes precedence.

5.1 Verilog attributes

The general format of a constraint with name `FOO` and value 42 applied to the register `foobar` is shown in Figure 3.

```
1 (* FOO=42 *)
2 reg[3:0] foobar = 0;
```

Figure 3: Example Verilog attribute constraint

5.2 PCF constraints

gp4par PCF files used a syntax similar to that of the industry standard Synopsys Design Constraints (SDC). Only constraints are supported; arbitrary TCL cannot be used.

The general format of a constraint with name `FOO` and value 42 applied to the register `foobar` is shown in Figure 4.

```
1 # set some random attribute
2 set_foo foobar 42
```

Figure 4: Example PCF constraint

5.3 Counter Extraction (COUNT_EXTRACT)

The COUNT_EXTRACT constraint controls inference of GP_COUNT8 and GP_COUNT14 cells from behavioral Verilog. It can be used to ensure that a given counter in behavioral logic will, or will not, be mapped to a hard macro.

5.3.1 Applicable Elements

The COUNT_EXTRACT constraint may only be used on vector registers.

5.3.2 Constraint Values

- **Vector register**
  One of the following:
  - **"AUTO":** Same behavior as not specifying any constraint. Counters will be inferred where possible.
  - **"NO":** Do not infer a counter even if the logic matches a supported inference structure.
  - **"FORCE":** Always infer a counter. If a supported inference structure is not found, a synthesis error is produced.
- **Other**
  This constraint will be silently ignored if used on any other entity.

5.3.3 Verilog Usage Example

Figure 5 is an example of a 5-bit down counter with a positive level triggered reset, configured to force counter inference.

```
1  (* COUNT_EXTRACT = "FORCE" *)
2  reg[4:0] count = COUNT_MAX;
3  wire out = (count == 0);
4  always @(posedge clk, posedge count_rst) begin
5
6      //level triggered reset
7      if(count_rst)
8          count       <= 0;
9
10     //counter
11     else begin
12
13         if(count == 0)
14             count   <= COUNT_MAX;
15         else
16             count   <= count - 1'd1;
17
18     end
19
20 end
```

Figure 5: Example for COUNT_EXTRACT constraint

5.4 Drive Strength (DRIVE_STRENGTH)

The DRIVE_STRENGTH constraint configures the output drive strength on the specified I/O pin. Drive strength refers to current sourcing or sinking capacity.

5.4.1 Applicable Elements

The DRIVE_STRENGTH constraint may only be used on top-level module ports.

5.4.2 Constraint Values

- **Top-level module port (IOB)**
  - **"1X":** Default drive strength. This is the default if no constraint is specified.
  - **"2X":** Two times as strong as the default drive strength.
  - **"4X":** Four times as strong as the default drive strength.
- **Other**
  This constraint may not be used on any other entity.

5.4.3 Verilog Usage Example

Figure 6 is an example of a top-level module with two ports a and b. Port a has 1X drive strength (the default) and port b has 2X drive strength.

```
1  module Foo(a, b);
2
3      (* DRIVE_STRENGTH = "1X" *)
4      output wire a;
5
6      (* DRIVE_STRENGTH = "2X" *)
7      output wire b;
8
9  endmodule
```

Figure 6: Example for DRIVE_STRENGTH constraint

5.5 Drive Type (DRIVE_TYPE)

The DRIVE_TYPE constraint configures the output driver on the specified I/O pin.

5.5.1 Applicable Elements

The DRIVE_TYPE constraint may only be used on top-level module ports.

5.5.2 Constraint Values

- **Top-level module port (IOB)**
  - **"NMOS_OD":** Open drain NMOS (pull-down only) driver.
  - **"PMOS_OD":** Open drain PMOS (pull-up only) driver.
  - **"PUSHPULL":** CMOS digital push-pull driver. This is the default if no constraint is specified.
- **Other**
  This constraint may not be used on any other entity.

5.5.3 Verilog Usage Example

Figure 7 is an example of a top-level module with two ports a and b. Port a has a push-pull driver and port b has an open-drain NMOS driver.

```
1  module Foo(a, b);
2
3      (* DRIVE_TYPE = "PUSHPULL" *)
4      output wire a;
5
6      (* DRIVE_TYPE = "NMOS_OD" *)
7      output wire b;
8
9  endmodule
```

Figure 7: Example for DRIVE_TYPE constraint

5.6 Input Buffer Type (IBUF_TYPE)

The IBUF_TYPE constraint configures the input buffer on the specified I/O pin.

5.6.1 Applicable Elements

The IBUF_TYPE constraint may only be used on top-level module ports.

5.6.2 Constraint Values

- **Top-level module port (IOB)**
  - **"ANALOG":** Analog buffer for mixed signal hard IP. Note that analog **outputs** must use this setting to disable the digital input buffer and ensure correct device operation.
  - **"LOW_VOLTAGE":** Low voltage threshold (see datasheet for exact value)
  - **"NORMAL":** Standard threshold (see datasheet for exact value)
- **Other**
  This constraint may not be used on any other entity.

5.6.3 Verilog Usage Example

Figure 8 shows an example of how to configure a top level module input for analog signals.

```
1 (* LOC = "P6" *)
2 (* IBUF_TYPE = "ANALOG" *)
3 input wire vin;
```

Figure 8: Example for IBUF_TYPE constraint

5.7 Physical Location (LOC)

The LOC constraint instructs *gp4par* to place the constrained entity at a specific physical site of the device. This is most commonly used to lock IOBs to specific pins on the package, however advanced users can use it to force arbitrary logic to use specific device resources.

The specified location must be valid for the entity being constrained. Attempting to constrain a cell to an illegal site causes the initial placement to fail with an error message. For example, figure 9 shows the results of attempting to constrain a 3-input combinatorial expression to a 2LUT site.

```
1 ERROR: Cell $abc$104$auto$blifparse.cc:347:parse_blif$106 has invalid LOC
2 constraint LUT2_0 (site is of type GP_2LUT, instance is of type GP_3LUT)
```

Figure 9: Error message produced by invalid LOC constraint

To constrain a multi-bit signal, separate the locations with spaces. For example, “P18 P17" constrains bit 0 to P17 and bit 1 to P18.

5.7.1 Applicable Elements

- IOBs may be constrained by placing a LOC constraint on the top-level module port declaration.
- Inferred LUTs and flipflops may be constrained by placing a LOC constraint on the net driven by the entity to be constrained. This technique cannot constrain inferred multiple-bit registers or inferred combinatorial logic with multiple levels of LUTs; these must be broken up or replaced with primitive instantiation.
- Instantiated primitives may be constrained by placing a LOC constraint on the primitive declaration OR on any net driven by the primitive. Multiple LOC constraints with the same value are ignored; if multiple LOC constraints with different values apply to the same element placement will fail with an error message (figure 10).
- The LOC constraint cannot be applied to inferred counters or shift registers. This will be fixed in a future software release.

```
1 ERROR: Multiple conflicting LOC constraints (COUNT8_5, COUNT8_4) are
2 attached to cell "lfosc_cnt". Please remove one or more of the constraints
3 to allow the design to be placed.
```

Figure 10: Error message produced by invalid LOC constraint

5.7.2 Constraint Values

- `Analog comparator`
  `"ACMP_n"` where `n` is the comparator number. Example: `"ACMP_2"`.

- `Counter`
  `"COUNTm_n"` where `m` is the counter depth and `n` is the counter number. Example: `"COUNT8_5"`.

- `Flipflop` **(DFF cell or inferred 1-bit register)**
  `"DFF_n"` where `n` is the flipflop number. Example: `"DFF_5"`.

- `IOB` **(top-level module port or IOB primitive)**
  `"Pn"` where `n` is the pin number of the device. Example: `"P3"`.

- `LUT`
  `"LUTm_n"` where `m` is the number of inputs for the LUT and `n` is the LUT number. Example: `"LUT3_0"`. Note that it is legal to constrain a smaller LUT to a larger site (but not vice versa); for example a 2-input function may be constrained to a 3LUT or 4LUT site.

- `Inverter`
  `"INV_n"` where `n` is the number of matrix the inverter is located in. Example: `"INV_0"`.

- `Shift register`
  `"SHREG_n"` where `n` is the number of matrix the shift register is located in. Example: `"SHREG_0"`.

- `Voltage reference`
  `"VREF_n"` where `n` is the number of the attached comparator. Example: `"VREF_2"`.

- **Other**
  While this constraint is legal to use on primitives that only have one instance in the device, this is redundant so the names of these sites are not documented here (although they may be obtained from the placement report).

5.7.3 Verilog Usage Example

Figure 11 is an example of a top-level module with four ports a, b, o, and p. These ports are constrained to package pins 20, 19, 18, and 17 respectively.

Signal o is driven by the inferred register o_int, which is constrained to D flipflop number 5. Note that o could not be directly declared as output reg because this infers both a DFF and IOB cell with the same net name. The attached LOC attribute would then apply to the IOB cell and not the DFF.

Signal p is driven by the inferred LUT p_int, which is constrained to 3-input LUT number 0 (rather than one of the 2-input LUTs, as would normally be the case). As with o, a separate named net is required in this case in order to avoid inferring both an IOB and LUT with ambiguous constraints.

```
1  module Foo(a, b, o, p);
2
3      (* LOC = "P20" *)
4      input wire a;
5
6      (* LOC = "P19" *)
7      input wire b;
8
9      (* LOC = "P18" *)
10     output wire o;
11
12     (* LOC = "P17" *)
13     output wire p;
14
15     (* LOC = "DFF_5" *)
16     reg o_int = 0;
17
18     assign o = o_int;
19
20     (* LOC = "LUT3_0" *)
21     wire p_int = (a & b);
22
23     assign p = p_int;
24
25 endmodule
```

Figure 11: Example for LOC constraint

5.8 Pull-Down Resistor (PULLDOWN)

The PULLDOWN constraint instructs *gp4par* to enable the pull-down resistor on the specified input. The exact resistor value ranges may be found in the device datasheet.

5.8.1 Applicable Elements

The PULLDOWN constraint may only be used on top-level module ports.

5.8.2 Constraint Values

- **Top-level module port (IOB)**
  Text string "10k", "100k", or "1M", case sensitive, to specify the nominal value of the pull-down resistor.
- **Other**
  This constraint may not be used on any other entity.

5.8.3 Verilog Usage Example

Figure 12 is an example of a top-level module with three ports a, b, and o. Ports a and b have 10kΩ pull-down resistors; port o is floating.

```
1  module Foo(a, b, o);
2
3      (* PULLDOWN = "10k" *)
4      input wire a;
5
6      (* PULLDOWN = "10k" *)
7      input wire b;
8
9      input wire o;
10
11 endmodule
```

Figure 12: Example for PULLDOWN constraint

5.9 Pull-Up Resistor (PULLUP)

The PULLUP constraint instructs *gp4par* to enable the pull-up resistor on the specified input. The exact resistor value ranges may be found in the device datasheet.

