<Name>

Release 0.0.1

Zero ASIC

Apr 26, 2024

USAGE

Welcome to the demo edition of Logik. In this edition, the following features are showcased:

- RTL to bitstream automation flow for FPGA/eFPGA architectures, powered by Silicon Compiler
- *logik_demo Example eFPGA Architecture Model* with the following resources:

4-input LUTs	6576
Registers	6576
GPIOs	64
UMI Interfaces	3
4KB Block RAMs	16
Multiply-Add Engines (MAEs)	16

- Example designs for reference in adopting the flow
- Documentation providing step-by-step guidance for setup and execution of the flow

CHAPTER

ONE

DOCUMENTATION TABLE OF CONTENTS

1.1 System Software Requirements

1.1.1 Supported Operating Systems

The following operating systems are supported without the requirement of running within a docker container:

- Ubuntu 20.04

Additional OS support is provided by running within a docker container.

1.1.2 General Purpose Software Requirements

The following general purpose software must be installed on your system to use this flow:

- Python 3.8 or higher
- git

1.1.3 Required EDA Software Tools

- Silicon Compiler
- Surelog
- Yosys
- VPR

For VHDL support, GHDL is also required

For SystemVerilog support, sv2v is also required.

For links to all EDA software Github repositories and documentation pages, please consult the *External Links for Open Source Tools*.

1.1.4 Optional EDA Software Tools

While not required to run the RTL-to-bitstream flow, HDL simulation support is required to run HDL simulations on provided examples.

Either of the following open-source simulators may be used for HDL simulation:

- Icarus Verilog
- Verilator

For waveform viewing, GTKWave is an available open source viewer:

- GTKWave

For links to all EDA software Github repositories and documentation pages, please consult the *External Links for Open Source Tools*.

1.2 Installing Required Software

There are two ways to install the above software tools:

1. Run within the Silicon Compiler tools docker image
2. Build from source yourself

1.2.1 Silicon Compiler Tools Docker Image Setup

First, install Docker according to the instructions for your operating system:

- Linux Docker installation
- macOS Docker installation
- Windows Docker installation

Once Docker is installed, launch the Docker Desktop application.

- Ubuntu: *Ctrl-Alt-T* or run the Terminal application from the Dash
- macOS: *Cmd-Space* to open Spotlight, then type *Terminal* and press *Enter*
- Windows: Open *Windows PowerShell* from the Start menu.

```
docker run -it -v "${PWD}/sc_work:/sc_work" ghcr.io/siliconcompiler/sc_runner:latest
```

1.2.2 Building Software From Source

Several resources are offered to assist in the installation process when tools need to be built from source.

Silicon Compiler's documentation provides reference install scripts for tools used in its flows. These can be used as startpoints for developing install scripts for your particular system and are available at https://docs.siliconcompiler.com/en/stable/user_guide/installation.html#external-tools

Additionally, the *External Links for Open Source Tools* page in this documentation points to documentation for open source software used in the flow. Each of these software tools maintains its own installation instructions. For the most up-to-date information on each software's installation procedure, its own documentation should be consulted.

1.3 Design Preparation

Prior to running the RTL-to-bitstream flow, design data must be aggregated and organized so that it can be found during flow execution. The effort to do this is minimal and outlined below.

1.3.1 Create a Working Directory

Because the flow is command-line driven, organization of files is performed at the command line rather than through an integrated development environment (IDE) project-based infrastructure. It is strongly recommended that users create a dedicated directory tree in which to store HDL files, constraint files, and their Silicon Compiler run script. It is also recommended (though not required) to execute the RTL-to-bitstream flow from the directory containing the Silicon Compiler run script.

1.3.2 Aggregate Input Files

The following file types should be aggregated

- HDL files
- Timing Constraints
- Pin Constraints

With these in place, a Silicon Compiler run script will have all required input files for execution.

1.4 Preparing the Silicon Compiler Run Script

Developing a Silicon Compiler run script for RTL-to-bitstream flow execution follows the same fundamental approach as developing a script for any Silicon Compiler flow execution. Additional resources for understanding Silicon Compiler fundamentals are available at docs.siliconcompiler.com

For most designs, the example Silicon Compiler run scripts provided with <tool_name> can be used as templates for creating your own. The commands used in these examples and the general method for constructing run scripts are described below.

Constructing a Silicon Compiler run script can be broken down into the following steps:

- *Import Modules*
- *Create Main Function*
- *Create Chip Object*
- *Select Part Name*
- *Register Packages (if needed)*
- *Set Input Source Files*
- *Adding Source Files From a Registered Package*
- *Set Timing Constraints*
- *Set Pin Constraints*
- *Add Options*
- *Configure Remote Execution (optional)*

- *Add Execution Calls*

1.4.1 Import Modules

All Silicon Compiler run scripts are pure Python scripts that import Silicon Compiler functionality like any other Python module. Similarly, the <tool_name> RTL-to-bitstream flow is enabled as a set of Python modules that integrate to Silicon Compiler.

The minimum import requirements in a Logik Silicon Compiler run script are:

```
import siliconcompiler
from logik.targets import logik_target
```

Additional module imports may be required depending on project-specific requirements.

