LegUp Documentation

Release 4.0

University of Toronto

October 17, 2015

1 Getting Started **3**
1.1 Installation . . . 3
1.2 Hybrid Flow: Hardware/Software Partitioning . . . 5
1.3 Readings . . . 6

2 User Guide **7**
2.1 Pure Hardware Flow . . . 9
2.2 Loop Pipelining . . . 10
2.3 Parallel Flow . . . 12
2.4 Hardware/Software Hybrid Flow . . . 12
2.5 Pure Software Flow . . . 18
2.6 Custom Verilog & Propagating I/O . . . 18

3 Hardware Architecture **21**
3.1 Modules . . . 21
3.2 Memory architecture . . . 22
3.3 Local Memories . . . 23
3.4 Global Memories . . . 23
3.5 Memory Controller . . . 24
3.6 Function Calls . . . 25
3.7 Signed/Unsigned . . . 25
3.8 Mult-dimensional Arrays . . . 25
3.9 Functional Units . . . 25
3.10 Structs . . . 26
3.11 Avalon Signals . . . 26

4 Programmer's Manual **29**
4.1 LLVM Backend Pass . . . 29
4.2 LLVM Frontend Passes . . . 30
4.3 Execution Flow . . . 30
4.4 Test Suite . . . 32
4.5 LLVM Passes . . . 32
4.6 Source Files . . . 33
4.7 Important Classes . . . 33
4.8 Binding . . . 37
4.9 LegupConfig . . . 42
4.10 PreLTO . . . 42
4.11 LLVM . . . 42
4.12 Tips/Tricks . . . 43
4.13 LegUp Quality of Results . . . 44

5 Debugging 47
5.1 Installation . . . 47
5.2 Configuring your Design . . . 48
5.3 Compiling your Design . . . 49
5.4 Using the Debugger . . . 49
5.5 Reading Variables . . . 52
5.6 SW/RTL Discrepancy Checking . . . 53
5.7 References . . . 54

6 Hybrid Partition Analysis 55
6.1 Software Cycles . . . 55
6.2 Hybrid Cycles . . . 56

7 LegUp over PCIe 63
7.1 User information . . . 63
7.2 Future work . . . 64

8 LegUp on Xilinx FPGAs 65
8.1 Options . . . 65
8.2 Makefile Support . . . 65
8.3 Devices . . . 66
8.4 Examples . . . 66

9 Frequently Asked Questions 67

10 Constraints Manual 71
10.1 Most Commonly Used Constraints . . . 71
10.2 Other Advanced Constraints . . . 88

11 Release Notes 185
11.1 LegUp 4.0 . . . 185
11.2 LegUp 3.0 . . . 186
11.3 LegUp 2.0 . . . 187
11.4 LegUp 1.0 . . . 189

Bibliography 191

Index 193

LegUp is a high-level synthesis research infrastructure being actively developed at the University of Toronto since early 2010. Our goal is to allow researchers to experiment with new high-level synthesis algorithms without building a new infrastructure from scratch. Our long-term vision is to make FPGA programming easier for software developers.

The documentation is comprised of the following sections:

- *Getting Started*: Installation and a quick start guide
- *User Guide*: How to use LegUp to generate hardware
- *Hardware Architecture*: Details of the synthesized circuit architecture
- *Programmer's Manual*: Describes the layout of the LegUp codebase and the class hierarchy
- *Constraints Manual*: Constraints manual
- *Frequently Asked Questions*: Frequently asked questions
- *Release Notes*: New features and known problems with each release

If you have questions, patches, and suggestions please email them to the LegUp development mailing list, legup-dev@legup.org, or email us directly at legup@eecg.toronto.edu.

If you find a bug in LegUp, please file it in Bugzilla.

CHAPTER ONE

GETTING STARTED

It is strongly suggested that you use the Virtual Machine image provided on the LegUp site:

```
http://legup.eecg.utoronto.ca/download.php
```

The VM comes with LegUp 4.0 and all the required software and libraries pre-installed. This is cleaner than installing everything manually, and thus is the recommended method of downloading LegUp.

If you prefer to install LegUp from scratch instead, then this guide should quickly get you started using LegUp to synthesize C into Verilog. We assume that you are using a Linux 32/64-bit environment, we have not tested LegUp on Windows or Mac OS.

1.1 Installation

1.1.1 Required Packages on Ubuntu

To install all the required packages, first run:

```
sudo apt-get update
```

Then run:

```
sudo apt-get install git tcl8.5-dev dejagnu expect texinfo build-essential \
liblpsolve55-dev libgmp3-dev automake libtool clang-3.5 libmysqlclient-dev \
qemu-system-arm qemu-system-mips gcc-4.8-plugin-dev libc6-dev-i386 meld \
libqt4-dev libgraphviz-dev libfreetype6-dev buildbot-slave libXi6:i386 \
libpng12-dev:i386 libfreetype6-dev:i386 libfontconfig1:i386 libxft2:i386 \
libncurses5:i386 libsm6:i386 libxtst6:i386 vim gitk kdiff3 gxemul libgd-dev \
openssh-server mysql-server python3-mysql.connector python3-serial \
python3-pyqt5
```

Note: When installing the `mysql-server` package, you will be asked to set the password for user 'root'. By default, the LegUp debug tool assumes 'root' and 'letmein' as user and password for `mysql`. If a different username or password is used, please update these two paramters `DEBUG_DB_USER` and `DEBUG_DB_PASSWORD` accordingly in the file `$(LEGUP_REPO)/examples/legup.tcl`, after you downloaded LegUp repository.

For clang, setup a symlink with:

```
sudo ln -s /usr/bin/clang-3.5 /usr/bin/clang
```

If you're running version 11.10 of Ubuntu or older, run:

```
sudo apt-get install gxemul
```

If you're running version 12.04 of Ubuntu or newer, download and install the appropriate gxemul_0.6.0.2 for your architecture from:

```
http://archive.ubuntu.com/ubuntu/pool/universe/g/gxemul/
```

1.1.2 Compile LegUp Source

Download and compile the LegUp source:

```
tar xvzf legup-4.0.tar.gz
cd legup-4.0
make
export PATH=$PWD/llvm/Release/bin:$PATH
```

Note: If you want to speed up the compilation run make using 4 parallel threads: `make -j4`

1.1.3 Modelsim and Quartus

You will need Modelsim to simulate the Verilog and Quartus to synthesize the Verilog for an FPGA. You can download Quartus Web Edition and Modelsim for free from Altera. We recommend using Quartus 15.0.