5.9.1 Applicable Elements

The PULLUP constraint may only be used on top-level module ports.

5.9.2 Constraint Values

- **Top-level module port (IOB)**
  Text string "10k", "100k", or "1M", case sensitive, to specify the nominal value of the pull-up resistor.
- **Other**
  This constraint may not be used on any other entity.

5.9.3 Verilog Usage Example

Figure 13 is an example of a top-level module with three ports a, b, and o. Ports a and b have 10kΩ pull-up resistors; port o is floating.

```
1  module Foo(a, b, o);
2
3      (* PULLUP = "10k" *)
4      input wire a;
5
6      (* PULLUP = "10k" *)
7      input wire b;
8
9      input wire o;
10
11 endmodule
```

Figure 13: Example for PULLUP constraint

5.10 Schmitt Trigger (SCHMITT_TRIGGER)

The SCHMITT_TRIGGER constraint instructs *gp4par* to enable the Schmitt trigger on the specified input. The level of hysterisis provided may be found in the device datasheet.

5.10.1 Applicable Elements

The SCHMITT_TRIGGER constraint may only be used on top-level module ports configured in bidirectional or input mode.

5.10.2 Constraint Values

- **Top-level module port (IOB)**
  Specify integer 1 or no value to enable the Schmitt trigger. Specify integer 0, or no constraint, to disable it.
- **Other**
  This constraint may not be used on any other entity.

5.10.3 Verilog Usage Example

Figure 14 is an example of a top-level module with three ports a, b, and o. The Schmitt trigger is enabled for ports a and b, but not o.

```
1  module Foo(a, b, o);
2
3      (* SCHMITT_TRIGGER = 1 *)
4      input wire a;
5
6      (* SCHMITT_TRIGGER *)
7      input wire b;
8
9      (* SCHMITT_TRIGGER = 0 *)
10     input wire o;
11
12 endmodule
```

Figure 14: Example for SCHMITT_TRIGGER constraint

5.11 Shift Register Extraction (SHREG_EXTRACT) (NOT IMPLEMENTED)

The SHREG_EXTRACT constraint controls inference of GP_SHREG cells from behavioral Verilog. It can be used to ensure that a given shift register in behavioral logic will, or will not be, mapped to a hard macro.

5.11.1 Applicable Elements

The SHREG_EXTRACT constraint may only be used on 1-bit registers.

5.11.2 Constraint Values

- **Single-bit register**
  One of the following:
  - **"AUTO":** Same behavior as not specifying any constraint. Shift registers will be inferred where possible.
  - **"NO":** Do not infer a shift register even if the logic matches a supported inference structure.
  - **"FORCE":** Always infer a shift register. If a supported inference structure is not found, a synthesis error is produced.
- **Other**
  This constraint will be silently ignored if used on any other entity.

5.11.3 Verilog Usage Example

Figure 15 is an example of FIXME

```
1  FIXME
```

Figure 15: Example for SHREG_EXTRACT constraint

6 *gp4par* Timing Constraints

Static timing analysis is not yet implemented, thus this section is currently blank.

7 *gp4par* HDL Coding Techniques

When possible, we recommend inferring design elements to maximize design portability. In some cases, such as for hard IP blocks or when exact control over synthesis results is required, it may be necessary to manually instantiate device primitives.

7.1 Counters

Yosys provides limited inference capability for counters which match the capabilities of the hard macro counters (`GP_COUNT8` and `GP_COUNT14`) in the device.

Some hard macro capabilities (most notably input dividers and parallel output to DCMP/DAC blocks) are not yet supported for inference; if these capabilities are required then use explicit primitive instantiation. Future software releases will expand the set of counter features which may be inferred.

7.1.1 Inference Requirements

In order to be inferred, a counter must:

- Be less than 14 bits in width
- Count down only
- Be initialized to the same (maximum) value by both underflow and by power-on reset
- Have either no reset, or a positive level triggered reset to zero
- Not have any logic use the internal counter register. Only the “underflow" signal may be used by surrounding logic.

7.1.2 Counter Related Constraints

By default, the `greenpak4_counters` pass will attempt to infer a counter macro for every counter matching the requirements. If this is not desired, use the `COUNT_EXTRACT` constraint to control inference behavior.

7.1.3 Verilog Usage Example

The example in Figure 16 shows an example of how to infer a resettable down counter.

```
1  localparam COUNT_MAX = 31;
2  reg[4:0] count = COUNT_MAX;
3  wire underflow_out = (count == 0);
4  always @(posedge clk, posedge count_rst) begin
5      if(count_rst)
6          count           <= 0;
7
8      else begin
9          if(count == 0)
10             count       <= COUNT_MAX;
11         else
12             count       <= count - 1'd1;
13     end
14 end
```

Figure 16: Example for counter inference

7.1.4 Reporting

In order to determine whether a given counter was extracted, look at the synthesis report. Figure 17 shows an example synthesis report with a single inferred counter.

```
2.7.
Executing GREENPAK4_COUNTERS pass (mapping counters to hard IP blocks).
  Found 3-bit non-resettable down counter (from 7) for register count
    declared at Blinky.v:93
Extracted 1 counters
```

Figure 17: Sample extraction report

7.2 Shift Registers

Yosys provides inference capability for shift registers matching the capabilities of the “pipe delay" block in the *SLG46620/21*. The shift register is automatically initialized to zero at power-up; there is no support for initializing to any other value.

The internal inverter is not yet supported for inference. If this is required for your design, consider manual instantiation of a GP_SHREG primitive.

7.2.1 Inference Requirements

In order to be inferred as a single GP_SHREG primitive, a shift register must:

- Be at most 16 bits in depth
- Be initialized to zero, or uninitialized
- Have either no reset, or a positive level triggered reset to zero
- Have at most two taps in the shift register connected to external logic. If more taps are required, a second shift register and/or discrete flipflops may be inferred.

7.2.2 Shift Register Related Constraints

As of now there are no constraints to force or disable inference of shift registers. A constraint analogous to COUNT_EXTRACT will likely be added in a future software release.

7.2.3 Verilog Usage Example

The example in Figure 18 shows an example of how to infer a shift register with taps delayed 8 and 16 clocks from the input.

```
1  wire led_in;
2  reg[15:0] led_shreg = 0;
3  assign led1 = led_shreg[7];
4  assign led2 = led_shreg[15];
5  always @(posedge clk) begin
6      led_shreg   <= {led_shreg[14:0], led_in};
7  end
```

Figure 18: Example for shift register inference

7.2.4 Reporting

In order to determine whether a given shift register was extracted, look at the synthesis report. Figure 19 shows an example synthesis report with a single inferred shift register.

```
2.15. Executing SHREGMAP pass (map shift registers).
Converting Blinky.$auto$simplemap.cc:373:simplemap_dff$132 ...
  Blinky.$auto$simplemap.cc:373:simplemap_dff$147 to a shift
  register with depth 16.
Converted 16 dff cells into 1 shift registers.
```

Figure 19: Sample extraction report

8 *gp4par* Verilog Primitives

This section lists all of the device-dependent primitives wrapping hard IP blocks in the device. These may be used for features which are not yet supported by inference, or when exact control over synthesis results is needed.

Pay careful attention to port names and descriptions as these may not exactly match the Silego primitives; some have been changed in order to allow cleaner and more modular HDL. For example, the single “oscillator" block is represented in Verilog by a separate block for each of the three internal oscillators.

8.1 GP_2LUT: 2-Input Lookup Table

8.1.1 Introduction

This primitive corresponds to a single 2-input lookup table. It can implement any combinatorial function of two inputs and one output.

The LUT output is the {IN1, IN0}'th bit of the truth table supplied in the INIT attribute.

This primitive may be manually instantiated if exact control over LUT packing is required, but for most applications we recommend inferring logic.

Note that the placer may re-map GP_2LUT primitives to 3- or 4-input LUT sites to reduce routing congestion. The LOC constraint may be used to force the primitive to a specific 2-input LUT if necessary.

8.1.2 Port Descriptions

Port	Type	Width	Function
IN0	Input	1	Least significant input bit.
IN1	Input	1	Input bit.
OUT	Output	1	Lookup table output.

8.1.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Integer	4	LUT truth table.

8.1.4 Verilog Usage Example

The example shown in figure 20 sets o to the bitwise AND of a and b.

```
wire a;
wire b;
wire o;
GP_2LUT #(
    .INIT(4'h8)
) lut(
    .IN0(a),
    .IN1(b),
    .OUT(o)
);
```

Figure 20: Example usage of GP_2LUT

8.2 GP_3LUT: 3-Input Lookup Table

8.2.1 Introduction

This primitive corresponds to a single 3-input lookup table. It can implement any combinatorial function of three inputs and one output.

The LUT output is the {IN2, IN1, IN0}'th bit of the truth table supplied in the INIT attribute.

This primitive may be manually instantiated if exact control over LUT packing is required, but for most applications we recommend inferring logic.

Note that the placer may re-map GP_3LUT primitives to 4-input LUT sites to reduce routing congestion. The LOC constraint may be used to force the primitive to a specific 3-input LUT if necessary.

8.2.2 Port Descriptions

Port	Type	Width	Function
IN0	Input	1	Least significant input bit.
IN1	Input	1	Input bit.
IN2	Input	1	Most significant input bit.
OUT	Output	1	Lookup table output.

8.2.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Integer	8	LUT truth table.

8.2.4 Verilog Usage Example

The example shown in figure 21 sets o to the bitwise AND of a, b, and c.

```
wire a;
wire b;
wire c;
wire o;
GP_3LUT #(
    .INIT(8'h80)
) lut(
    .IN0(a),
    .IN1(b),
    .IN2(c),
    .OUT(o)
);
```

Figure 21: Example usage of GP_3LUT

8.3 GP_4LUT: 4-Input Lookup Table

8.3.1 Introduction

This primitive corresponds to a single 4-input lookup table. It can implement any combinatorial function of four inputs and one output.

The LUT output is the {IN3, IN2, IN1, IN0}'th bit of the truth table supplied in the INIT attribute.

This primitive may be manually instantiated if exact control over LUT packing is required, but for most applications we recommend inferring logic.

8.3.2 Port Descriptions

Port	Type	Width	Function
IN0	Input	1	Least significant input bit.
IN1	Input	1	Input bit.
IN2	Input	1	Input bit.
IN3	Input	1	Most significant input bit.
OUT	Output	1	Lookup table output.

8.3.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Integer	16	LUT truth table.

8.3.4 Verilog Usage Example

The example shown in figure 22 sets o to the bitwise AND of a, b, c, and d.

```
1  wire a;
2  wire b;
3  wire c;
4  wire d;
5  wire o;
6  GP_4LUT #(
7      .INIT(16'h8000)
8  ) lut(
9      .IN0(a),
10     .IN1(b),
11     .IN2(c),
12     .IN3(d)
13     .OUT(o)
14 );
```

Figure 22: Example usage of GP_4LUT

8.4 GP_ABUF: Analog Buffer

8.4.1 Introduction

This primitive represents an analog buffer which may be used to reduce loading on comparator inputs. Note that not all comparators have buffers on the input; see device datasheet for details.