1.4.2 Create Main Function

Since the Silicon Compiler run script is just a Python script, executing it from the command line requires the same infrastructure as any other Python script. In most design flows, the most convenient way to enable this will be to simply encapsulate the script in a main() function:

In Python, an executable main() function is implemented with the following code:

```
def main(<main_function_parameters (optional)>):

    #Insert your main function here

if __name__ == "__main__":
    main()
```

Experienced Python programmers may prefer to use their own scripting methodology for executing the script instead of the above. Any approach that conforms to both Python and Silicon Compiler requirements should work.

1.4.3 Create Chip Object

Silicon Compiler design information is encapsulated in a Python class called Chip. An instance of this class is required for all Silicon Compiler run scripts and is commonly referred to as the chip object.

The Chip class constructor requires one parameter: the name of the top level module in your RTL design. A complete Chip instantiation takes the form

```
chip = siliconcompiler.Chip('<your_top_module_name>')
```

Nearly all components of a Silicon Compiler run script are calls to member functions of this class instance; it should be the first (or nearly the first) line in your main function.

Throughout this documentation, "chip" will be used to refer to the Chip class instance. However, there is no requirement that the instance be assigned to this variable name.

1.4.4 Select Part Name

Silicon Compiler associates each FPGA/eFPGA architecture with a unique ID called a part name.

Note: As of this writing, the only part name that is enabled for use is "logik_demo"

In your Silicon Compiler run script, include the following call

```
chip.set('fpga', 'partname', 'logik_demo')
```

to select the logik_demo part as your selected part name.

1.4.5 Register Packages (if needed)

Designs with dependencies on third-party or packaged IP from previous projects may require a method for importing design IP from a source other than local working directories. In Silicon Compiler, such imports are supported via the Silicon Compiler package registry, and the import process is referred to as registering a package.

Registering a package is enabled with a dedicated Chip class member function called register_package_source(). For complete details on this function, refer to Silicon Compiler's documentation of the register_package_source() function.

An example use case for the package registry is shown below, outlining how to import a public Github repository so that its contents can be used as a package within Silicon Compiler. In this example, three parameters are provided to the register_package_source function: name, path, and ref. Name specifies a package name to be used when referring to the package elsewhere in code. Path specifies where Silicon Compiler can obtain the package; in this case, the package is obtained through Github. ref specifies to Silicon Compiler that the cloned Github repository should be checked out at a particular commit hash. Specifying ref is not necessary if the package is to be cloned from github on its main branch.

```
chip.register_package_source(
    name='picorv32',
    path='git+https://github.com/YosysHQ/picorv32.git',
    ref='a7b56fc81ff1363d20fd0fb606752458cd810552')
```

1.4.6 Set Input Source Files

All HDL source files must be added to the Silicon Compiler chip object for inclusion. For each HDL file, include the following call in your Silicon Compiler run script

```
chip.input(<your_hdl_file_name>)
```

Support is provided for Verilog, VHDL and SystemVerilog inputs.

Note: Mixed-language flows are not yet supported. All HDL source files must be written in the same language.

When using VHDL, it is required to add

```
chip.set('option', 'frontend', 'vhdl')
```

to your run script to trigger Silicon Compiler to execute ghdl prior to running synthesis.

When using SystemVerilog, it is required to add

```
chip.set('option', 'frontend', 'systemverilog')
```

to your run script to trigger Silicon Compiler to execute sv2v prior to running synthesis.

When using Verilog, the default frontend option, Surelog, is used, and no function call is required to enable it.

Note: Silicon Compiler supports additional front end options, including flows for high-level synthesis. For all front end compilation considerations not described above, please consult Silicon Compiler Frontend documentation

---

For large designs, it may be convenient to organize your HDL files into a directory tree that is processed using Python functions, so that the above calls can be embedded in loops.

Adding Source Files From a Registered Package

When importing IP from a package in the Silicon Compiler package registry, the same function calls are used as described above, but it is also necessary to specify the package name. The call takes the form:

```
chip.input('<your_file_name>', package='<package_name>')
```

1.4.7 Set Timing Constraints

Note: The demo architecture provided with this distrbution implements a unit delay model. Provided examples demonstrate the RTL-to-bitstream flow without an SDC file. Examples that include SDC files are planned for a future release.

Timing constraints must be provided in a single SDC file. The SDC file must be added to the Silicon Compiler chip object for inclusion. Include the call

```
chip.add('input', 'constraint', 'sdc', '<your_sdc_file_name>')
```

in your Silicon Compiler run script.

Note: If no SDC file is provided, the flow will still run to completion. Timing analysis will be disabled during the place and route steps.

1.4.8 Set Pin Constraints

Pin constraints may be provided in one of two files:

- A JSON pin constraints file (PCF)
- A VPR XML placement constraints file

Note: If you need to specify placement constraints for design logic blocks in addition to specifying pin constraints, the XML placement constraints file must be used.

JSON Pin Constraint Specification

The JSON pin constraint file is unique to this flow. For additional information on creating the JSON pin constraint file, see *Preparing Pin Constraints and Placement Constraints*.

The JSON placement constraints file must be added to the Silicon Compiler chip object for inclusion. Include the call

```
chip.input('<your_pcf_file_name>')
```

If your project defines itself as a package using Silicon Compiler's package registry, specify the package name as well:

```
chip.input('<your_pcf_file_name>', package=<your_package_name>)
```

in your Silicon Compiler run script

Note: The .pcf file extension must be used

VPR XML Placement Constraint Specification

VPR XML placement constraints are portable to any VPR-based place and route flow. For additional information on creating a VPR XML placement constraint file, see VPR's documentation for placement constraints.