After installing Quartus update your environment to add `sopc_builder` to your path:

```
export QUARTUS_ROOTDIR=/opt/altera/15.0/quartus/
export PATH=/opt/altera/15.0/quartus/sopc_builder/bin/:$PATH
```

Note: You must edit the path above to point to your particular Quartus installation location.

1.1.4 Test Suite

Now you can run the test suite to verify your installation:

```
cd examples
runtest
```

The test suite uses LegUp to synthesize hardware for all of our examples and then simulates the hardware in Modelsim to verify correctness. You should see the following output after a few minutes:

```
            ===  Summary ===

# of expected passes                476
```

For further details see *Test Suite*

1.1.5 Quick Start Tutorial

To get started with LegUp lets try synthesizing a simple square root approximation into hardware. This example is already provided in our test suite:

```
cd examples/sra
make
```

Take a look at `sra.v`. You've just synthesized your first C program into hardware!

Let's try to simulate the hardware in Modelsim. To do this, run the following inside the sra directory:

```
make v
```

Note: Make sure you have **vsim** (Modelsim) on your path:

```
$ which vsim
/opt/modelsim/install/modeltech/linux/vsim
```

Your simulation output should look something like:

```
# ...
# Loading work.main_tb(fast)
# Loading work.ALTERA_DEVICE_FAMILIES(fast)
# Loading work.ALTERA_MF_MEMORY_INITIALIZATION(fast)
# Loading work.ram_dual_port(fast__1)
# run 7000000000000000ns
# Result:        100
# RESULT: PASS
# At t=             1310000 clk=1 finish=1 return_val=       100
# Cycles:                    63
# ** Note: $finish    : sra.v(4330)
#    Time: 1310 ns  Iteration: 3  Instance: /main_tb
```

The circuit produced the expected result of 100 and took 63 clock cycles to complete. Now try synthesizing sra.v with Quartus targeting a Cyclone II FPGA:

```
make p
make f
```

Quartus should have no errors:

```
...
Info: Quartus II Shell was successful. 0 errors, 6 warnings
Info: Peak virtual memory: 83 megabytes
Info: Processing ended: Thu Dec 15 21:00:15 2011
Info: Elapsed time: 00:00:48
Info: Total CPU time (on all processors): 00:00:52
```

Note: If you want to use sra as a template for another program make sure to remove the lines **NO_OPT** and **NO_INLINE** from the Makefile. See *User Guide*

1.2 Hybrid Flow: Hardware/Software Partitioning

LegUp can compile an entire C program to hardware as above, or it can compile user designated functions to hardware while remaining program segments are executed in software on the soft TigerMIPS processor. This is referred to as the *hybrid* flow.

For example, let's accelerate the **float64_add** function of the `dfadd` CHStone benchmark:

```
cd examples/chstone_hybrid/dfadd
gedit config.tcl
```

To specify which functions should be in hardware, use the **set_accelerator_function** tcl command. Add the following to `config.tcl`:

```
set_accelerator_function "float64_add"
```

The **float64_add** function plus all of its descendants will now be compiled to a hardware accelerator. The **set_accelerator_function** can be used more than once to accelerate multiple functions. Now, run:

```
make hybridsim
```

Note: `make hybridsim` runs Altera's `sopc_builder` script which requires X11

LegUp will generate the hardware accelerator and simulate the system in Modelsim. The output should look like:

```
# ...
# Result: 46
#
# RESULT: PASS
#
# counter =                10651
```

The counter variable gives the total number of cycles for the complete execution of the program, which in this case is 10651 cycles, and the circuit produced the expected result of 46. `# RESULT: PASS` indicates that the test case passed. Each test runs a set of test vectors with known outputs, like a hardware built-in self test. The test passes if the simulation output matches the expected output.

1.3 Readings

Now that you've tried out LegUp you should read these publications. They will help you better understand our code:

- LegUp publications
- LegUp paper in FPGA 2011
- Introduction to High-Level Synthesis by Daniel Gajski
- CHStone benchmark paper
- SDC Scheduling paper

CHAPTER

TWO

USER GUIDE

LegUp accepts a vanilla ANSI C file as input, that is, no pragmas or special keywords are required, and produces a Verilog hardware description file as output that can be synthesized onto an Altera FPGA. Any C **printf** statements are converted to Verilog **$display** statements that are printed during a modelsim simulation, making it possible to compile the same C file with gcc and check its output against the simulation.

LegUp has two different synthesis *flows*:

- Pure hardware: Synthesizes the whole C file into hardware with no soft processor
- Hybrid: Execute a portion of the C file on the TigerMIPS soft processor and synthesize the rest into hardware

The LegUp synthesis flow is driven by TCL scripts and Makefiles mainly in the `examples` directory. The `examples` directory contains sample C benchmark programs that make up the LegUp test suite. There is on TCL script and are four global Makefiles:

- `legup.tcl`: Defines all of the default user defined settings that can be used to guide LegUp's hardware/software generation. Users may have to edit this Makefile to change the following variables:
  - **FAMILY**: specify the target FPGA device family, either: CycloneII (default) or StratixIV
  - **LEGUP_SDC_PERIOD**: specify the target clock period constraint (ns). Defaults to 15ns for CycloneII and 5ns for StratixIV.
- `Makefile.config`: This Makefile defines all the Makefile global variables. Most of these global variables are read from the settings with the TCL files and should not be modified in this file.
- `Makefile.common`: This is the central Makefile that is included by all other Makefiles in the `examples` directory. It includes all of the primary make targets that users would wish to invoke. This file includes all the other Makefiles in the example directory.
- `Makefile.ancillary`: This Makefile contains secondary make targets that are called by the primary targets in `Makefile.common`. Users should not call these target directly.
- `Makefile.aux`: This Makefile contains old or infrequently used targets. This target should not be used, as they may no longer work and are not tested on a regular basis.

If we look inside the `examples/array` directory there are three files:

- `array.c`: This is the C file we wish to synthesize into hardware.
- `dg.exp`: Test suite file. See *Test Suite*
- `Makefile`

Note: That a fourth file, `config.tcl`, can be included to overwrite any of the default settings in `legup.tcl`, such as the target board or whether loop pipeline is enabled. An example of enabling loop pipelining in the **Loop Pipelining** section below.