8.4.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Connect to analog output from IOB.
OUT	Output	1	Connect to VIN of a GP_ACMP block.

8.4.3 Parameter Descriptions

Parameter	Type	Width	Function
BANDWIDTH_KHZ	Integer	8	Buffer bandwidth, in kHz. Must be one of 1, 5, 20, 50. Values greater than 1 require Vdd > 2.7V and use more power.

8.4.4 Verilog Usage Example

The example shown in figure 23 buffers the signal vin to vin_buf with 5 kHz bandwidth.

```
1  wire vin;
2  wire vin_buf;
3  GP_ABUF #(
4      .BANDWIDTH_KHZ(5)
5  ) abuf(
6      .IN(vin),
7      .OUT(vin_buf)
8  );
```

Figure 23: Example usage of GP_ABUF

8.5 GP_ACMP: Analog Comparator

8.5.1 Introduction

This primitive represents an analog comparator.

The comparator may not be powered on until the power-on reset process has completed. The PWREN input must be tied to either the reset-done output of the GP_POR block, or an arbitrary digital expression ANDed with the reset-done output.

8.5.2 Port Descriptions

Port	Type	Width	Function
PWREN	Input	1	**When 1**: normal operation **When 0**: power down mode.
OUT	Output	1	**1**: when VIN > VREF and comparator is running **0**: when VIN < VREF or comparator is powered down.
VIN	Input	1	Input voltage (Vdd or external analog input from IOB).
VREF	Input	1	Input reference voltage.

8.5.3 Parameter Descriptions

Parameter	Type	Width	Function
BANDWIDTH	String	N/A	Comparator bandwidth. Legal values are "LOW" and "HIGH", see device datasheet for actual bandwidth values.
VIN_ATTEN	Integer	3	Attenuation for the input, represented as inverse gain. Legal values are 1, 2, 3, 4. Note that not all comparators support selectable attenuation, check device datasheet for details.
VIN_ISRC_EN	Boolean	1	Set to 1 to enable the 100μA current source on the input. Note that not all comparators support the current source, check device datasheet for details.
HYSTERESIS	Integer	8	Hysteresis, in mV. Legal values are 0, 25, 50, 200. See device datasheet for offset notes.

8.5.4 Verilog Usage Example

The example shown in figure 24 sets cout1 to true if vin is greater than vref_750. The comparator is set to have 25 mV of hysteresis and low bandwidth.

```
1  wire por_done;
2  wire cout1;
3  wire vin;
4  wire vref_750;
5  GP_ACMP #(
6      .BANDWIDTH("LOW"),
7      .VIN_ATTEN(4'd1),
8      .VIN_ISRC_EN(1'b0),
9      .HYSTERESIS(8'd25)
10 ) cmp (
11     .PWREN(por_done),
12     .OUT(cout1),
13     .VIN(vin),
14     .VREF(vref_750)
15 );
```

Figure 24: Example usage of GP_ACMP

8.6 GP_BANDGAP: Bandgap Voltage Reference

8.6.1 Introduction

This primitive allows configuration of the bandgap voltage reference. It produces a 1.0V reference which is used internally by the GP_VREF block. (This connection is implicit and does not need to be provided in your HDL.)

8.6.2 Port Descriptions

Port	Type	Width	Function
OK	Output	1	Goes high when bandgap voltage reference is stable.

8.6.3 Parameter Descriptions

Parameter	Type	Width	Function
AUTO_PWRDN	Boolean	1	**When 1**: Automatically power down bandgap when all loads are powered down. **When 0**: Automatic power-down is disabled. The bandgap is always on.
CHOPPER_EN	Boolean	1	Specify whether to enable the chopper stabilization for the bandgap op-amp. Should always be 1.
OUT_DELAY	Integer	1	Time, in μs, to wait after bandgap startup before asserting OK. Legal values are 100 or 550.

8.6.4 Verilog Usage Example

The example shown in figure 25 sets bg_ok high 550μs after reset.

```
1  wire bg_ok;
2  wire bandgap_vout;
3  GP_BANDGAP #(
4      .AUTO_PWRDN(0),
5      .CHOPPER_EN(1),
6      .OUT_DELAY(550)
7  ) bandgap (
8      .OK(bg_ok)
9  );
```

Figure 25: Example usage of GP_BANDGAP

8.7 GP_CLKBUF: Clock Buffer

8.7.1 Introduction

This primitive is a buffer that drives the a clock signal from fabric to a dedicated clock network used by hard IP cores.

A GP_CLKBUF must be explicitly instantiated to use a general fabric signal as a clock for counters, DCMP/PWM blocks, and the ADC. A future toolchain version may support inferring clock buffers in these cases.

8.7.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Input clock from fabric routing.
OUT	Output	1	Buffered output to hard IP.

8.7.3 Parameter Descriptions

No parameters.

8.7.4 Verilog Usage Example

The example shown in figure 26 buffers the signal clkin to clkbuf.

```
1  wire clkin;
2  wire clkbuf;
3  GP_CLKBUF cbuf(
4      .IN(clkin),
5      .OUT(clkbuf)
6  );
```

Figure 26: Example usage of GP_CLKBUF

8.8 GP_COUNT8: 8-Bit Resettable Down Counter

8.8.1 Introduction

This primitive represents an 8-bit down counter. The count register is initialized to COUNT_TO at power-on reset, and to 0 by the reset pin.

Note that this primitive does **not** always map to a hard IP block with **exactly** 8 bit depth. The technology mapper may map GP_COUNT8 cells to unused 14-bit counters in order to relieve routing pressure. In this case, the high 6 bits of the counter will always be zero.

8.8.2 Port Descriptions

Port	Type	Width	Function
CLK	Input	1	The input clock signal.
RST	Input	1	Reset input (polarity depends on RESET_MODE). When triggered, resets the count register to zero.
OUT	Output	1	Counter underflow output. High whenever the count register equals zero.
POUT	Output	8	Parallel counter output to GP_DCMP or GP_DAC

8.8.3 Parameter Descriptions

Parameter	Type	Width	Function
CLKIN_DIVIDE	Integer	8	Input clock divider. Legal values depend on what source CLK is driven by, see device datasheet.
COUNT_TO	Integer	8	Value to set the counter to on underflow.
RESET_MODE	String	N/A	**"RISING":** Resets the counter on a rising edge of RST. **"FALLING":** Resets the counter on a falling edge of RST. **"BOTH":** Resets the counter on any edge of RST. **"LEVEL":** Resets the counter when RST is high.

8.8.4 Verilog Usage Example

The example shown in figure 27 begins counting from 8'hcc down to zero, on rising edges of clk, as soon as the device exits power-on reset. When the counter reaches zero underflow goes high for one clk cycle and the counter is reset to 8'hcc. The counter also resets immediately to zero, and is held in reset, when count_rst is high.

```
wire underflow;
wire count_rst;
wire clk;
GP_COUNT8 #(
    .RESET_MODE("LEVEL"),
    .COUNT_TO(8'hcc),
    .CLKIN_DIVIDE(1)
) lfosc_cnt (
    .CLK(clk),
    .RST(count_rst),
    .OUT(underflow)
);
```

Figure 27: Example usage of GP_COUNT8

8.9 GP_COUNT8_ADV: 8-Bit Resettable Up/Down Counter With Clock Gating

8.9.1 Introduction

This primitive represents an 8-bit up/down counter. The count register is initialized to COUNT_TO at power-on reset, underflow or overflow, and to a configurable value by the reset pin.

8.9.2 Port Descriptions

Port	Type	Width	Function
CLK	Input	1	The input clock signal.
RST	Input	1	Reset input (polarity depends on RESET_MODE).
UP	Input	1	Direction input.
KEEP	Input	1	Clock gating input.
OUT	Output	1	Counter underflow/overflow output. **When UP is 0**: High whenever the count register equals zero. **When UP is 1**: High whenever the count register equals 8'hff.
POUT	Output	8	Parallel counter output to GP_DCMP or GP_DAC

8.9.3 Parameter Descriptions

Parameter	Type	Width	Function
CLKIN_DIVIDE	Integer	8	Input clock divider. Legal values depend on what source CLK is driven by, see device datasheet.
COUNT_TO	Integer	8	Value to set the counter to on underflow.
RESET_MODE	String	N/A	**"RISING":** Resets the counter on a rising edge of RST. **"FALLING":** Resets the counter on a falling edge of RST. **"BOTH":** Resets the counter on any edge of RST. **"LEVEL":** Resets the counter when RST is high.
RESET_VALUE	String	N/A	**"COUNT_TO":** Resets the counter to COUNT_TO. **"ZERO":** Resets the counter to 0.

8.9.4 Verilog Usage Example

The example shown in figure 28 begins counting from 8'hcc up to 8'hff, on rising edges of clk, as soon as the device exits power-on reset. When the counter reaches 8'hff, overflow goes high for one clk cycle and the counter is reset to 8'hcc. The counter also resets immediately to 8'hcc, and is held in reset, when count_rst is high.

```
wire clk;
wire count_rst;
wire overflow;
GP_COUNT8_ADV #(
    .CLKIN_DIVIDE(1)
    .COUNT_TO(8'hcc),
    .RESET_MODE("LEVEL"),
    .RESET_VALUE("COUNT_TO"),
) up_cnt (
    .CLK(clk),
    .RST(count_rst),
    .OUT(overflow),
    .UP(1'b1),
    .KEEP(1'b0)
);
```

Figure 28: Example usage of GP_COUNT8_ADV

8.10 GP_COUNT14: 14-Bit Resettable Down Counter

8.10.1 Introduction

This primitive represents an 14-bit down counter. The count register is initialized to COUNT_TO at power-on reset, and to 0 by the reset pin.

In the current software, this primitive currently will always map to a 14-bit counter cell in the device. A planned future optimization will allow the technology mapper to map 14-bit counters in the netlist to 8-bit counter cells if the provided count value is less than 256. This should allow better packing density for parameterizable modules containing counters of variable depth.

8.10.2 Port Descriptions

Port	Type	Width	Function
CLK	Input	1	The input clock signal.
RST	Input	1	Reset input (polarity depends on RESET_MODE). When triggered, resets the count register to zero.
OUT	Output	1	Counter underflow output. High whenever the count register equals zero.

8.10.3 Parameter Descriptions

Parameter	Type	Width	Function
CLKIN_DIVIDE	Integer	8	Input clock divider. Legal values depend on what source CLK is driven by, see device datasheet.
COUNT_TO	Integer	14	Value to set the counter to on underflow.
RESET_MODE	String	N/A	**"RISING":** Resets the counter on a rising edge of RST. **"FALLING":** Resets the counter on a falling edge of RST. **"BOTH":** Resets the counter on any edge of RST. **"LEVEL":** Resets the counter when RST is high.