The XML placement constraints file must be added to the Silicon Compiler chip object for inclusion. Include the call

```
chip.add('input', 'constraint', 'pins', '<your_xml_file_name>')
```

in your Silicon Compiler run script.

1.4.9 Add Options

Numerous options can be added to your run script to control Silicon Compiler behavior or configure tools in the RTL-to-bitstream flow to behave as desired. For complete Silicon Compiler option specifications, refer to Silicon Compiler's documentation for supported option settings.

In particular, any compiler directives that are required for HDL synthesis should be specified as Silicon Compiler options. These are furnished with Chip class member function calls of the form

```
chip.add('option', 'define', <compiler_directive>)
```

1.4.10 Configure Remote Execution (optional)

Silicon Compiler supports job submission to remote servers.

There are multiple ways to enable this execution model. Consult Silicon Compiler remote processing documentation for details.

1.4.11 Add Execution Calls

The final two lines of every run script should be the same:

```
chip.run()
chip.summary()
```

The run() call invokes the RTL-to-bitstream flow with all settings specified. The summary() call reports results of the run in tabular form. Included in the summary results are key design metrics such as FPGA resource utilization and tool execution runtimes.

1.5 RTL-to-Bitstream Flow Execution

Within your Python virtual environment or wherever you have installed Silicon Copmiler, simply execute your Silicon Compiler run script

```
python3 <your_sc_run_script>
```

1.6 Preparing Timing Constraints for VPR

Note: The demo architecture provided with this distrbution implements a unit delay model. Provided examples demonstrate the Logik without an SDC file. Examples that include SDC files are planned for a future release.

VPR is the place and route engine used in the Logik RTL to bitstream flow. To support VPR's timing-driven place and route flow, timing constraints are provided in a Synopsys Design Constraint (SDC) file.

The minimum requirement is to specify a target clock frequency using the *create_clock* constraint:

```
create_clock -period <float> <name of clock port>
```

For specifics on VPR's supported timing constraints, please consult the VPR SDC Commmand support page

1.7 Preparing Pin Constraints and Placement Constraints

Execution of the placement and routing steps with VPR do not require specifying constraints for the location of top-level ports in the user RTL design. When no constraints are specified, VPR will automatically choose locations for ports using its placement algorithm.

In some architectures (including the architecture supplied for the examples in this distribution), pin constraints are required in order to specify which of a subset of FPGA pins may be used as clock signals. More practically, constraints are typically required on all ports so that signals are routed to the correct physical locations for interaction with other components. Constraints may also be specified for the physical location of logic blocks.

The methods of supplying these constraints are described below.

1.7.1 VPR Placement Constraints

VPR natively supports pin constraint specification as a subset of its generic placement constraint settings. VPR placement constraints are specified in XML format. Complete documentation of this format is provided as part of VPR documentation.

To make use of these constraints it is necessary to have detailed information about the structure of the FPGA that is being targeted. FPGA resources are mapped in VPR to a grid laid out on an (X,Y) coordinate system.

Use of this format is required in order to constrain the placement of logic blocks. However, typically specifying placement constraints for logic blocks is not necessary in this RTL-to-bitstream flow.

If only pin constraints need to be specified, knowledge of this coordinate system is not necessary. Instead, a port-to-port mapping between user ports and eFPGA ports/FPGA pins can be specified in JSON format. This format is described below.

1.7.2 JSON Pin Constraints (PCF)

Specifying pin constraints in JSON format is supported so that users can specify a mapping between ports in their top-level RTL and port or pin names defined in the FPGA architecture. The required structure of the JSON file is referred to within Silicon Compiler as PCF format to distinguish it from other JSON files, and the .pcf file extension is used as an indicator that a file is of this type.

The PCF file is organized as a dictionary of JSON objects where keys in the dictionary are user port names and values are two-element dictionaries containing the port direction and the FPGA top level port name to which that user port should be mapped. The port direction is specified with the "direction" key and the the FPGA top level port name is specified with the "pin" key. This syntax is shown in the example below:

Example Pin Constraint Syntax

```
"resetn": {
  "direction": "input",
  "pin": "gpio_in[1]"
},
```

1.8 Automating Pin Constraint Generation for UMI Ports

A full UMI interface with both device and host request and response ports requires 1180 signals. To automate the pin constraint generation for UMI ports, two strategies are possible:

- Users may make use of UMI pin constraint templates provided with this software
- Users may develop their own automation scripts for generating UMI port constraints

1.8.1 Importing UMI Pin Constraints from a Template

eFPGA devices with UMI interfaces require an exact set of pin constraints that is used to connect user-defined UMI ports to the corresponding UMI port signals internal to the FPGA core. The required pin constraints are the same for all users, so a template for these constraints is provided for import into users' Silicon Compiler run scripts.