The contents of `Makefile` are:

```
NAME=array
ifeq ($(NO_OPT),)
    NO_OPT=1
endif
ifeq ($(NO_INLINE),)
    NO_INLINE=1
endif
LEVEL = ..
include $(LEVEL)/Makefile.common
```

There are two important environment variables defined here:

- **NO_OPT**: Disable all compiler optimizations. This passes the flag `-O0` to `clang`.
- **NO_INLINE**: Disable all function inlining

The reason we turn off all optimizations and disable inlining for this simple benchmark is for testing purposes. We want to avoid the LLVM compiler optimizing away the whole program. For most complex programs (like CHStone) you will want to remove these Makefile variables to enable full LLVM optimizations.

Note that this Makefile includes `examples/Makefile.common`, which uses the **NAME** and **LEVEL** variables to customize the LegUp synthesis flow for this specific benchmark.

The central `examples/Makefile.common` defines the LegUp synthesis flow. To run LegUp use the following commands:

- **make**: run the pure hardware flow
- **make hybrid**: run the hybrid flow.
- **make sw**: run the software only flow.

A few other useful commands for the pure hardware flow are:

- **make v**: simulate the output Verilog file with Modelsim in textual mode
- **make w**: simulate the output Verilog file with Modelsim and show waveforms
- **make p**: create a Quartus project in the current directory
- **make q**: run the Quartus mapper on the Verilog file
- **make f**: run a full Quartus compile Verilog file
- **make watch**: debug the hardware implementation by comparing a Modelsim simulation trace to a pure software trace. See *make watch*.
- **make dot**: compile all .dot graph files in the current directory into .ps files

A few other useful commands for the hybrid and software only flows are:

- **make hybridsim**: run the hybrid flow and simulate the output Verilog with Modelsim
- **make swsim**: run the software only flow and simulate the MIPS processor executing the software with Modelsim
- **make hybridquartus**: run a full Quartus compile on the hybrid system created with make hybrid
- **make emul**: simulate MIPS assembly on GXemul MIPS emulator

Note: For examples that use the hybrid flow look in `examples/chstone_hybrid/`

2.1 Pure Hardware Flow

The pure hardware flow synthesizes the entire C file into hardware with no soft processor. To run this flow use:

```
make
```

This is similar to other high-level synthesis tools. To look at an example, change into the `legup/examples/array` directory and type `make`. This will run the following commands:

```
../mark_labels.pl array.c > array_labeled.c
```

`mark_labels.pl` annotates loop that have labels as required for loop pipelining.

```
clang-3.5 array_labeled.c -emit-llvm -c -fno-builtin -I ../lib/include/
-m32 -I /usr/include/i386-linux-gnu -O0 -mllvm -inline-threshold=-100
-fno-inline -fno-vectorize -fno-slp-vectorize -o array.prelto.1.bc
```

`clang` compiles the `array.c` file into LLVM byte code file: `array.prelto.1.bc`. Note that inlining is off (`-mllvm -inline-threshold=-100`) and optimizations are off (`-O0`). The next command:

```
../../llvm/Release+Asserts/bin/opt -mem2reg -loops -loop-simplify <
array.prelto.cv.bc > array.prelto.2.bc
```

This uses the LLVM `opt` command to run a LegUp LLVM passes called `-mem2reg`, `-loops` and `-loop-simplify`, which performs promotes memory references to be register references and general loop optimization. The command produces `array.prelto.2.bc`. The next command:

```
../../llvm/Release+Asserts/bin/opt
-load=../../llvm/Release+Asserts/lib/LLVMLegUp.so
-legup-config=../legup.tcl  -disable-inlining -disable-opt
-legup-prelto < array.prelto.linked.1.bc > array.prelto.6.bc
```

This uses the LLVM `opt` command to run a LegUp LLVM pass called `-legup-prelto`, which performs LLVM intrinsic function lowering and produces `array.prelto.6.bc`. The next command:

```
../../llvm/Release+Asserts/bin/opt
-load=../../llvm/Release+Asserts/lib/LLVMLegUp.so
-legup-config=../legup.tcl  -disable-inlining -disable-opt
-std-link-opts < array.prelto.6.bc -o array.prelto.bc
```

This uses the LLVM `opt` command to run a LLVM pass called `-std-link-opts`, which performs standard LLVM link-time optimizations and produces `array.prelto.bc`. The next command:

```
../../llvm/Release+Asserts/bin/llvm-link  array.prelto.bc
../lib/llvm/liblegup.bc ../lib/llvm/libm.bc -o array.postlto.6.bc
```

This uses the LLVM `llvm-link` command to link in one of llvm's libraries, `libm.bc` and produces `array.postlto.6.bc`. The next command:

```
../../llvm/Release+Asserts/bin/opt -internalize-public-api-list=main
-internalize -globaldce array.postlto.6.bc -o array.postlto.8.bc
```

This uses the LLVM `opt` command to run a LLVM pass called `-globaldce`, which performs standard LLVM dead-code elimination (DCE) to remove all unused functions and produces `array.postlto.8.bc`. The next command:

```
../../llvm/Release+Asserts/bin/opt
-load=../../llvm/Release+Asserts/lib/LLVMLegUp.so
-legup-config=../legup.tcl  -disable-inlining -disable-opt
-instcombine -std-link-opts < array.postlto.8.bc -o array.postlto.bc
```

This uses the LLVM `opt` command to run a LLVM pass called `-std-link-opts`, which performs standard LLVM link-time optimizations and produces `array.postlto.bc`. The next command:

```
# iterative modulo scheduling
../../llvm/Release+Asserts/bin/opt
-load=../../llvm/Release+Asserts/lib/LLVMLegUp.so
-legup-config=../legup.tcl  -disable-inlining -disable-opt -basicaa
-loop-simplify -indvars2  -loop-pipeline array.postlto.bc -o array.1.bc
../../llvm/Release+Asserts/bin/opt
-load=../../llvm/Release+Asserts/lib/LLVMLegUp.so
-legup-config=../legup.tcl  -disable-inlining -disable-opt
-instcombine array.1.bc -o array.bc
```

This uses the LLVM `opt` command to run a LegUp LLVM pass called `-loop-pipeline`, which pipeline loops if pipelining is enabled and produces `array.bc`. The following commands:

```
../../llvm/Release+Asserts/bin/llvm-dis array.prelto.linked.bc
../../llvm/Release+Asserts/bin/llvm-dis array.prelto.6.bc
../../llvm/Release+Asserts/bin/llvm-dis array.prelto.bc
../../llvm/Release+Asserts/bin/llvm-dis array.postlto.bc
../../llvm/Release+Asserts/bin/llvm-dis array.postlto.6.bc
../../llvm/Release+Asserts/bin/llvm-dis array.postlto.8.bc
../../llvm/Release+Asserts/bin/llvm-dis array.1.bc
../../llvm/Release+Asserts/bin/llvm-dis array.bc
```

Disassemble the LLVM bytecode using `llvm-dis` and create text files holding the LLVM intermediate representation for all stages of the LegUp flow. The final command:

```
../../llvm/Debug+Asserts/bin/llc
-legup-config=../legup.tcl -march=v array.bc -o array.v
```

This uses the LLVM `llc` compiler targeting architecture `v` (Verilog). `llc` reads the `examples/legup.tcl` file containing LegUp synthesis parameters, and including a device database file for the selected family, which holds the delay and area information for hardware operations. Finally, `llc` calls LegUp backend pass (see `runOnModule()` in `llvm/lib/Target/Verilog/LegupPass.cpp`) to produce the Verilog file `array.v` from the LLVM bytecode `array.bc`.