8.10.4 Verilog Usage Example

The example shown in figure 29 begins counting from 14'd3141 down to zero, on rising edges of clk, as soon as the device exits power-on reset. When the counter reaches zero underflow goes high for one clk cycle and the counter is reset to 14'd3141. The counter also resets immediately to zero, and is held in reset, when count_rst is high.

```
wire underflow;
wire count_rst;
wire clk;
GP_COUNT14 #(
    .RESET_MODE("LEVEL"),
    .COUNT_TO(14'd3141),
    .CLKIN_DIVIDE(1)
) lfosc_cnt (
    .CLK(clk),
    .RST(count_rst),
    .OUT(underflow)
);
```

Figure 29: Example usage of GP_COUNT14

8.11 GP_COUNT14_ADV: 14-Bit Resettable Up/Down Counter With Clock Gating

8.11.1 Introduction

This primitive represents a 14-bit up/down counter. The count register is initialized to COUNT_TO at power-on reset, underflow or overflow, and to a configurable value by the reset pin.

8.11.2 Port Descriptions

Port	Type	Width	Function
CLK	Input	1	The input clock signal.
RST	Input	1	Reset input (polarity depends on RESET_MODE).
UP	Input	1	Direction input.
KEEP	Input	1	Clock gating input.
OUT	Output	1	Counter underflow/overflow output. **When UP is 0:** High whenever the count register equals zero. **When UP is 1:** High whenever the count register equals 14'h3fff.
POUT	Output	8	Parallel counter output to GP_DCMP or GP_DAC. Note that only the 8 lowest bits of the count register are routed to POUT.

8.11.3 Parameter Descriptions

Parameter	Type	Width	Function
CLKIN_DIVIDE	Integer	8	Input clock divider. Legal values depend on what source CLK is driven by, see device datasheet.
COUNT_TO	Integer	14	Value to set the counter to on underflow.
RESET_MODE	String	N/A	**"RISING":** Resets the counter on a rising edge of RST. **"FALLING":** Resets the counter on a falling edge of RST. **"BOTH":** Resets the counter on any edge of RST. **"LEVEL":** Resets the counter when RST is high.
RESET_VALUE	String	N/A	**"COUNT_TO":** Resets the counter to COUNT_TO. **"ZERO":** Resets the counter to 0.

8.11.4 Verilog Usage Example

The example shown in figure 30 begins counting from `14'd3141` up to `14'h3fff`, on rising edges of `clk`, as soon as the device exits power-on reset. When the counter reaches `14'h3fff`, `overflow` goes high for one `clk` cycle and the counter is reset to `14'd3141`. The counter also resets immediately to `14'd0`, and is held in reset, when `count_rst` is high.

```
wire clk;
wire count_rst;
wire overflow;
GP_COUNT14_ADV #(
    .CLKIN_DIVIDE(1)
    .COUNT_TO(14'd3141),
    .RESET_MODE("LEVEL"),
    .RESET_VALUE("ZERO")
) up_cnt (
    .CLK(clk),
    .RST(count_rst),
    .OUT(overflow),
    .UP(1'b1),
    .KEEP(1'b0)
);
```

Figure 30: Example usage of `GP_COUNT14_ADV`

8.12 GP_DAC: Digital to Analog Converter

8.12.1 Introduction

This primitive corresponds to an 8-bit digital to analog converter. The DAC operates combinatorially and does not require a clock.

Note that the DAC's input uses dedicated routing and not a general fabric connection; see the device datasheet for information on legal connections.

8.12.2 Port Descriptions

Port	Type	Width	Function
DIN	Input	8	Input data.
VREF	Input	1	Analog reference voltage from GP_VREF.
VOUT	Output	1	Analog output.

8.12.3 Parameter Descriptions

No parameters.

8.12.4 Verilog Usage Example

The example shown in figure 31 FIXME.

```
1  wire[7:0] dcode;         //assigned elsewhere
2  wire vref;               //assigned elsewhere
3  wire vdac;
4  GP_DAC dac(
5      .DIN(dcode),
6      .VOUT(vdac),
7      .VREF(vref)
8  );
```

Figure 31: Example usage of GP_DAC

8.13 GP_DCMP: Digital Comparator

8.13.1 Introduction

This primitive compares two 8-bit unsigned integers. It outputs the the comparison result as a binary "greater than" and "equal" bit.

Note that the inputs use dedicated routing and cannot be driven by general fabric routing. See device datasheet for connectivity options.

8.13.2 Port Descriptions

Port	Type	Width	Function
INP	Input	8	Input to left side of comparison
INN	Input	8	Input to right side of comparison
CLK	Input	1	Clock source
PWRDN	Input	1	Power-down input. Shared by all GP_DCMP blocks.
GREATER	Output	1	True if INP is greater than (or, optionally, equal to) INN.
EQUAL	Output	1	True if INP is equal to INN.

8.13.3 Parameter Descriptions

Parameter	Type	Width	Function
GREATER_OR_EQUAL	Boolean	1	**When 1**: *GREATER* is high when $INP \geq INN$ **When 0**: *GREATER* is high when $INP > INN$
CLK_EDGE	String	N/A	**"RISING":** Outputs are updated on the rising edge of *CLK* **"FALLING":** Outputs are updated on the falling edge of *CLK*
PWRDN_SYNC	Boolean	1	Set true to synchronize *PWRDN* with *CLK*. This adds two cycles of latency when waking the comparator from sleep mode, and holds the output at its current state in sleep. When false, the output is reset to zero in sleep.

8.13.4 Verilog Usage Example

The example shown in figure 32 outputs true on *result* if *foo* is greater than *bar*.

```
wire[7:0] foo;
wire[7:0] bar;
wire result;
GP_DCMP #(
    .GREATER_OR_EQUAL(1'b0),
    .CLK_EDGE("RISING"),
    .PWRDN_SYNC(1'b1)
) comparator(
    .INP(foo),
    .INN(bar),
    .CLK(clk_2mhz),
    .PWRDN(1'b0),
    .GREATER(foo),
    .EQUAL()
    );
```

Figure 32: Example usage of GP_DCMP

8.14 GP_DCMPMUX: Digital Comparator Constant Multiplexer

8.14.1 Introduction

This primitive is a 4:1 mux that can drive the inputs of a GP_DCMP block with any of four 8-bit constant values.

The inputs must be sourced by a GP_DCMPREF block in the current toolchain. A future version of the toolchain may support inferring DCMPREF blocks driving the input of the mux.

8.14.2 Port Descriptions

Port	Type	Width	Function
SEL	Input	2	Mux selector
IN0	Output	8	Input from GP_DCMPREF block
IN1	Output	8	Input from GP_DCMPREF block
IN2	Output	8	Input from GP_DCMPREF block
IN3	Output	8	Input from GP_DCMPREF block
OUTA	Output	8	Multiplexer output to DCMP0 INP input, indexed by SEL
OUTB	Output	8	Multiplexer output to DCMP1 INN input, indexed by 3-SEL.

8.14.3 Parameter Descriptions

None

8.14.4 Verilog Usage Example

The example shown in figure 33 drives *ino* to *refA* and *in3* to *refB*.

```
1  wire[7:0] refA;
2  wire[7:0] refB;
3  wire[1:0] select = 2'b0;
4  wire[7:0] in0;
5  wire[7:0] in1;
6  wire[7:0] in2;
7  wire[7:0] in3;
8  GP_DCMPMUX mux(
9      .SEL(select),
10     .IN0(in0),
11     .IN1(in1),
12     .IN2(in2),
13     .IN3(in3),
14     .OUTA(refA),
15     .OUTB(refB)
16     );
```

Figure 33: Example usage of GP_DCMPMUX

8.15 GP_DCMPREF: Digital Comparator Constant Reference

8.15.1 Introduction

This primitive drives an 8-bit constant value to the input of one or more GP_DCMP or GP_DCMPMUX blocks.

8.15.2 Port Descriptions

Port	Type	Width	Function
OUT	Output	8	Constant output, always equal to *REF_VAL*.

8.15.3 Parameter Descriptions

Parameter	Type	Width	Function
REF_VAL	Integer	8	Constant reference value

8.15.4 Verilog Usage Example

The example shown in figure 34 drives 8'ha3 to the signal *dcref*.

```
1  wire[7:0] dcref;
2  GP_DCMPREF #(.REF_VAL(8'ha3))
3      ref (.OUT(dcref));
```

Figure 34: Example usage of GP_DCMPREF

8.16 GP_DELAY: Programmable Digital Delay Line

8.16.1 Introduction

This primitive corresponds to a programmable digital delay line with four taps. It uniformly delays an incoming digital waveform by a fixed amount. A glitch filter may be enabled to add an additional 200 ns delay (PTV dependent) while rejecting glitches during this period.

For the SLG46620V each tap is nominally 165 ns at 3.3V. The exact range of tap delay values is PTV dependent; see device datasheet for values.

TODO: is jitter characterized anywhere?

8.16.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Input data.
OUT	Output	1	Delayed output.

8.16.3 Parameter Descriptions

Parameter	Type	Width	Function
DELAY_STEPS	Integer	3	Number of delay taps (1-4)
GLITCH_FILTER	Boolean	1	True to enable the glitch filter (approximately 200 ns delay)

8.16.4 Verilog Usage Example

The example shown in figure 35 shows a delay with a nominal value of 110 ns at 3.3V Vdd.

```
1  GP_DELAY #(
2      .DELAY_STEPS(1),
3      .GLITCH_FILTER(0)
4  ) delay(
5      .IN(clk),
6      .OUT(clk_delayed)
7      );
```

Figure 35: Example usage of GP_DELAY

8.17 GP_DFF: Positive Edge Triggered D Flipflop

8.17.1 Introduction

This primitive corresponds to a single D flipflop. It may be mapped to either a GP_DFF or a GP_DFFSR cell by the placer depending on resource utilization and routing congestion.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.17.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
Q	Output	1	Output signal.

8.17.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop.

8.17.4 Verilog Usage Example

The example shown in figure 36 shows a flipflop initialized to zero at powerup.

```
wire clk;
wire din;
wire dout;
GP_DFF #(
    .INIT(1'b0)
) ff (
    .CLK(clk),
    .D(din),
    .Q(dout)
);
```

Figure 36: Example usage of GP_DFF

8.18 GP_DFFI: Positive Edge Triggered D Flipflop with Inverted Output

8.18.1 Introduction

This primitive corresponds to a single D flipflop with an inverted output. It may be mapped to either a GP_DFF or a GP_DFFSR cell by the placer depending on resource utilization and routing congestion.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.18.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
nQ	Output	1	Output signal.

8.18.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop. The power-up value of Q is inverted with respect to INIT.