To make use of the pin constraints template, users must do the following:

- Integrate the constraints template into Python code for generating pin constraints.
- Follow the UMI port naming convention described below in their RTL

Constraints Template Integration

eFPGA UMI pin constraint templates are implemented as Python functions. For each UMI-enabled eFPGA device, there is a corresponding Python module called umi_pin_constraints.py provided as part of the templates submodule of this software. Each module contains a function called

```
generate_umi_pin_constraints
```

This can be imported into a user-defined pin constraints generator using

```
import logik.templates.<eFPGA part name>.umi_pin_constraints as <eFPGA part name>
```

The function takes a standard set of parameters that can be used to control constraints generation behavior. The API is as follows:

```
generate_umi_pin_constraints(fpga_ports_per_umi=300,
                             umi_cmd_width=32,
                             umi_data_width=128,
                             umi_addr_width=64,
                             umi_ports_used=[1, 2, 3],
                             port_types=["uhost_req",
                                         "uhost_resp",
                                         "udev_req",
                                         "udev_resp"],
                             umi_port_num_offset=1,
                             index_control_bits=True)
```

The meaning of these parameters is described in the table below

fpga_por	this parameter is used to define how many fpga pins are internally allocated for each UMI interface port. Except in rare circumstances, the default value should be used.
umi_cm	sets the number of bits in the UMI command bus
umi_dat	sets the number of bits in the UMI data bus
umi_add	sets the number of bits in the UMI source and destination address busses
umi_por	Each UMI port in is numbered. UMI port 0 is reserved for bitstream loading. UMI ports 1, 2, and 3 are user-accessible. This parameter allows specification of which of the UMI ports are used in user RTL code. Changing this value to enumerate only ports that are used prevents unused constraints from being generated.
umi_por	Defines an offset between FPGA internal bus indices and UMI port numbers. Except in rare circumstances, the default value should be used.
index_con	Defines whether constraints are generated such that UMI port ready and valid signals are specified with bus indices. In general, user RTL code with multiple UMI ports defined should set this to True, while user RTL code using a single UMI port should set this to False

Pin Constraint Template UMI Port Naming Conventions

The following tables show what signal names must be used in user RTL to name UMI interface signals for compatibility with eFPGA UMI pin constraint templates.

All signals with names ending in "ready" or "valid" may be specified as scalars or as multi-bit busses. When specified as scalars, calls to the generate_umi_pin_constraints constraint template generator functions must set the index_control_bits parameter to False.

Device Request Port

Ready	udev_req_ready
Command	udev_req_cmd
Data	udev_req_data
Source Addresss	udev_req_srcaddr
Destination Address	udev_req_dstaddr
Valid	udev_req_valid

Device Response Port

Ready	udev_resp_ready
Command	udev_resp_cmd
Data	udev_resp_data
Source Addresss	udev_resp_srcaddr
Destination Address	udev_resp_dstaddr
Valid	udev_resp_valid

Host Request Port

Ready	uhost_req_ready
Command	uhost_req_cmd
Data	uhost_req_data
Source Addresss	uhost_req_srcaddr
Destination Address	uhost_req_dstaddr
Valid	uhost_req_valid

Host Response Port

Ready	uhost_resp_ready
Command	uhost_resp_cmd
Data	uhost_resp_data
Source Addresss	uhost_resp_srcaddr
Destination Address	uhost_resp_dstaddr
Valid	uhost_resp_valid

1.8.2 Developing Custom Automation Scripts for Generating UMI Pin Constraints

It is also possible to generate a custom automation script for generating UMI port pin constraints. While always possible, it is only required if the UMI port naming convention documented above cannot be followed in user RTL.

Any automation technique that produces a valid JSON pin constraints file may be used. However, only a Python-based approach can be inlined with a Silicon Compiler run script.

1.9 Bitstream Formatting

Bitstream data is generated in this flow in multiple formats:

- FASM is the bitstream format supported within the Verilog-to-Routing flow. This format is a plain text, human-readable format originally developed for F4PGA and its predecessor projects. The genfasm step in this flow emits bitstreams in FASM format.
- A JSON intermediate representation of the bitstream is generated to provide a convenient intermediate representation between the FASM-formatted bitstream and the final binary bitstream. It is generally used for internal purposes only. See below for details on configuration bit organization.
- A binary representation of the bitstream is generated for delivery of the bitstream to other applications. Generation of this file is referred to in the flow as bitstream finishing.

Understanding the details of bitstream formatting is typically necessary only for writing software drivers to perform bitstream loading for particular FPGA chips.

Note: The logik_demo eFPGA architecture supported in this flow is supplied for demonstration purposes and is not available in physical chips.

The following sections outline how bitstreams are organized in a general sense without specifying the bitstream format for a particular FPGA or eFPGA architecture.

1.9.1 Working with FASM Bitstream Data

A FASM file is comprised of an enumeration of "features" that are enabled or have non-zero values. Each feature is assigned a plain-text label.

End user post-processing of the FASM file is not recommended as making use of the data requires precise knowledge of the underlying FPGA architecture. For all considerations related to parsing the format, please consult FASM documentation

1.9.2 Working with JSON Bitstream Data

JSON is used as the file format for storing an intermediate representation (IR) of bitstream data between FASM and binary formats.

The IR organizes the bitstream into a four-dimensional on-chip address space organized by array location. Each bit in the bitstream can thus be indexed by an X coordinate, Y coordinate, word address, and bit index in this IR.

For each FPGA architecture supported by the flow, a bitstream map file is provided that maps FASM features into this address space. The bitstream finishing step in Logik uses this bitstream map file to convert FASM to the IR.

Note: Architecture bitstream map files are cached by Silicon Compiler for use within the flow and may not be easily accessible by end users.

1.9.3 Working with Binary Bitstream Data

The binary representation of bitstream data consolidates the bitstream IR described above into a ROM-style format. The four-dimensional address space of individual bits is collapsed into a one-dimensional address space of bitstream words. The mapping is as shown in the table below

most significant bits	next most significant bits	least significant bits
Y coordinate	X coordinate	word address

When mapped into this address space, bitstream words are ordered from lowest address value to highest address value.