2.2 Loop Pipelining

Loop pipelining is a feature introduced in LegUp 3.0. To look at some examples that utilize loop pipelining navigate to the `legup/examples/pipeline/simple` directory. Take a look in the `Makefile`

```
NAME=simple
LOCAL_CONFIG = -legup-config=config.tcl

# don't unroll the loop
CFLAG += -mllvm -unroll-threshold=0

LEVEL = ../..
include $(LEVEL)/Makefile.common
```

The **LOCAL_CONFIG** variable specifies a local configuration tcl file named `config.tcl` in the current directory. Also note that we've turned the LLVM loop unroll threshold to 0 so that the four iteration loop in this example is not unrolled. Open `config.tcl`:

```
source ../config.tcl
```

```
loop_pipeline "loop"

set_parameter LOCAL_RAMS 1
```

The *loop_pipeline* tcl command specifies that we wish to pipeline the loop with label “loop” in `simple.c`. The *LOCAL_RAMS* tcl command causes LegUp to use local memory when possible, instead of storaging arrays in a global memory.

Open `../config.tcl`:

```
source ../../legup.tcl

set_parameter PRINTF_CYCLES 1

set_operation_latency multiply 0

set_project StratixIV DE4-530 Tiger_DDR2
```

The *PRINTF_CYCLES* tcl command causes ModelSim to print the cycle count each time `printf` is called. The *set_operation_latency multiply* tcl command causes LegUp to use multiplier that have a latency of zero cycles. The *set_project* tcl command tells LegUp what FPGA Family, Development Kit and project to target.

Open `simple.c` and verify the for loop has a label:

```
loop: for (i = 0; i < N; i++) {
```

Now run `make` and `make v`. Your modelsim output should look like:

```
# Cycle:          52 Time:        1090    Loop body
# Cycle:          53 Time:        1110    Loop body
# Cycle:          53 Time:        1110    a[          0] =   1
# Cycle:          53 Time:        1110    b[          0] =   5
# Cycle:          54 Time:        1130    Loop body
# Cycle:          54 Time:        1130    a[          1] =   2
# Cycle:          54 Time:        1130    b[          1] =   6
# Cycle:          55 Time:        1150    Loop body
# Cycle:          55 Time:        1150    a[          2] =   3
# Cycle:          55 Time:        1150    b[          2] =   7
# Cycle:          55 Time:        1150    c[          0] =   6
# Cycle:          56 Time:        1170    a[          3] =   4
# Cycle:          56 Time:        1170    b[          3] =   8
# Cycle:          56 Time:        1170    c[          1] =   8
# Cycle:          57 Time:        1190    c[          2] =  10
# Cycle:          58 Time:        1210    c[          3] =  12
# Cycle:          61 Time:        1270    c[          0] =   6
# Cycle:          63 Time:        1310    c[          1] =   8
# Cycle:          65 Time:        1350    c[          2] =  10
# Cycle:          67 Time:        1390    c[          3] =  12
# At t=      1410000 clk=1 finish=1 return_val=        36
# Cycles:            68
```

Notice how the print statements are happening out-of-order? For instance `a[2]` is printing out before `c[0]`. To get more information about the iterative modulo schedule of the loop body open `pipelining.legup.rpt` and scroll to the bottom. You will see that the initiation interval (II) of the loop is 1. Each instruction is scheduled into a stage of the pipeline. To get a better look at the pipeline run `make w`. When asked “Are you sure you want to finish?” select No. Use the `ctrl-s` shortcut to search for a signal called `loop_1_pipeline_start`. Hit tab on this signal to get to scroll to when it’s asserted. Zoom out a bit and you will be able to see the `loop_1_valid_bit_*` signals for when each time step of the pipeline is valid.

Try commenting out the `loop_pipeline` tcl command in `config.tcl`. Run `make` and `make v`. Notice that the circuit gets slower, with latency in cycles increasing to 77. Also the print statements are now happening in order.

Look through the other benchmarks in `legup/examples/pipeline/` to get more examples of using loop pipelining. For more details see *Loop Pipelining*.

2.3 Parallel Flow

LegUp can also execute multiple accelerators in parallel. This is done using Pthreads and OpenMP. Each thread is compiled into an accelerator, and LegUp instantiates as many accelerators as the number of threads used in the C program. Using Pthreads, you can either execute the same function in parallel using multiple threads, or you can also execute different functions in parallel. OpenMP can be used to execute a loop in parallel.

LegUp currently supports the following Pthread and OpenMP functions/pragmas:

Pthread Functions	OpenMP Pragmas	OpenMP Functions
pthread_create	omp parallel	omp_get_num_threads
pthread_join	omp parallel for	omp_get_thread_num
pthread_exit	omp master	
pthread_mutex_lock	omp critical	
pthread_mutex_unlock	omp atomic	
pthread_barrier_init	reduction(operation: var)	
pthread_barrier_wait		

For working examples that use Pthreads and OpenMP, look in `legup/examples/parallel/`

The following make targets are relevant for the parallel flow:

- `make`: compile Pthreads applications to pure hardware
- `make parallel`: compile OpenMP and Pthreads+OpenMP applications to pure hardware
- `make v`: simulate parallel hardware with ModelSim
- `make w`: simulate parallel hardware with ModelSim, showing waveforms

2.4 Hardware/Software Hybrid Flow

LegUp can automatically compile one or more selected C functions into hardware accelerators while running the remaining program segments on the processor. Communication between the processor and hardware accelerators is performed over the Avalon Interconnection Fabric, which is automatically generated by Altera's QSys System Integration Tool.

The hybrid flow can target either a soft Tiger MIPS processor, or a hard ARM Cortex-A9 processor.