8.18.4 Verilog Usage Example

The example shown in figure 37 shows a flipflop initialized to zero at powerup.

```
wire clk;
wire din;
wire dout;
GP_DFFI #(
    .INIT(1'b0)
) ff (
    .CLK(clk),
    .D(din),
    .nQ(dout)
);
```

Figure 37: Example usage of GP_DFFI

8.19 GP_DFFR: Positive Edge Triggered D Flipflop with Reset

8.19.1 Introduction

This primitive corresponds to a single D flipflop with active-low reset. It is internally remapped to a GP_DFFSR cell by the placer.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.19.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
nRST	Input	1	Active-low reset.
Q	Output	1	Output signal.

8.19.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop.

8.19.4 Verilog Usage Example

The example shown in figure 38 shows a flipflop initialized to zero at power-up, and cleared to zero when nrst goes low.

```
wire clk;
wire din;
wire dout;
wire nrst;
GP_DFFR #(
    .INIT(1'b0)
) ff (
    .CLK(clk),
    .D(din),
    .Q(dout),
    .nRST(nrst)
);
```

Figure 38: Example usage of GP_DFFR

8.20 GP_DFFRI: Positive Edge Triggered D Flipflop with Reset and Inverted Output

8.20.1 Introduction

This primitive corresponds to a single D flipflop with active-low reset and an inverted output. It is internally remapped to a GP_DFFSR cell by the placer.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.20.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
nRST	Input	1	Active-low reset.
nQ	Output	1	Output signal.

8.20.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop. The power-up value of Q is inverted with respect to INIT.

8.20.4 Verilog Usage Example

The example shown in figure 39 shows a flipflop initialized to zero at power-up, and cleared to zero when nrst goes low.

```
1  wire clk;
2  wire din;
3  wire dout;
4  wire nrst;
5  GP_DFFRI #(
6      .INIT(1'b0)
7  ) ff (
8      .CLK(clk),
9      .D(din),
10     .nQ(dout),
11     .nRST(nrst)
12 );
```

Figure 39: Example usage of GP_DFFRI

8.21 GP_DFFS: Positive Edge Triggered D Flipflop with Set

8.21.1 Introduction

This primitive corresponds to a single D flipflop with active-low set. It is internally remapped to a GP_DFFSR cell by the placer.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.21.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
nSET	Input	1	Active-low set.
Q	Output	1	Output signal.

8.21.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop.

8.21.4 Verilog Usage Example

The example shown in figure 40 shows a flipflop initialized to zero at power-up, and set to one when nset goes low.

```
wire clk;
wire din;
wire dout;
wire nset;
GP_DFFS #(
    .INIT(1'b0)
) ff (
    .CLK(clk),
    .D(din),
    .Q(dout),
    .nSET(nset)
);
```

Figure 40: Example usage of GP_DFFS

8.22 GP_DFFSI: Positive Edge Triggered D Flipflop with Set and Inverted Output

8.22.1 Introduction

This primitive corresponds to a single D flipflop with active-low set and an inverted output. It is internally remapped to a GP_DFFSR cell by the placer.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.22.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
nSET	Input	1	Active-low set.
nQ	Output	1	Output signal.

8.22.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop. The power-up value of Q is inverted with respect to INIT.

8.22.4 Verilog Usage Example

The example shown in figure 41 shows a flipflop initialized to zero at power-up, and set to one when nset goes low.

```
1  wire clk;
2  wire din;
3  wire dout;
4  wire nset;
5  GP_DFFSI #(
6      .INIT(1'b0)
7  ) ff (
8      .CLK(clk),
9      .D(din),
10     .nQ(dout),
11     .nSET(nset)
12 );
```

Figure 41: Example usage of GP_DFFSI

8.23 GP_DFFSR: Positive Edge Triggered D Flipflop with Set or Reset

8.23.1 Introduction

This primitive corresponds to a single D flipflop with active-low set or reset (but not both), selectable at synthesis time by a parameter.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.23.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
nSR	Input	1	Active-low set/reset.
Q	Output	1	Output signal.

8.23.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop.
SRMODE	Boolean	1	Set to 1 for nSR to act as a set, or 0 for reset.

8.23.4 Verilog Usage Example

The example shown in figure 42 shows a flipflop initialized to zero at power-up, and set to one when `nset` goes low.

```
wire clk;
wire din;
wire dout;
wire nset;
GP_DFFSR #(
    .INIT(1'b0),
    .SRMODE(1'b1)
) ff (
    .CLK(clk),
    .D(din),
    .Q(dout),
    .nSR(nset)
);
```

Figure 42: Example usage of GP_DFFSR

8.24 GP_DFFSRI: Positive Edge Triggered D Flipflop with Set or Reset and Inverted Output

8.24.1 Introduction

This primitive corresponds to a single D flipflop with active-low set or reset (but not both), selectable at synthesis time by a parameter. The flipflop output is inverted.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.24.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
CLK	Input	1	Input clock.
nSR	Input	1	Active-low set/reset.
nQ	Output	1	Output signal.

8.24.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the flipflop. The power-up value of Q is inverted with respect to INIT.
SRMODE	Boolean	1	Set to 1 for nSR to act as a set, or 0 for reset.

8.24.4 Verilog Usage Example

The example shown in figure 43 shows a flipflop initialized to zero at power-up, and set to one when nset goes low.

```
wire clk;
wire din;
wire dout;
wire nset;
GP_DFFSRI #(
    .INIT(1'b0),
    .SRMODE(1'b1)
) ff (
    .CLK(clk),
    .D(din),
    .nQ(dout),
    .nSR(nset)
);
```

Figure 43: Example usage of GP_DFFSRI

8.25 GP_DLATCH: Negative Level Triggered D Latch

8.25.1 Introduction

This primitive corresponds to a single D latch. The latch updates when the clock is negative and holds when the clock is positive.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.25.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
Q	Output	1	Output signal.

8.25.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch.

8.25.4 Verilog Usage Example

The example shown in figure 44 shows a latch initialized to zero at power-up

```
1  wire clk;
2  wire din;
3  wire dout;
4  GP_DLATCH #(
5      .INIT(1'b0)
6  ) lat (
7      .CLK(clk),
8      .D(din),
9      .Q(dout)
10 );
```

Figure 44: Example usage of GP_DLATCH

8.26 GP_DLATCHI: Negative Level Triggered D Latch with Inverted Output

8.26.1 Introduction

This primitive corresponds to a single D latch. The latch updates when the clock is negative and holds when the clock is positive. The flipflop output is inverted.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.26.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
nQ	Output	1	Output signal.

8.26.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch. The power-up value of nQ is inverted with respect to INIT.

8.26.4 Verilog Usage Example

The example shown in figure 47 shows a latch initialized to zero at power-up.

```
wire clk;
wire din;
wire dout;
GP_DLATCHI #(
    .INIT(1'b0)
) lat (
    .CLK(clk),
    .D(din),
    .nQ(dout)
);
```

Figure 45: Example usage of GP_DLATCHI

8.27 GP_DLATCHR: Negative Level Triggered D Latch with Reset

8.27.1 Introduction

This primitive corresponds to a single D latch with active-low reset. The latch updates when the clock is negative and holds when the clock is positive.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.27.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
nRST	Input	1	Active-low reset.
Q	Output	1	Output signal.

8.27.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch.

8.27.4 Verilog Usage Example

The example shown in figure 46 shows a latch initialized to zero at power-up, and cleared to zero when nrst goes low.

```
wire clk;
wire din;
wire dout;
wire nrst;
GP_DLATCHR #(
    .INIT(1'b0)
) lat (
    .CLK(clk),
    .D(din),
    .Q(dout),
    .nRST(nrst)
);
```

Figure 46: Example usage of GP_DLATCHR

8.28 GP_DLATCHRI: Negative Level Triggered D Latch with Reset and Inverted Output

8.28.1 Introduction

This primitive corresponds to a single D latch with active-low reset. The latch updates when the clock is negative and holds when the clock is positive. The flipflop output is inverted.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.28.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
nRST	Input	1	Active-low reset.
nQ	Output	1	Output signal.

8.28.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch. The power-up value of nQ is inverted with respect to INIT.

8.28.4 Verilog Usage Example

The example shown in figure 47 shows a latch initialized to zero at power-up, and cleared to zero when nrst goes low.

```
1  wire clk;
2  wire din;
3  wire dout;
4  wire nrst;
5  GP_DLATCHRI #(
6      .INIT(1'b0)
7  ) lat (
8      .CLK(clk),
9      .D(din),
10     .nQ(dout),
11     .nRST(nrst)
12 );
```

Figure 47: Example usage of GP_DLATCHRI

8.29 GP_DLATCHS: Negative Level Triggered D Latch with Set

8.29.1 Introduction

This primitive corresponds to a single D latch with active-low set. The latch updates when the clock is negative and holds when the clock is positive.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.29.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
nSET	Input	1	Active-low set.
Q	Output	1	Output signal.

8.29.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch.

8.29.4 Verilog Usage Example

The example shown in figure 48 shows a latch initialized to zero at power-up, and set to one when nset goes low.

```
wire clk;
wire din;
wire dout;
wire nset;
GP_DLATCHS #(
    .INIT(1'b0),
    .SRMODE(1'b1)
) lat (
    .CLK(clk),
    .D(din),
    .Q(dout),
    .nSET(nset)
);
```

Figure 48: Example usage of GP_DLATCHS

8.30 GP_DLATCHSI: Negative Level Triggered D Latch with Set and Inverted Output

8.30.1 Introduction

This primitive corresponds to a single D latch with active-low set. The latch updates when the clock is negative and holds when the clock is positive. The flipflop output is inverted.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.30.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
nSET	Input	1	Active-low set.
nQ	Output	1	Output signal.

8.30.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch. The power-up value of nQ is inverted with respect to INIT.

8.30.4 Verilog Usage Example

The example shown in figure 49 shows a latch initialized to zero at power-up, and set to one when nset goes low.

```
wire clk;
wire din;
wire dout;
wire nset;
GP_DLATCHSI #(
    .INIT(1'b0)
) lat (
    .CLK(clk),
    .D(din),
    .nQ(dout),
    .nSET(nset)
);
```

Figure 49: Example usage of GP_DLATCHSI

8.31 GP_DLATCHSR: Negative Level Triggered D Latch with Set or Reset

8.31.1 Introduction

This primitive corresponds to a single D latch with active-low set or reset (but not both), selectable at synthesis time by a parameter. The latch updates when the clock is negative and holds when the clock is positive.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.31.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
nSR	Input	1	Active-low set/reset.
Q	Output	1	Output signal.

8.31.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch.
SRMODE	Boolean	1	Set to 1 for nSR to act as a set, or 0 for reset.