The binary format does not specify a word size; instead, this is dictated by the FPGA/eFPGA architecture. Words are treated as binary strings that, when read left to right, are read MSB to LSB. The packing of bitstream words into bytes in a binary file is dictated by Python's tofile() function. Implementations of a binary bitstream file reader should account for this so that when the bitstream is read back word ordering in this address space is preserved.

To make this more concrete, consider the logik_demo architecture. logik_demo is organized as a 37x31 array of elements. This means that six bits are needed to represent the X coordinate and five bits are needed to represent the Y coordinate. The number of word addresses needed at each (X,Y) coordinate in the array varies. To make a uniform address space, the maximum required word address dictates the number of bits of word address used. In the case of logik_demo, the maximum word address is 141, so eight bits of word address are needed. The binary bitstream address format is thus nineteen bits wide and organized as follows:

[18:14]	[13:8]	[7:0]
Y coordinate	X coordinate	word address

logik_demo uses 8-bit bitstream words. The ordering of the words in the binary file thus begins with word address 0 at array coordinate (0,0) and the last word is word address 141 at array coordinate (37, 31).

1.10 logik_demo Example eFPGA Architecture Model

Logik ships with access to and support for an example eFPGA architecture model called logik_demo. The logik_demo architecture does not target a specific process technology and so does not contain realistic timing information. Instead, a unit delay model is used, where each delay parameter in the architecture is set to 1 ns.

The features of the logik_demo architecture are summarized in the table below:

4-input LUTs	6576
Registers	6576
GPIOs	64
UMI Interfaces	3
4KB Block RAMs	16
Multiply-Add Engines (MAEs)	16

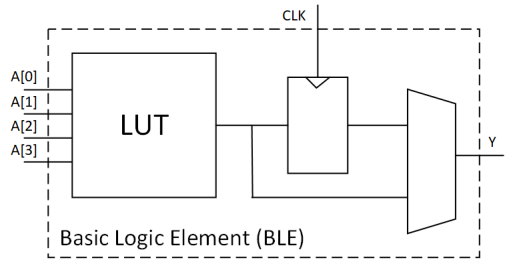
1.10.1 logik_demo Logic Resources

The following sections briefly describe the logic resources of the logik_demo eFPGA at a high level. Emphasis is placed on topics that are useful in studying the outputs of flow steps in Logik.

Lookup Tables and Registers

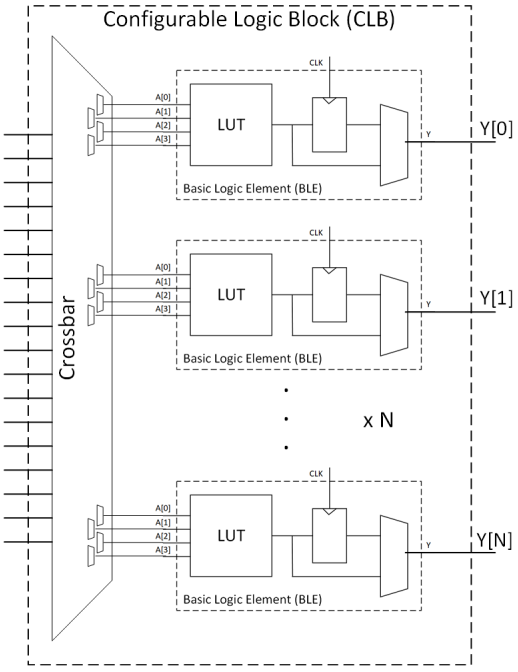
logik_demo contains 4-input lookup tables (LUTs) for implementing general purpose logic. During logic synthesis, digital logic is mapped to these LUTs.

Each LUT is paired with a D flip-flop to form a basic logic element (BLE), as shown in the block diagram below.



BLE inputs and LUT inputs are wired together and thus equivalent. Each flip flop can be configured with or without any of the following features: asynchronous set, asynchronous reset, built-in enable bit. Logic synthesis determines how many flip-flops are needed of each configuration type. Placement and routing determine whether or not to pair LUT with its flip-flop. If the lut is paired, the BLE output receives the flip-flop output; otherwise, it receives the LUT output. The BLE output interfaces to the FPGA's programmable interconnect.

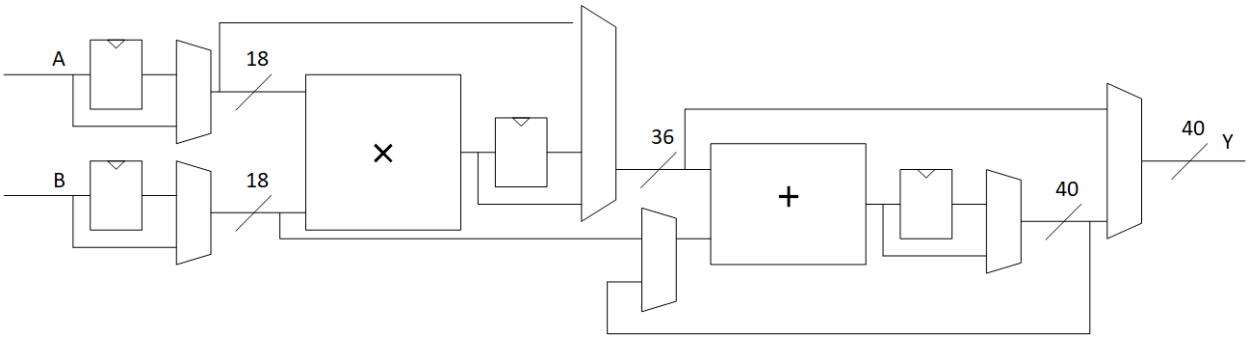
All LUTs are clustered into groups of eight to form configurable logic blocks (CLBs), as shown in in the block diagram below.