2.4.1 Hybrid Flow Overview

The steps of the hybrid flow are as follows:

1. The C source is compiled to LLVM IR
2. The LLVM IR is partitioned into a hardware section, and a software section
3. A wrapper function is generated to replace each accelerated function in the software section
4. The software section is compiled to either a MIPS or ARM executable

5. The hardware section is compiled to verilog, taking global variable addresses from the compiled software section
6. QSys is used to generate a system that includes the appropriate processor, the generated accelerator(s), and any additional hardware including caches, profilers, etc. QSys automatically generates any necessary interconnect.
7. The system can now be simulated (MIPS only), or synthesized and run on the board (ARM only)

If the function designated for acceleration has descendants (other functions which are called by the designated function), all of its descendants are also moved to hardware. Descendant functions which have been moved to hardware which are not called by other software functions are removed from the software section to reduce the program footprint. All remaining functions are compiled to a MIPS or ARM executable that can be run on the processor.

2.4.2 Wrapper Functions

LegUp generates a C wrapper function for every function to be accelerated.

For example, look at the example in `legup/examples/matrixmultiply`. There are three files in this directory:

- `matrixmultiply.c`: C source for the application
- `Makefile`: local makefile for the application
- `config.tcl`: local LegUp configuration for the application

Let's say we want to accelerate the multiply function, shown below:

```
int multiply(int i, int j)
{
        int k, sum = 0;
        for(k = 0; k < SIZE; k++)
        {
                sum += A1[i][k] * B1[k][j];
        }
        resultAB1[i][j] = sum;
        return sum;
}
```

To accelerate this function, put the function name in the `config.tcl` file as shown below:

```
set_accelerator_function "multiply"
```

Run `make hybrid`. LegUp will generate a C wrapper function, **legup_sequential_multiply**, to replace the **multiply** function. The wrapper function can be seen in the LLVM IR file `matrixmultiply.sw.ll`:

```
define internal fastcc i32 @legup_sequential_multiply(i32 %i, i32 %j) {
  volatile store i32 %i, i32* inttoptr (i32 -268435444 to i32*)
  volatile store i32 %j, i32* inttoptr (i32 -268435440 to i32*)
  volatile store i32 1, i32* inttoptr (i32 -268435448 to i32*)
  %1 = volatile load i32* inttoptr (i32 -268435456 to i32*)
  ret i32 %1
}
```

Equivalent C code for the wrapper is shown below:

```
// memory mapped addresses
#define add_DATA   (volatile int *)0xf00000000
#define add_STATUS (volatile int *)0xf00000008
#define add_ARG1   (volatile int *)0xf0000000C
#define add_ARG2   (volatile int *)0xf00000010
```

```c
int legup_sequential_multiply(int i, int j)
{
    // pass arguments to accelerator
    *add_ARG1 = i;
    *add_ARG2 = j;
    // give start signal
    *add_STATUS = 1;
    // get return data
    return = *add_DATA;
}
```

The wrapper function sends its arguments to the hardware accelerator then asserts the accelerator start signal, at which point the accelerator will stall the processor by asserting the Avalon waitrequest signal. When the accelerator finishes and sets waitrequest to 0, the processor resumes and retrieves the return value from the accelerator.

2.4.3 MIPS Hybrid Flow

The MIPS flow should work on any development board with a compatible FPGA. The DE1-SoC, SoCKit, DE4, and DE5 boards are supported.

Running the MIPS Hybrid Flow

First, ensure that a Tiger MIPS project has been selected in `legup/examples/legup.tcl`:

```
set_project CycloneV DE1-SoC Tiger_SDRAM
```

Next, ensure the local `config.tcl` contains the function to be accelerated:

```
set_accelerator_function "multiply"
```

The following make targets are relevant for the MIPS hybrid flow:

- `make hybrid`: generate the hybrid system
- `make hybridsim`: generate the hybrid system and simulate it in ModelSim
- `make simulation`: simulate the system in ModelSim (`make hybrid` must have been run previously)
- `make simulation_with_wave`: simulate the system in ModelSim with waveforms (`make hybrid` must have been run previously)
- `make hybrid_compile`: run a full Quartus compile on the hybrid system (`make hybrid` must have been run previously)

Memory Coherency

In order to keep memory coherent, all global variables which are not constants are stored in main memory, which is shared between the processor and accelerators. When a hardware accelerator tries to access global variables it first checks the on-chip data cache, which is also shared between the processor and all accelerators. If there is a cache hit, the data is retrieved from the cache. If there is a cache miss, the off-chip main memory is accessed, which takes many more cycles to return the data. All constant variables in the hardware accelerator are stored in local block RAMs, since they will never be modified and thus it does not make sense to store them in high latency off-chip memory. All hardware accelerator local variables are also stored in local block RAMs.

2.4.4 ARM Hybrid Flow

Thy hybrid flow can target either a Tiger MIPS processor, or an ARM Cortex-A9. To test the ARM flow, it is necessary to have a board with a CycloneV SoC. The DE1-SoC and SoCKit boards are supported.

Running the ARM Hybrid Flow

First, ensure that a ARM project has been selected in `legup/examples/legup.tcl`:

```
set_project CycloneV DE1-SoC ARM_Simple_Hybrid_System
```

Next, ensure the local `config.tcl` contains the function to be accelerated:

```
set_accelerator_function "multiply"
```

The following make targets are relevant for the ARM hybrid flow:

- `make hybrid`: generate the hybrid system
- `make hybrid_compile`: run a full Quartus compile on the hybrid system (`make hybrid` must have been run previously)
- `make program_board`: program the board with the generated .sof file (`make hybrid_compile` must have been run previously)
- `make run_on_board`: to connect to the processor, and download and run the program (`make program_board` must have been run previously)

Altera does not provide a simulation model for the ARM core; therefore, it is not possible to simulate an ARM hybrid system.

Note: Any accelerator generated with the ARM hybrid flow should be the same as one generated with the MIPS hybrid flow, with the exception of global variable addresses. Hence, if an accelerator simulates properly in the MIPS flow, it should also work properly in the ARM flow.

Memory Coherency

Unlike the MIPS system, it is not possible to connect the accelerator(s) directly to the processor cache. However, cache coherency can still be maintained.