8.31.4 Verilog Usage Example

The example shown in figure 50 shows a latch initialized to zero at power-up, and set to one when nset goes low.

```
wire clk;
wire din;
wire dout;
wire nset;
GP_DLATCHSR #(
    .INIT(1'b0),
    .SRMODE(1'b1)
) lat (
    .CLK(clk),
    .D(din),
    .Q(dout),
    .nSR(nset)
);
```

Figure 50: Example usage of GP_DLATCHSR

8.32 GP_DLATCHSRI: Negative Level Triggered D Latch with Set or Reset and Inverted Output

8.32.1 Introduction

This primitive corresponds to a single D latch with active-low set or reset (but not both), selectable at synthesis time by a parameter. The latch updates when the clock is negative and holds when the clock is positive. The flipflop output is inverted.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.32.2 Port Descriptions

Port	Type	Width	Function
D	Input	1	Input data.
nCLK	Input	1	Input clock.
nSR	Input	1	Active-low set/reset.
nQ	Output	1	Output signal.

8.32.3 Parameter Descriptions

Parameter	Type	Width	Function
INIT	Boolean	1	Power-on initialization value of the latch. The power-up value of nQ is inverted with respect to INIT.
SRMODE	Boolean	1	Set to 1 for nSR to act as a set, or 0 for reset.

8.32.4 Verilog Usage Example

The example shown in figure 51 shows a latch initialized to zero at power-up, and set to one when nset goes low.

```
wire clk;
wire din;
wire dout;
wire nset;
GP_DLATCHSRI #(
    .INIT(1'b0),
    .SRMODE(1'b1)
) lat (
    .CLK(clk),
    .D(din),
    .nQ(dout),
    .nSR(nset)
);
```

Figure 51: Example usage of GP_DLATCHSRI

8.33 GP_EDGEDET: Edge detector

8.33.1 Introduction

This primitive is a monostable delay line with four taps. It can be configured to detect rising edges, falling, or both. An additional delay can be added for glitch filtering.

For the SLG46620V each tap is nominally 150 ns at 3.3V. The exact range of tap delay values is PTV dependent; see device datasheet for values.

8.33.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Input data.
OUT	Output	1	Edge detector output.

8.33.3 Parameter Descriptions

Parameter	Type	Width	Function
DELAY_STEPS	Integer	3	Number of delay taps (1-4)
EDGE_DIRECTION	String		One of “RISING", “FALLING", “BOTH"
GLITCH_FILTER	Boolean	1	True to enable the glitch filter (approximately 200 ns delay)

8.33.4 Verilog Usage Example

The example shown in figure 52 drives clk_edge high for approximately 150 ns each time clk goes high.

```
1  GP_EDGEDET #(
2      .DELAY_STEPS(1),
3      .EDGE_DIRECTION("RISING"),
4      .GLITCH_FILTER(0)
5  ) edgedet(
6      .IN(clk),
7      .OUT(clk_edge)
8      );
```

Figure 52: Example usage of GP_EDGEDET

8.34 GP_IBUF: Input Buffer

8.34.1 Introduction

This primitive corresponds to a top-level input buffer (IOB).

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

Note that in the current *gp4par* version, constraints must be attached to the pad wire (signal attached to IN), not to the IOB itself. Constraints on the IOB (if manually instantiated) are ignored.

8.34.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Input signal from top-level module port.
OUT	Output	1	Output signal to fabric routing.

8.34.3 Parameter Descriptions

No parameters.

8.34.4 Verilog Usage Example

The example shown in figure 53 shows explicit instantiation of a GP_IBUF on a top-level port.

```
1  module foo(a, b);
2      input wire a;
3      output wire b;
4
5      wire a_ibuf;
6      GP_IBUF ibuf(.IN(a), .OUT(a_ibuf));
7
8      //logic using a_ibuf
9
10     ...
11 endmodule
```

Figure 53: Example usage of GP_IBUF

8.35 GP_INV: Inverter

8.35.1 Introduction

This primitive corresponds to a single NOT gate.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

8.35.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Input bit.
OUT	Output	1	Inverted signal.

8.35.3 Parameter Descriptions

No parameters.

8.35.4 Verilog Usage Example

The example shown in figure 54 sets o to the complement of a.

```
1  wire a;
2  wire o;
3  GP_INV inv(
4      .IN(a),
5      .OUT(o)
6  );
```

Figure 54: Example usage of GP_INV

8.36 GP_IOBUF: Input/Output Buffer

8.36.1 Introduction

This primitive corresponds to a top-level input/output buffer (IOB).

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

Note that in the current *gp4par* version, constraints must be attached to the pad wire (signal attached to IO), not to the IOB itself. Constraints on the IOB (if manually instantiated) are ignored.

8.36.2 Port Descriptions

Port	Type	Width	Function
IO	Bidirectional	1	I/O signal to top-level module port.
IN	Input	1	Input signal from fabric routing.
OE	Input	1	Tri-state output enable. When low, the output driver is disabled.
OUT	Output	1	Output signal to fabric routing.

8.36.3 Parameter Descriptions

No parameters.

8.36.4 Verilog Usage Example

The example shown in figure 55 shows explicit instantiation of a GP_IOBUF on a top-level port.

```
1  module foo(a, b,c);
2      input wire a;
3      input wire b;
4      inout wire c;
5
6      wire c_ibuf;
7      wire c_obuf;
8      wire oe;
9      GP_IOBUF iobuf(
10         .IN(c_obuf),
11         .OUT(c_ibuf),
12         .OE(oe),
13         .IO(c)
14     );
15
16     //logic driving oe/c_obuf and using c_ibuf
17
18     ...
19
20 endmodule
```

Figure 55: Example usage of GP_IOBUF

8.37 GP_LFOSC: Low Frequency Oscillator

8.37.1 Introduction

This primitive represents the low-frequency oscillator block. It has a nominal frequency of 1730 Hz on the *SLG46620V* and *SLG46621V*, and 1900 Hz on the *SLG46140V*.

Note that all of the oscillator blocks in *GreenPAK4* internally share the PWRDN input. As a result, all oscillator blocks in the design that have PWRDN_EN set to 1 must have PWRDN connected to the same net. Failure to follow this rule will result in a physical DRC failure.

8.37.2 Port Descriptions

Port	Type	Width	Function
PWRDN	Input	1	**When** PWRDN_EN **is 1**: Set high to power down the oscillator. **When** PWRDN_EN **is 0**: Ignored, tie to 1'b0.
CLKOUT	Output	1	Clock signal.

8.37.3 Parameter Descriptions

Parameter	Type	Width	Function
AUTO_PWRDN	Boolean	1	**When 1**: Automatically power down the oscillator when all loads are powered down. **When 0**: Automatic power-down is disabled. The PWRDN_EN and PWRDN inputs are unaffected.
OUT_DIV	Integer	8	Output divider. Legal values: 1, 2, 4, 16
PWRDN_EN	Boolean	1	Set to 1 to enable the PWRDN pin, or 0 to disable.

8.37.4 Verilog Usage Example

The example shown in figure 56 drives a 108 Hz clock onto clk_108hz. All power management features are disabled.

```
wire clk_108hz;
GP_LFOSC #(
    .PWRDN_EN(0),
    .AUTO_PWRDN(0),
    .OUT_DIV(16)
) lfosc (
    .PWRDN(1'b0),
    .CLKOUT(clk_108hz)
);
```

Figure 56: Example usage of GP_LFOSC

8.38 GP_OBUF: Output Buffer

8.38.1 Introduction

This primitive corresponds to a top-level output buffer (IOB).

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

Note that in the current *gp4par* version, constraints must be attached to the pad wire (signal attached to OUT), not to the IOB itself. Constraints on the IOB (if manually instantiated) are ignored.

8.38.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Input signal from fabric routing.
OUT	Output	1	Output signal to top-level module port.

8.38.3 Parameter Descriptions

No parameters.

8.38.4 Verilog Usage Example

The example shown in figure 57 shows explicit instantiation of a GP_OBUF on a top-level port.

```
1  module foo(a, b);
2      output wire a;
3      input wire b;
4
5      wire a_obuf;
6      GP_OBUF obuf(.IN(a_obuf), .OUT(a));
7
8      //logic driving a_obuf
9
10     ...
11 endmodule
```

Figure 57: Example usage of GP_OBUF

8.39 GP_OBUFT: Output Buffer with Tri-State

8.39.1 Introduction

This primitive corresponds to a top-level output buffer (IOB) with tri-state control. It is logically equivalent to a GP_IOBUF with the OUT port left unconnected.

This primitive may be manually instantiated if exact control over packing is required, but for most applications we recommend inferring logic.

Note that in the current *gp4par* version, constraints must be attached to the pad wire (signal attached to OUT), not to the IOB itself. Constraints on the IOB (if manually instantiated) are ignored.

8.39.2 Port Descriptions

Port	Type	Width	Function
IN	Input	1	Input signal from fabric routing.
OE	Input	1	Output enable. When low, the output floats.
OUT	Output	1	Output signal to top-level module port.

8.39.3 Parameter Descriptions

No parameters.

8.39.4 Verilog Usage Example

The example shown in figure 58 shows explicit instantiation of a GP_OBUFT on a top-level port.

```
1  module foo(a, b);
2      output wire a;
3      input wire b;
4
5      wire a_obuf;
6      wire oe;
7      GP_OBUFT obuf(.IN(a_obuf), .OUT(a), .OE(oe));
8
9      //logic driving a_obuf and oe
10
11     ...
12 endmodule
```

Figure 58: Example usage of GP_OBUFT

8.40 GP_PGA: Programmable-Gain Amplifier

8.40.1 Introduction

This primitive represents a programmable-gain analog amplifier.

If the PGA is only used to drive the ADC, it will automatically power on when the ADC is running and otherwise sleep. If the PGA's output is used by other logic, it will remain on regardless of ADC power state. There is no way to control the PGA's power domain from fabric logic.

Note that VIN_SEL uses dedicated routing to an IOB (see device datasheet for pin numbering) and cannot be driven by fabric logic or a constant zero.

8.40.2 Port Descriptions

Port	Type	Width	Function
VIN_P	Input	1	Positive differential input, or single-ended input A. Tie to 1'b0 if unused.
VIN_N	Input	1	Negative differential input, or single ended input B. Tie to 1'b0 if unused.
VIN_SEL	Input	1	Input mux selector. 1 = VIN_P, 0 = VIN_N. • Differential or pseudo-differential modes: Ignored, tie to 1'b1 • Single-ended mode: Tie to 1'b1 or input signal from IOB.
VOUT	Output	1	Analog output.

8.40.3 Parameter Descriptions

Parameter	Type	Width	Function
GAIN	Real	N/A	Gain of the PGA. Legal values: • Differential or pseudo-differential modes: 1, 2, 4, 8, or 16 • Single-ended mode: 0.25, 0.5, 1, 2, 4, or 8
INPUT_MODE	String	N/A	Input buffer configuration. **"SINGLE":** single-ended input **"DIFF":** true differential input **"PDIFF":** pseudo-differential input

8.40.4 Verilog Usage Example

The example shown in figure 59 shows a single analog input being amplified 2x and fed to the signal pgaout.

```
wire ain1;
wire pgaout;
GP_PGA #(
    .GAIN(2),
    .INPUT_MODE("SINGLE")
) pga (
    .VIN_P(ain1),
    .VIN_N(),
    .VIN_SEL(1'b1),
    .VOUT(pgaout)
);
```

Figure 59: Example usage of GP_PGA

8.41 GP_PGEN: Pattern Generator

8.41.1 Introduction

This primitive represents a digital sequence generator. It outputs a repeating sequence of 2...16 user-selected bits on successive clock edges.