Each CLB shares 18 inputs amongst its eight LUTs. The CLB contains local interconnect encapsulated in a circuit called the crossbar that allows a subset of the 18 CLB inputs, the 8 LUT outputs, and the 8 flip flop outputs to be selected as input to each BLE. For clarity, feedback paths from LUT/BLE outputs to the crossbar are not shown in the diagram.

Multiply-Add Engine (MAE)

The multiply-add engine (MAE) is a configurable arithmetic unit suitable for use in many digital signal processing (DSP) applications. A block diagram is shown below



The key blocks are a multiplier and an adder, which can be used one at a time or together as a multiply-accumulate (MAC) unit. An 18x18 multiplier receives data directly from the logik_demo global interconnect or optionally from flip-flops to re-register data for improved throughput. The output of the multiplier can be routed directly out of the MAE to global interconnect to realize a purely combinational multiplier circuit or routed through a bank of flip-flops. In either case, this output can instead be routed through a 40-bit accumulator so that the MAE can be used as a MAC rather than a multiplier. Both the multiplier and accumulator can be registered to create a pipelined MAC. The accumulator can also be used by itself, in which case MAE inputs are routed directly to it.

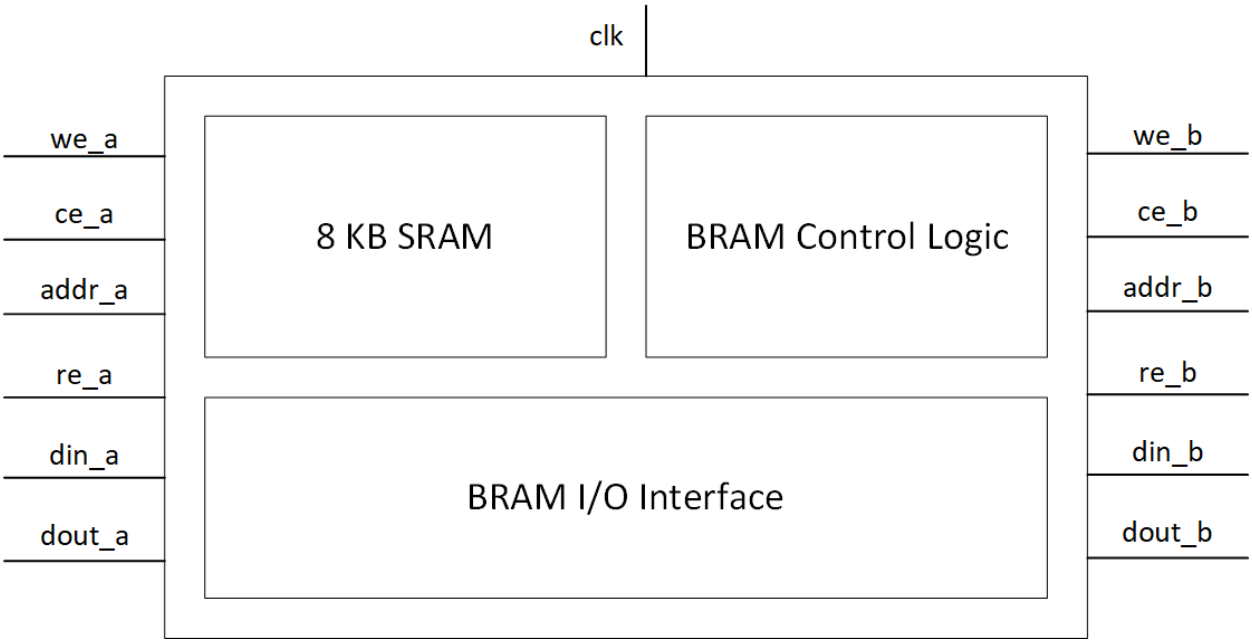
Each of these options maps to a specific MAE operational mode set by the configuration bitstream. The MAE operational modes each map to a particular netlist macro name during synthesis. The modes and their features are shown in the table below.

Function	Synthesis Macro Name	Ports Registered	Latency (clock cycles)
Multiply	efpga_mult	None	0
Multiply	efpga_mult_regi	Inputs	1
Multiply	efpga_mult_rego	Outputs	1
Multiply	efpga_mult_regio	Inputs and Outputs	2
MAC	efpga_macc	Unpipelined, Inputs unregistered	1
MAC	efpga_macc_regi	Unpipelined, Inputs registered	2
MAC	efpga_macc_pipe	Pipelined, Inputs unregistered	2
MAC	efpga_macc_pipe_regi	Pipelined, inputs registered	3
Adder	efpga_adder	None	0
Adder	efpga_adder_regi	Inputs	1
Adder	efpga_adder_rego	Outputs	1
Adder	efpga_adder_regio	Inputs and Outputs	2
Accumulator	efpga_macc_pipe	Inputs unregistered	1
Accumulator	efpga_macc_pipe_regi	Inputs registered	2

Block RAMs (BRAMs)

Each block RAM (BRAM) consists of 8KB of single-port SRAM, organized as 2K 32-bit words. The 8KB of SRAM can be used to emulate 4KB of simple dual-port SRAM with one write port, one read port, and one common clock. It may also be used to emulate 1K 64-bit words when configured as single-port RAM.