In the ARM system, the processor caches are part of the hard processor system, or HPS. The following figure shows the architecture of the Cyclone V Soc device found on the DE1-SoC and SoCKit boards:

The top of the figure shows the FPGA side, and the bottom shows the hard processor system, or HPS, side. The FPGA bridge facilitates communication between the FPGA and the HPS. The L3 interconnect connects the FPGA bridge, SDRAM controller, microprocessor unit, L2 cache, and peripherals (not shown). The microprocessor unit contains a Cortex-A9 MPCore processor with two CPUs. Each CPU has separate 32 KB instruction and data caches. There is a shared 512 KB L2 cache. The microprocessor unit also contains a snoop control unit, or SCU, and the accelerator coherency port, or ACP. The SCU maintains cache coherency between the two CPUs. The ACP allows masters on the L3 interconnect to perform memory accesses that are cache coherent with the microprocessor unit.

Memory accesses made to the ACP get routed through the SCU. The SCU subsequently routes the request to the L1 and L2 cache. If the request misses in both caches, it is routed to the SDRAM controller. In addition to coherency, ACP accesses have the added benefit, (in the case of a cache hit), of being faster than going directly to the SDRAM controller. In the case of a miss, access times are the same as going directly to the SDRAM controller through the L3 interconnect.

The ACP can be accessed through the FPGA-to-HPS bridge. The 4 GB of address space is divided as follows:

- `0x00000000` to `0x7FFFFFFF` maps to the SDRAM controller
- `0x80000000` to `0xBFFFFFFF` maps to the ACP

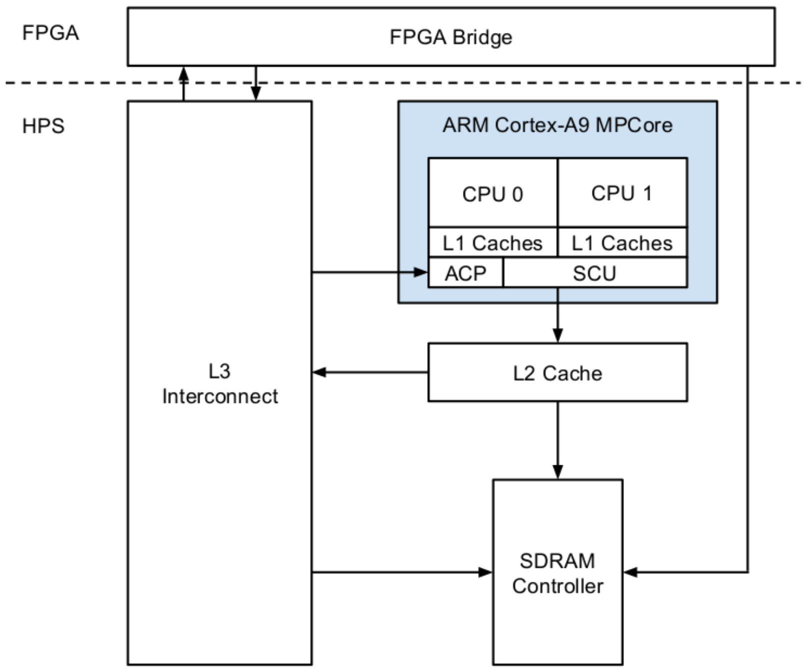


Figure 2.1: Cyclone V SoC Architecture

- `0xC0000000` to `0xFFFFFFFF` maps to the FPGA slaves and peripherals

LegUp forces all global memory accesses from the accelerator to go through the ACP, ensuring cache coherency.

2.4.5 Hybrid Parallel Flow

It is also possible to use the parallel and hybrid flows together. `make hybrid` can be used to compile Pthreads applications `make hybridparallel` can be used to compile OpenMP and Pthreads+OpenMP applications

This parallel hybrid flow has all the same properties as the sequential hybrid flow, except for a few minor differences. Instead of stalling the processor after calling an accelerator, the processor continues to call all of the accelerators that execute in parallel and then polls on the accelerators to check if they are done. The return value is retrieved after polling. Hence LegUp generates a pair of C wrappers for each parallel function, which is called a calling wrapper and a polling wrapper. The calling wrapper sends all of the arguments to the accelerator and asserts the start signal, and the polling wrapper polls on the accelerator to check if it is done, then retrieves the return value if necessary. For example, let's say we want to parallelized the following add function:

```
int add (int * a, int * b, int N)
{
    int sum=0;
    for (int i=0; i<N; i++)
        sum += a[i]+b[i];
    return sum;
}
```

The add function needs to be re-written so that it can be used with Pthreads, as shown below:

```
void *add (void *threadarg)
{
    int sum=0;
    struct thread_data* arg = (struct thread_data*) threadarg;
    int *a = arg->a;
    int *b = arg->b;
    int N = arg->N;
    for (int i=0; i<N; i++)
    {
        sum += a[i]+b[i];
    }
    pthread_exit((void*)sum);
}
```

This add function can execute in parallel using two threads with the following code:

```
pthread_create(&threads[0], NULL, add, (void *)&data[0]);
pthread_create(&threads[1], NULL, add, (void *)&data[1]);

pthread_join(threads[0], (void**)&result[0]);
pthread_join(threads[1], (void**)&result[1]);
```

LegUp automatically replaces the call to pthread_create with calls to LegUp calling wrappers and replaces the call to pthread_join with calls to LegUp polling wrappers. With two threads executing the add function, the following shows the C-code equivalent of the wrapper functions that are generated:

```
#define add0_DATA   (volatile int * ) 0xf0000000
#define add0_STATUS (volatile int * ) 0xf0000008
#define add0_ARG1   (volatile int * ) 0xf000000c
#define add0_ARG2   (volatile int * ) 0xf0000010

void legup_call_add0(char *threadarg)
{
    *add0_ARG1 = (volatile int) threadarg;
    *add0_ARG2 = (volatile int) 1;
    *add0_STATUS = 1;
}

#define add1_DATA   (volatile int * ) 0xf0000020
#define add1_STATUS (volatile int * ) 0xf0000028
#define add1_ARG1   (volatile int * ) 0xf000002c
#define add1_ARG2   (volatile int * ) 0xf0000030

void legup_call_add1(char *threadarg)
{
    *add1_ARG1 = (volatile int) threadarg;
    *add1_ARG2 = (volatile int) 2;
    *add1_STATUS = 1;
}

char *legup_poll_add0()
{
    while (*add0_STATUS == 0){}
    return (char*)*add0_DATA;
}

char *legup_poll_add1()
{
    while (*add1_STATUS == 0){}
    return (char*)*add1_DATA;
}
```

legup_call indicates a calling wrapper, and legup_poll indicates a polling wrapper. The first argument pointer, ARG1, passes in the function argument, threadarg, and the second argument pointer, ARG2, passes in the threadID. This threadID is determined at compiled time by looking at the number of threads which are accelerated.