8.41.2 Port Descriptions

Port	Type	Width	Function
nRST	Input	1	Active-low reset. OUT is set to PATTERN_DATA[0] during reset.
CLK	Input	1	Pattern clock
OUT	Output	1	Generated pattern data.

8.41.3 Parameter Descriptions

Parameter	Type	Width	Function
PATTERN_DATA	Integer	16	The pattern to be generated (LSB sent first).
PATTERN_LEN	Integer	5	Length of the pattern to send.

8.41.4 Verilog Usage Example

The example shown in figure 60 outputs an alternating sequence of 1s and 0s on the signal *data*, shifting phase after 8 bits have been sent.

```
1  wire por_done;
2  GP_PGEN #(
3      .PATTERN_DATA(16'h55AA),
4      .PATTERN_LEN(5'd16)
5  ) pgen (
6      .nRST(nrst),
7      .CLK(clk),
8      .OUT(data)
9  );
```

Figure 60: Example usage of GP_PGEN

8.42 GP_POR: Power-On Reset

8.42.1 Introduction

This primitive represents the power-on reset. It may be used in user logic to gate signals that can toggle early in the startup process, in order to avoid glitches.

8.42.2 Port Descriptions

Port	Type	Width	Function
RST_DONE	Output	1	Reset-done output. Goes high when the device has finished initializing.

8.42.3 Parameter Descriptions

Parameter	Type	Width	Function
POR_TIME	Integer	16	Delay, in μs, from power-good to release of RST_DONE. Legal values are 4, 500.

8.42.4 Verilog Usage Example

The example shown in figure 61 sets por_done high $500\mu s$ after initialization is complete.

```
wire por_done;
GP_POR #(
    .POR_TIME(500)
) por (
    .RST_DONE(por_done)
);
```

Figure 61: Example usage of GP_POR

8.43 GP_PWRDET: Power Detector

8.43.1 Introduction

This primitive represents the power detector for the charge pump driving the analog circuitry.

(FIXME: check this) This block is only active when –disable-charge-pump is not specified.

8.43.2 Port Descriptions

Port	Type	Width	Function
VDD_LOW	Output	1	Power-detect output. High when Vdd is less than 2.7V.

8.43.3 Parameter Descriptions

No parameters.

8.43.4 Verilog Usage Example

The example shown in figure 62 drives analog_brownout high when VDD drops below 2.7V.

```
1  wire analog_brownout;
2  GP_PWRDET det(
3      .VDD_LOW(analog_brownout)
4      );
```

Figure 62: Example usage of GP_PWRDET

8.44 GP_RCOSC: RC Oscillator

8.44.1 Introduction

This primitive represents the RC oscillator block. This block can be configured to oscillate at either 2 MHz or 25 kHz by setting the OSC_FREQ parameter.

The oscillator has two cascaded dividers on its output: a pre-divider and post-divider. The pre-divider output drives some hard IP blocks (counters and PWM) via dedicated routing, as well as the input of the post-divider. The post-divider output drives general fabric routing.

Note that all of the oscillator blocks in *GreenPak4* internally share the PWRDN input. As a result, all oscillator blocks in the design that have PWRDN_EN set to 1 must have PWRDN connected to the same net. Failure to follow this rule will result in a physical DRC failure.

8.44.2 Port Descriptions

Port	Type	Width	Function
PWRDN	Input	1	**When** PWRDN_EN **is 1**: Set high to power down the oscillator. **When** PWRDN_EN **is 0**: Ignored, tie to 1'b0.
CLKOUT_HARDIP	Output	1	Pre-divided clock output to hard IP.
CLKOUT_FABRIC	Output	1	Final clock signal to general fabric routing.

8.44.3 Parameter Descriptions

Parameter	Type	Width	Function
AUTO_PWRDN	Boolean	1	**When 1**: Automatically power down the oscillator when all loads are powered down. **When 0**: Automatic power-down is disabled. The PWRDN_EN and PWRDN inputs are unaffected.
OSC_FREQ	String	N/A	Oscillator frequency select. Must be one of "2M" or "25k".
HARDIP_DIV	Integer	8	Output pre-divider for CLKOUT_HARDIP. Legal values are 1, 2, 4, 8.
FABRIC_DIV	Integer	8	Output post-divider for CLKOUT_FABRIC. Legal values are 1, 2, 3, 4, 8, 12, 24, 64.
PWRDN_EN	Boolean	1	Set to 1 to enable the PWRDN pin, or 0 to disable.

8.44.4 Verilog Usage Example

The example shown in figure 63 drives a 6.25 kHz clock to `clk_fast` and a 521 Hz clock to `clk_slow`. All power management features are disabled.

```
wire clk_fast;
wire clk_slow;
GP_RCOSC #(
    .PWRDN_EN(0),
    .AUTO_PWRDN(0),
    .OSC_FREQ("25k"),
    .HARDIP_DIV(4),
    .FABRIC_DIV(12)
) rcosc (
    .PWRDN(1'b0),
    .CLKOUT_HARDIP(clk_fast),
    .CLKOUT_FABRIC(clk_slow)
);
```

Figure 63: Example usage of `GP_RCOSC`

8.45 GP_RINGOSC: Ring Oscillator

8.45.1 Introduction

This primitive represents the ring oscillator block. The output frequency is highly PTV dependent but is nominally 25 MHz for the SLG46140V and 27 MHz for the SLG46620V.

The oscillator has two cascaded dividers on its output: a pre-divider and post-divider. The pre-divider output drives some hard IP blocks (counters and PWM) via dedicated routing, as well as the input of the post-divider. The post-divider output drives general fabric routing.

Note that all of the oscillator blocks in *GreenPak4* internally share the PWRDN input. As a result, all oscillator blocks in the design that have PWRDN_EN set to 1 must have PWRDN connected to the same net. Failure to follow this rule will result in a physical DRC failure.

8.45.2 Port Descriptions

Port	Type	Width	Function
PWRDN	Input	1	**When** PWRDN_EN **is 1**: Set high to power down the oscillator. **When** PWRDN_EN **is 0**: Ignored, tie to 1'b0.
CLKOUT_HARDIP	Output	1	Pre-divided clock output to hard IP.
CLKOUT_FABRIC	Output	1	Final clock signal to general fabric routing.

8.45.3 Parameter Descriptions

Parameter	Type	Width	Function
AUTO_PWRDN	Boolean	1	**When 1**: Automatically power down the oscillator when all loads are powered down. **When 0**: Automatic power-down is disabled. The PWRDN_EN and PWRDN inputs are unaffected.
HARDIP_DIV	Integer	8	Output pre-divider for CLKOUT_HARDIP. Legal values are 1, 4, 8, 16.
FABRIC_DIV	Integer	8	Output post-divider for CLKOUT_FABRIC. Legal values are 1, 2, 3, 4, 8, 12, 24, 64.
PWRDN_EN	Boolean	1	Set to 1 to enable the PWRDN pin, or 0 to disable.

8.45.4 Verilog Usage Example

The example shown in figure 64 drives a 6.75 MHz clock onto `clk_fast` and a 562.5 kHz clock onto `clk_slow`. All power management features are disabled.

```
wire clk_fast;
wire clk_slow;
GP_RINGOSC #(
    .PWRDN_EN(0),
    .AUTO_PWRDN(0),
    .HARDIP_DIV(4),
    .FABRIC_DIV(12)
) ringosc (
    .PWRDN(1'b0),
    .CLKOUT_HARDIP(clk_fast),
    .CLKOUT_FABRIC(clk_slow)
);
```

Figure 64: Example usage of `GP_RINGOSC`

8.46 GP_SHREG: Shift Register

8.46.1 Introduction

This primitive represents the “pipe delay" block, a 16-bit shift register with programmable tap position. It has two major differences from most FPGA shift register blocks: there are two independent taps instead of one, and the tap positions are fixed at synthesis time rather than being runtime programmable.

8.46.2 Port Descriptions

Port	Type	Width	Function
nRST	Input	1	Active-low reset input. Clears the shift register to zero when asserted. Tie high if not used.
CLK	Input	1	Clock for the shift register.
IN	Input	1	Input data for the shift register.
OUTA	Output	1	Tap A for the shift register.
OUTB	Output	1	Tap B for the shift register.

8.46.3 Parameter Descriptions

Parameter	Type	Width	Function
OUTA_TAP	Integer	5	Number of register stages for output A. Legal range is 1 to 16 inclusive.
OUTA_INVERT	Boolean	1	Set to 1 to invert tap A’s output.
OUTB_TAP	Integer	5	Number of register stages for output B. Legal range is 1 to 16 inclusive.

8.46.4 Verilog Usage Example

The example shown in figure 65 uses both outputs of a GP_SHREG in non-inverting mode. bar_delay1 is delayed by 8 cycles and bar_delay2 is delayed by 16 cycles. The output inverter is off.

```
wire bar_delay1;
wire bar_delay2;
GP_SHREG #(
    .OUTA_TAP(8),
    .OUTA_INVERT(0),
    .OUTB_TAP(16)
) shreg (
    .nRST(1'b1),
    .CLK(clk_108hz),
    .IN(foo),
    .OUTA(bar_delay1),
    .OUTB(bar_delay2)
);
```

Figure 65: Example usage of GP_SHREG

8.47 GP_SPI: SPI Slave

8.47.1 Introduction

This primitive is a unidirectional SPI slave.

It can receive or transmit up to two bytes of data at a time to the bus master. The direction of data transfer is set statically at compile time and cannot be changed, since the input or output serial data both use the same pin on the device. Chip select and clock inputs are routed through the switch matrix and can connect to arbitrary external pins or even fabric logic.

8.47.2 Port Descriptions

Port	Type	Width	Function
SCK	Input	1	SPI clock signal (must be buffered by GP_CLKBUF).
SDAT	Inout	1	SPI input or output. Must be connected to pin 10 (*SLG4662x*) or 12 (*SLG46140*)
CSN	Input	1	Chip select input from IOB
INT	Output	1	Transaction-complete status flag. Goes high for one *SCK* cycle when transaction completes.
TXD_HIGH	Input	8	High word of data to send to master. Must connect to parallel output of a *GP_COUNTx* block. Only valid if *DATA_WIDTH* is 16 and *DIRECTION* is "OUTPUT", connect to 8'h00 otherwise.
TXD_LOW	Input	8	Low word of data to send. Must connect to parallel output of a *GP_COUNTx* or GP_ADC block. Only valid if *DIRECTION* is "OUTPUT", connect to 8'h00 otherwise.
RXD_HIGH	Output	8	8-bit high word of data from master. Must connect to parallel input of a *GP_COUNTx* or GP_DCMP block. Only valid if *DATA_WIDTH* is 16 and *DIRECTION* is "INPUT", leave unconnected otherwise.
RXD_LOW	Output	8	8-bit high word of data from master. Must connect to parallel input of a *GP_COUNTx* or GP_DCMP block. (The hardware supports parallel output to general fabric routing however this is not currently implemented in *gp4par*.) Only valid if *DIRECTION* is "INPUT", leave unconnected otherwise.