An abstract representation of the BRAM is shown in the block diagram below. The block diagram also shows the pinout of the BRAM. Two ports are provided, denoted with the port name suffixes _a and "_b". Each port has a write enable (we), chip enable (ce), read enable (re), address (addr), data input (din) and data output (dout) port. A common clock is shared between the two ports.



For end users of logik_demo, knowledge of the BRAM port list is not required. Typically, HDL code is used to infer BRAM instances, Yosys documentation of supported memory patterns discusses coding styles to help infer memories.

The BRAM instance shown above is not directly instantiated in synthesized netlists. Instead, an instance corresponding to a specific operating mode of the BRAM is instantiated. The BRAM operating modes are discussed next.

BRAM Operating Modes

Both single-port and dual-port operation support configurable bit widths and address depth. Technology mapping during synthesis automatically maps RAMs of size > 8KB into multiple block RAM instances and selects one of the operating modes. Each operating mode is delineated by a unique instance type in the synthesized netlist and specifies single-port or dual port, the effective word count, and the bits per word for the BRAM when in that mode.

The table below enumerates the BRAM operating modes, and the netlist macro names corresponding to each mode.

Port Type	#Words	Bits/Word	Synthesis Macro Name
single-port	1024	64	spram_1024x64
single-port	2048	32	spram_2048x32
single-port	4096	16	spram_4096x16
single-port	8192	8	spram_8192x8
single-port	16384	4	spram_16384x4
single-port	32768	2	spram_32768x2
single-port	65536	1	spram_65536x1
simple dual-port	1024	32	dpram_1024x32
simple dual-port	2048	16	dpram_2048x16
simple dual-port	4096	8	dpram_4096x8
simple dual-port	8192	4	dpram_8192x4
simple dual-port	16384	2	dpram_16384x2
simple dual-port	32768	1	dpram_32768x1

1.10.2 logik_demo eFPGA Port List

The table below enumerates logik_demo ports. Each of these ports may be specified in a JSON pin constraints file (PCF) to specify where a user port should be mapped during place and route.

For more information on PCF, see *Preparing Pin Constraints and Placement Constraints*

The logik_demo architecture has three types of I/O resources:

- Clocks – three clock signals are provided. All user clocks must map to one of these three ports. Designs with more than three clocks do not fit on this architectre.
- GPIOs – 64 general purpose I/Os are provided. Each GPIO is associated with one index of both the gpio_in port and the gpio_out port of the logik_demo top level. For example, once a user port is assigned to gpio_in[0], gpio_out[0] may not be used for a user output.
- UMI interfaces – logik_demo implements UMI interfaces as subsections of a wide I/O bus comprised of the umi_io_in and umi_io_out ports. Like the GPIOs, each bit of the UMI interface bus is associated with one index of both the umi_io_in and umi_io_out busses. For example, once a user port is assigned to umi_io_in[0], umi_io_out[0] may not be used for a user output.

The table below enumerates the I/O ports in logik_demo and specifies their bus widths. All indices in the bit range are legal options for specifying pin constraints, provided that the GPIO and UMI input/output usage restrictions described above are observed.

Port Name	Direction	Bit Range	Notes
clk	input	[2:0]	All user clocks must map to these ports
gpio_in	input	[63:0]	Pin locations are shared with gpio_out
gpio_out	output	[63:0]	Pin locations are shared with gpio_in
umi_io_in	input	[3599:0]	Pin locations are shared with umi_io_out
umi_io_out	output	[3599:0]	Pin locations are shared with umi_io_in

logik_demo UMI Port Mapping

Included in the Logik flow support for logik_demo is a reference template auto-generating constraints that the UMI interfaces to top level ports. In a complete eFPGA solution with UMI ports, the constraints generation template must correctly map eFPGA top level ports to specific locations elsewhere on chip that exchange UMI data between the eFPGA and other parts of the system. The tables below show how the logik_demo umi_io_in and umi_io_out busses map to the three UMI ports supported by the architecture.

For more information about how these ports are used in UMI interfaces, please consult the Signal UMI Layer section of the UMI Github repository README

Device Request Port

UMI Signal	UMI signal name	UMI Port 1 Signal	UMI Port 2 Signal	UMI Port 3 Signal
Ready	udev_req_ready	umi_io_out[889]	umi_io_out[2089]	umi_io_out[3289]
Command	udev_req_cmd	umi_io_in[632:601]	umi_io_in[1832:1801]	umi_io_in[3032:3001]
Data	udev_req_data	umi_io_in[888:761]	umi_io_in[2088:1961]	umi_io_in[3288:3161]
Source Addresss	udev_req_srcaddr	umi_io_in[760:697]	umi_io_in[1960:1897]	umi_io_in[3160:3097]
Destination Address	udev_req_dstaddr	umi_io_in[696:633]	umi_io_in[1896:2133]	umi_io_in[3096:3033]
Valid	udev_req_valid	umi_io_in[600]	umi_io_in[1800]	umi_io_in[3000]