2.4.6 Hybrid Flow Limitations

LegUp's hybrid flow does not work with the following features:

- local memories
- shared local memories

2.5 Pure Software Flow

LegUp also has a pure software flow that can be used for testing your C code. The pure software flow works for both MIPS and ARM processors. To target a specific processor architecture, make sure an appropriate project has been selected in `legup/examples/legup.tcl`.

The following make targets are relevant to the pure software flow:

- `make sw`: generate an ELF file for the desired processor architecture
- `make swsim`: compile the application and simulate it with ModelSim (MIPS only)
- `make simulation`: simulate execution of the application on the processor using ModelSim (`make sw` must have been run previously)(MIPS only)
- `make simulation_with_wave`: simulate execution of the application on the processor using ModelSim, with waveforms (`make sw` must have been run previously)(MIPS only)
- `make run_on_board`: run the application on the board (`make sw` must have been run previously)(ARM only)
- `make emul`: compile and run the application in an emulator: gxemul for MIPS and QEMU for ARM

2.6 Custom Verilog & Propagating I/O

You can tell LegUp to instantiate particular C functions in the generated verilog but leave the hardware implementation of those function up to you. LegUp will not attempt to compile the code inside of any functions you mark as custom verilog, so you can include code that LegUp cannot normally compile.

Every time you run LegUp it overwrites the generated verilog files, so you need to write your custom verilog modules in separate files and tell LegUp to include those files. To tell legup to include a file add the following command to the TCL configuration file:

```
set_custom_verilog_file "file_name"
```

To mark functions as custom verilog you add commands to the TCL configuration file for the project. The syntax for this command is as follows:

```
set_custom_verilog_function "function_name" <memory or noMemory> \
    <input or output> high_bit:low_bit signal_name
```

The `function_name` is the name of your function as it appears in your C code. The next token can be either `memory` or `noMemory` to specify whether or not your custom verilog requires access to the LegUp memory signals. Following the memory token are sets of tokens that describe input and output signals. The signals specified here will propagate up the call tree and exist in the top level module. You can specify as many inputs and outputs as you want, but every input or output requires three tokens to describe it: * `input` or `output` specifies whether the signal should be a verilog input or output * `high_bit:low_bit` specifies the bits to which your signal should connect. `high_bit` and `low_bit` must be integers greater than zero. * `signal_name` specifies the name of your signal

A complete example:

```
set_custom_verilog_function "assignSwitchesToLEDs" noMemory \
                                                   output 5:0 LEDR \
                                                   input 5:0 SW \
                                                   input 3:0 KEY
```

In addition to specifying your function as custom verilog your Verilog and C code must meet some specifications that ensure that your code integrates into the LegUp generated code. These specifications are described in Specifications for Custom Verilog C Code and Specifications for Custom Verilog Modules.

2.6.1 Specifications for Custom Verilog C Code

LegUp uses compiler optimizations to improve the performance of your code. As a result, some functions will be inlined or removed. For custom verilog functions inlining is equivalent to removing, so you must tell the compiler not to remove or inline your custom verilog functions. To do this, add the `noinline` and `used` C attributes to your function definitions. An example of this is provided below:

```
void __attribute__((noinline)) __attribute__((used)) exampleFunction() {..}
```

Additionally, you should add a volatile memory call to your function if the C implementation does not call any functions. To do this, you can add the following code snippet to the body of your function:

```
volatile int i = 0;
```

2.6.2 Specifications for Custom Verilog Modules

LegUp converts custom verilog function calls from C to module instantiations in Verilog. The module instantiations that LegUp generates define the naming scheme for all input and output signals in your custom verilog. As a result, the modules you write must have the following signals:

- `input clk`: The standard clock for the circuit
- `input clk2x`: A clock running twice as fast as the standard clock
- `input clk1x_follower`: The standard clock with a phase shift of 180 degrees
- `input reset`: The reset signal for the circuit
- `input start`: A 1 cycle pulse signifying that the function should start executing
- `output finish`: Set to 1 to tell the LegUp state machine that it can move on to the next state (if you never set this to 1 in your custom verilog your program will stop executing after it starts your function)
- `return_val`: Only necessary if your C function is non-void. Has the bit width of the return type specified in the C file. Value should be held until the start signal is asserted.

Additionally, your custom modules must have signals for any propagating I/O that you specify, any arguments and return values that your C function has, and all of the memory controller signals if you specify in the config file that your function requires access to memory. The propagating I/O signals can be specified as they were declared in the config file. The arguments to your function have the bit width of their C type (e.g. `int` is `[31:0]`) and the same name with the prefix `arg_`. So an argument declared in C as `int counter` would be declared in your verilog module as `input [31:0] arg_counter`. The return value is called `return_val` and is also the same size as the C type. For information on the memory controller signals, please see the Memory Controller section of Hardware Architecture.

2.6.3 Custom Top Level Modules

LegUp allows you to specify a custom top level module. The top level module you specify will be set to the top level module of any projects made with the `make p` command. To specify a custom top level module add the following command to the config file for your project:

```
set_custom_top_level_module "topLevelModuleName"
```

For convenience, you can put your custom top level module in any of the files you include with the `set_custom_verilog_file` command.

2.6.4 Custom Test Benches

LegUp allows you to specify a custom test bench module. The module you specify will be used whenever you run the `make v` or `make w` commands. To specify a test bench module add the following command to the TCL configuration file:

```
set_custom_test_bench_module "testBenchModuleName"
```

Make sure that your custom test bench modules are in verilog files included with the `set_custom_verilog_file` command.

2.6.5 Known Limitations of Custom Verilog

At the present time propagating I/O is not supported for the hybrid flow or pthreads.

You cannot call a custom verilog function inside of a loop that you have marked for loop pipelining.

There is currently no support for inout style arguments (pass by reference) other than storing a value in memory and passing in a pointer to that value.

Propagating signals can only be `input` or `output`.

Bit width specifiers for propagating I/O can only be integers.

CHAPTER

THREE

HARDWARE ARCHITECTURE

This section will explain the architecture of the hardware produced by LegUp when synthesizing C into Verilog.