8.47.3 Parameter Descriptions

Parameter	Type	Width	Function
DATA_WIDTH	Integer	5	Width of the SPI data buffer, in bits. Must be 8 or 16.
SPI_CPHA	Boolean	1	SPI clock phasing **When 0**: data eye is centered around active clock edge **When 1**: data eye is centered around inactive clock edge
SPI_CPOL	Boolean	1	SPI clock polarity **When 0**: active clock edge is rising **When 1**: active clock edge is falling
DIRECTION	String	N/A	**"INPUT":** Receive data from master; *RXD_** ports are valid **"OUTPUT":** Send data to master; *TXD_** ports are valid

8.47.4 Verilog Usage Example

The example shown in figure 66 needs to be written.

```
GP_SPI #(
    .FIXME(4)
) spi (
    .FIXME(garbage)
);
```

Figure 66: Example usage of GP_SPI

8.48 GP_SYSRESET: System Reset

8.48.1 Introduction

This primitive represents the system reset block. This is an asynchronous global reset of most on-chip logic (see the device datasheet for exact functionality) and is independent of the power-on reset, which cannot be configured.

Note that this block has slightly different timing than the power-on reset, which may cause glitches in some designs. To ensure consistent behavior between the runtime reset and boot, it may be necessary to instantiate the GP_POR block and gate glitching signals with RST_DONE.

8.48.2 Port Descriptions

Port	Type	Width	Function
RST	Input	1	System reset input. Must connect directly to to pin 2 of the device.

8.48.3 Parameter Descriptions

Parameter	Type	Width	Function
RESET_MODE	String	1	**"EDGE":** One-shot reset on rising edge of RST. **"LEVEL":** System is held in reset while RST is high.
EDGE_SPEED	Integer	9	Delay speed for the edge detector, in μs. Must be 4 or 500. Ignored if RESET_MODE is LEVEL.

8.48.4 Verilog Usage Example

The example shown in figure 67 resets the device when rst is high.

```
1  GP_SYSRESET #(
2      .RESET_MODE("LEVEL"),
3      .EDGE_SPEED(4)
4  ) reset_ctrl (
5      .RST(rst)
6  );
```

Figure 67: Example usage of GP_SYSRESET

8.49 GP_VDD: Power Connection

8.49.1 Introduction

This design element is internally used by *gp4par* as the source for all nets tied to a constant "1" value. It does not correspond to a hard IP block in the device.

It is documented here for completeness but should never be instantiated. If a constant "1" value is needed in a design, simply use the value `1'b1`.

8.49.2 Port Descriptions

Port	Type	Width	Function
OUT	Output	1	Constant value "1".

8.49.3 Parameter Descriptions

None.

8.49.4 Verilog Usage Example

The example shown in figure 68 ties the signal `foo` high.

```
1  wire foo;
2  GP_VDD vdd(
3      .OUT(foo)
4  );
```

Figure 68: Example usage of GP_VDD

8.50 GP_VREF: Voltage Reference

8.50.1 Introduction

This primitive represents an *abstracted view* of a reference voltage source. It buffers and optionally divides the incoming voltage, or a constant voltage from the on-chip bandgap reference, to drive the reference input to GP_ACMP cells, GP_DAC cells, and external reference output pins.

One GP_VREF block may drive any number of loads which use the same voltage, however internal device connectivity restrictions must be observed (for example, off-die reference inputs cannot be used for GreenPAK4 DACs). See the device datasheet for full details on which reference configurations are legal for which analog IP blocks.

Note that this block does *not* directly correspond to a physical GreenPAK device primitive (one GP_VREF may configure multiple reference generators attached to different hard IP) and thus the LOC constraint cannot be used with it.

8.50.2 Port Descriptions

Port	Type	Width	Function
VIN	Input	1	Input voltage, if required. Connect depending on the source of the reference: • Constant voltage: 1'b0 • Vdd: 1'b1 • DAC: Analog output from DAC • Off die: Analog input from IOB
VOUT	Output	1	The generated reference voltage.

8.50.3 Parameter Descriptions

Parameter	Type	Width	Function
VIN_DIV	Integer	4	Divider for input voltage. Legal values depend on the source of the reference: • Constant voltage: 1 • Vdd: 3 or 4 • DAC: 1 • Off die: 1 or 2
VREF	Integer	16	Constant output voltage, in mV. Legal values range from 50 to 1200 in 50 mV increments. If output voltage is not constant, do not specify this parameter.

8.50.4 Verilog Usage Example

The example shown in figure 69 drives a 750 mV reference voltage onto vref_750.

```
1  wire vref_750;
2  GP_VREF #(
3      .VIN_DIV(4'd1),
4      .VREF(16'd750)
5  ) vr750 (
6      .VIN(1'b0),
7      .VOUT(vref_750)
8  );
```

Figure 69: Example usage of GP_VREF

8.51 GP_VSS: Ground Connection

8.51.1 Introduction

This design element is internally used by *gp4par* as the source for all nets tied to a constant “0" value. It does not correspond to a hard IP block in the device.

It is documented here for completeness but should never be instantiated. If a constant “0" value is needed in a design, simply use the value `1'b0`.

8.51.2 Port Descriptions

Port	Type	Width	Function
OUT	Output	1	Constant value “0".

8.51.3 Parameter Descriptions

None.

8.51.4 Verilog Usage Example

The example shown in figure 70 ties the signal `foo` low.

```
1  wire foo;
2  GP_VSS vss(
3      .OUT(foo)
4  );
```

Figure 70: Example usage of GP_VSS

9 *gp4par* Command Line Usage

9.1 Introduction

All argument and value names are case sensitive.

The example shown in figure 71 runs place-and-route on the netlist Analog.json, targeting the SLG46620V, with top-level module Analog, and saves the generated bitstream to Analog-hdl.txt. Unused pins are pulled down with $10k\Omega$ resistors.

```
1  gp4par -p SLG46620V Analog.json -o Analog-hdl.txt --unused-pull down --
       unused-drive 10k
```

Figure 71: Example usage of *gp4par*

9.2 [file name]

The netlist filename must be supplied for all place-and-route operations. It may be included anywhere in the argument list, although we recommend it be the first argument for better readability of the command.

9.3 `--boot-retry`

The `--boot-retry` argument is optional. It must be followed by an integer from 1 to 4, specifying the number of times to re-try the boot process in case of a NVM read failure. If not specified, the retry count defaults to 1. This option is only supported for the SLG46140V.

9.4 `--constraints, -c`

The `--constraints` argument, which is also accepted as `-c`, is optional. If used, it must be immediately followed by the name of a Physical Constraints File (PCF).

9.5 `--debug`

The `--debug` argument is optional. When it is specified once, it causes *gp4par* to print many debugging messages, normally useful only for development of *gp4par*.

9.6 `--disable-charge-pump`

The `--disable-charge-pump` argument is optional. If set, the on-die charge pump is disabled, rather than automatically turning on when the supply voltage drops below 2.7V. This may cause analog components to malfunction if the supply voltage drops below 2.7V. It's unclear from Silego's documentation why this would ever be desirable, but the option is provided for completeness.

9.7 `--help`

The `--help` argument must be used alone, with no other arguments. It causes *gp4par* to print a usage example to the console and then quit.

9.8 `--io-precharge`

The `--io-precharge` argument is optional. If set, a nominal $2K\Omega$ resistor is connected in parallel with any pull-up/down resistors during boot, so that external signals will reach stable values sooner.

9.9 --ldo-bypass

The `--ldo-bypass` argument is optional. If set, the internal LDO is disabled and the `Vdd` pin drives the on-chip core voltage directly. This may result in lower power consumption, but requires the external supply to be a regulated 1.8V source.

9.10 --logfile, -l

The `--logfile` argument, which is also accepted as `-l`, is optional. If used, it must be immediately followed by a file name, where *gp4par* will write all diagnostic messages, even those suppressed by `--quiet`.

9.11 --logfile-lines, -L

The `--logfile-lines` argument, which is also accepted as `-L`, is optional. It is identical to `--logfile`, except that the file is line-buffered, that is every message is written to the file as soon as it is completely emitted by *gp4par*.

9.12 --nocolors

The `--nocolors` argument is optional. If set, do not use ANSI color escape sequences in stdout (logfiles never use colors). This is primarily intended for automated batch flows which do filtering of messages.

9.13 --output, -o

The `--output` argument is required for all place-and-route operations. It must be immediately followed by the filename to which the generated bitstream will be written.

9.14 --part, -p

The `--part` argument is required for all place-and-route operations. It must be immediately followed by the part number (ex: SLG46620V, SLG46140V).

9.15 --quiet, -q

The `--quiet` argument, which is also accepted as `-q`, is optional. When it is specified once, it causes *gp4par* to only print error and warning messages to the console; when it is specified twice, only error messages will be printed.

9.16 --read-protect

The `--read-protect` argument is optional. If set, prevent the bitstream from being read off the programmed device.

9.17 --stdout-only

The `--stdout-only` argument is optional. When specified, *gp4par* will print all messages to *stdout* regardless of severity. By default, messages of fatal, error, and warning severity are printed to *stderr* and all lower severity messages are printed to *stdout*.

This argument was implemented for easier integration with unit testing systems such as *CTest* and is unlikely to be useful in general usage.

9.18 --usercode

The `--usercode` argument is optional. If used, it must be immediately followed by a hexadecimal integer. This ID code is written to SRAM and/or NVM on the device during programming and can be used for distinguishing firmware revisions, board ID, or similar applications.

The length of the ID code varies by device family. For the *SLG46620V* it is one byte (two hex digits).

9.19 --unused-drive

The `--unused-drive` argument is optional. If used, it must be immediately followed by "`10k`", "`100k`", or "`1M`", to specify the nominal value in ohms of the pull-up/down resistor on unused I/O pins. The default behavior if not specified is "`1M`".

9.20 --unused-pull

The `--unused-pull` argument is optional. If used, it must be immediately followed by "`down`", "`up`", "`none`", or "`float`", to specify which direction to pull unused I/O pins. The default behavior if not specified is "`float`".

The behavior of "`none`" and "`float`" is identical; both names are accepted for convenience.

9.21 --verbose

The `--verbose` argument is optional. When it is specified once, it causes *gp4par* to print messages, useful in rare cases.

9.22 --version

The `--version` argument must be used alone, with no other arguments. It causes *gp4par* to print the version number (currently always 0.1) to the console and then quit.