Device Response Port

UMI Signal	UMI signal name	UMI Port 1 Signal	UMI Port 2 Signal	UMI Port 3 Signal
Ready	uhost_req_ready	umi_io_in[1189]	umi_io_in[2389]	umi_io_in[3589]
Command	uhost_req_cmd	umi_io_out[932:901]	umi_io_out[2132:2101]	umi_io_out[3332:3301]
Data	uhost_req_data	umi_io_out[1188:1061]	umi_io_out[2388:2261]	umi_io_out[3588:3461]
Source Addresss	uhost_req_srcaddr	umi_io_out[1060:997]	umi_io_out[2260:2197]	umi_io_out[3460:3397]
Destination Address	uhost_req_dstaddr	umi_io_out[996:933]	umi_io_out[2196:2133]	umi_io_out[3396:3333]
Valid	uhost_req_valid	umi_io_out[900]	umi_io_out[2100]	umi_io_out[3300]

Host Request Port

UMI Signal	UMI signal name	UMI Port 1 Signal	UMI Port 2 Signal	UMI Port 3 Signal
Ready	uhost_req_ready	umi_io_in[289]	umi_io_in[1489]	umi_io_in[2689]
Command	uhost_req_cmd	umi_io_out[32:1]	umi_io_out[1232:1201]	umi_io_out[2432:2401]
Data	uhost_req_data	umi_io_out[288:161]	umi_io_out[1488:1361]	umi_io_out[2688:2561]
Source Addresss	uhost_req_srcaddr	umi_io_out[160:97]	umi_io_out[1360:1297]	umi_io_out[2560:2497]
Destination Address	uhost_req_dstaddr	umi_io_out[96:33]	umi_io_out[1296:1233]	umi_io_out[2496:2433]
Valid	uhost_req_valid	umi_io_out[0]	umi_io_out[1200]	umi_io_out[2400]

Host Response Port

UMI Signal	UMI signal name	UMI Port 1 Signal	UMI Port 2 Signal	UMI Port 3 Signal
Ready	uhost_resp_ready	umi_io_out[589]	umi_io_out[1789]	umi_io_out[2989]
Command	uhost_resp_cmd	umi_io_in[332:301]	umi_io_in[1532:1501]	umi_io_in[2732:2701]
Data	uhost_resp_data	umi_io_in[588:461]	umi_io_in[1788:1661]	umi_io_in[2988:2861]
Source Addresss	uhost_resp_srcaddr	umi_io_in[460:397]	umi_io_in[1660:1597]	umi_io_in[2860:2797]
Destination Address	uhost_resp_dstaddr	umi_io_in[396:333]	umi_io_in[1596:1533]	umi_io_in[2796:2733]
Valid	uhost_resp_valid	umi_io_in[300]	umi_io_in[1500]	umi_io_in[2700]

1.10.3 Notes on logik_demo Model for Developers

Note: The developer model for adding new FPGAs to Logik is a work in progress. Collaboration is strongly recommended to assist in the bringup of a new FPGA architecture in Logik.

Developers interested in studying the logik_demo model as a reference model for adding a new FPGA to Logik may wish to understand the model in more detail. Below is a summary of the required FPGA model files that developers must provide to support an FPGA in Logik.

- A VPR architecture XML file is required. For bitstream generation support, it must contain FASM feature metadata for all required features.
- A VPR routing resource graph XML file is also required. While VPR supports flows that do not use this file, routing resource graph XML metadata is required for bitstream generation with genfasm.
- A bitstream map file is required for Logik bitstream finishing. The bitstream map file is a JSON document that embeds the location of each FASM feature within a four-dimensional address space defined by the architecture's bitstream loading sequence.
- A constraints map file is required for support of JSON pin constraints (PCF) to VPR's native XML placement constraints format.
- For support of technology mapping by Yosys of FPGA hard macros, Yosys-compatible Verilog models are required. These must be co-designed with the VPR architecture XML to ensure compatibility across all steps of the flow.

In addition to these model files, a part driver must be added to Logik for any group of related FPGAs or eFPGAs (referred to as FPGA/eFPGA families). The part driver may share information between multiple FPGAs in a family, or define data only for a single FPGA/eFPGA. The part driver is a Python file created as a module within the Logik Python package hierarchy. This means that the part driver must be formally integrated into a Logik release.

Within the logik_demo part driver provided with Logik, these files are specified and registered as Silicon Compiler packages. Silicon Compiler is then able to acquire the files for use in the Logik flow. Additional FPGA/eFPGA design-specific data required by CAD tools, such as the input counts of LUTs in the FPGA or the number of routing resources, must also be specified. Consult the logik_demo example for reference on these details.

1.11 External Links for Open Source Tools

Below is a set of links to documentation for open source tools used by Logik. For each tool, the "Github" to go to the project github page. The "Documentation" link links to the project's online documentation/home page. If a tool does not contain a documentation link below, please consult its Github repository README for documentation.

F4PGA (FASM format authors)	F4PGA Github	F4PGA Documentation
GHDL	GHDL Github	GHDL Documentation
GTKWave	Gtkwave Github	Gtkwave Documentation
Icarus Verilog	Icarus Github	Icarus Documentation
Silicon Compiler	SC Github	SC Documentation
Surelog	Surelog Github	
sv2v	sv2v Github	
Verilator	Verilator Github	Verilator Documentation
Verilog-to-Routing (VPR)	VPR Github	VPR Documentation
Yosys	Yosys Github	Yosys Documentation

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search