3.1 Modules

Each C function corresponds to a Verilog module. For instance, the following C prototype:

```
int function(int a, int* b);
```

Would generate a module with the following interface:

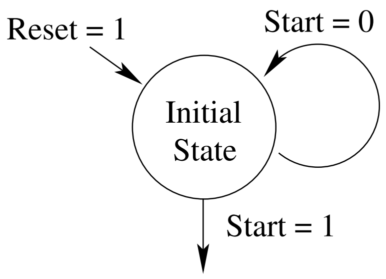
```
module function
    input clk;
    input reset;
    input start;
    output reg finish;

    output reg [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] memory_controller_address;
    output reg  memory_controller_enable;
    output reg  memory_controller_write_enable;
    input memory_controller_waitrequest;
    output reg [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_in;
    input [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_out;

    input [31:0] a;
    input [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] b;

    output reg [31:0] return_val;
endmodule
```

The start/reset signals are used by the first state of the state machine:

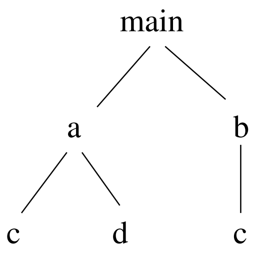


The finish signal is kept low until the last state of the state machine, where finish is set to 1 when waitrequest is low. Memory signals to the memory controller are shown below. These memory ports are only created if the memory controller exists in the circuit. This is described in more detail later in this document.

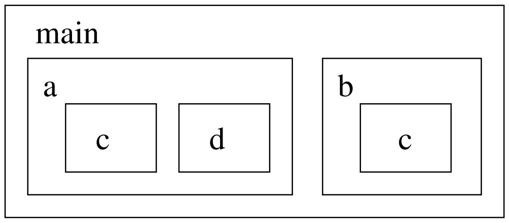
Memory signal	Description
memory_controller_address	32-bit address of memory
memory_controller_enable	enable reading/writing this cycle
memory_controller_write_enable	if 1 then write, else read
memory_controller_waitrequest	if this is 1 then hold the state machine constant
memory_controller_in	data being read from memory
memory_controller_out	data to write into memory

Function parameters are provided by ports a (integer), and b (pointer). The return_val port passes back the function return value.

The module instantiation hierarchy is dependent on the call graph of the C code. For instance, with the function call graph shown by:



Module instantiation hierarchy is shown by:



Note that module **c** is instantiated twice.

3.2 Memory architecture

LLVM has three types of memory: stack, globals, and the heap. We can ignore the heap because Legup does not support dynamically allocated memory. The stack is used for local variables but there is no equivalent of a stack in hardware.

In Legup, there exists 4 hierachies of memories: 1) Local memory, 2) global memory, 3) cache memory, and 4) off-chip memory. 1 and 2 are implemented inside a hardware accelerator for data local to the accelerator, 3 and 4 are shared between all hardware accelerators and the processor. In the pure hardware case, where the processor doesn't exist in the system, only the first two levels of memories exist.

LegUp uses points-to analysis to determine which arrays are used by which functions. If an array is determined to be used by a single function, that array is implemented as a local array, which is created and connected directly inside the module that accesses it. If an array is accessed by multiple functions or if the points-to analysis cannot determine the exact array for a pointer, the array is implemented as a global array, which is created inside the memory controller. The memory controller, which is described below, allows memory accesses to be steered to the correct array at runtime.

By default, each local/global memories are stored in a separate block RAM. This allows all local memories to be accessed in parallel. For global memories inside the memory controller, they are limited to 2 memory accesses per cycle, due to RAMs being dual-ported memories. However, storing them in separate RAMs can help for debugging, as it is easier for the hardware designer to debug if he/she sees individual arrays from his C program in separate rams rather than buried in a large ram. There is an option to merge memories in the memory controller to single RAMs, in order to reduce the RAM usage. This can be done by turning on the GROUP_RAMS tcl parameter.

3.3 Local Memories

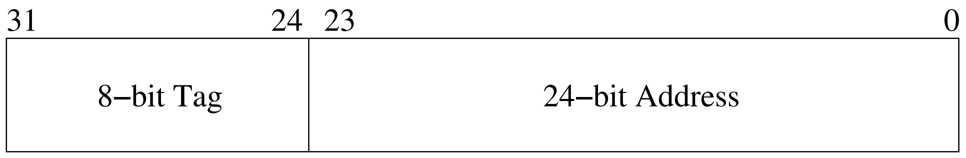
Any memory that is used by a single function is designated as a local memory, and is created inside the module being used. There can be many local memories inside a module and they can all be accessed in parallel, since they are implemented in separate block RAMs. Since local memories are connected directly in a module, they do not require expensive multiplexing, as in the case of global memories. This helps to reduce area and improve Fmax. Also they do not connect through the memory controller ports shown earlier but connect directly through wires inside the module.

All local memories are implemented in dual-ported RAMs and have a latency of 1 clock cycle.

3.4 Global Memories

Any memories which are used by multiple functions, or if the points-to analysis fails to determine the exact array for a pointer, that array is designated as a global array, and is created inside the memory controller. Each global memory is identified by a unique number called a tag. We first describe below the address format for global memories below.

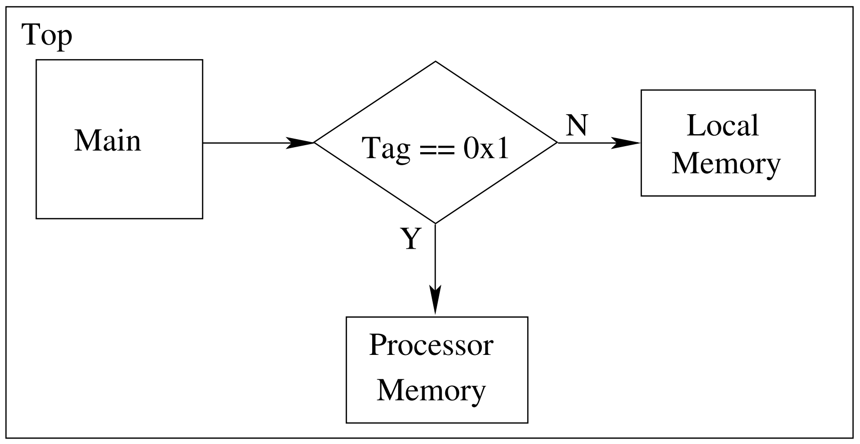
32-bit Address Format:



The upper 8 bits of memory addresses are reserved for tag bits, allowing 255 memory locations. The tag bits are used at circuit runtime to steer each memory access into the memory controller to the correct RAM, or to the processor memory. Tag 0x0 is reserved for null pointers. Tag 0x1 is reserved for processor memory. The 24 bit address allows a 16MB byte-addressable address space. Because the lower bits are used for the pointer address, this scheme allows pointer arithmetic, incrementing the address won't affect the tag bits.

For instance:

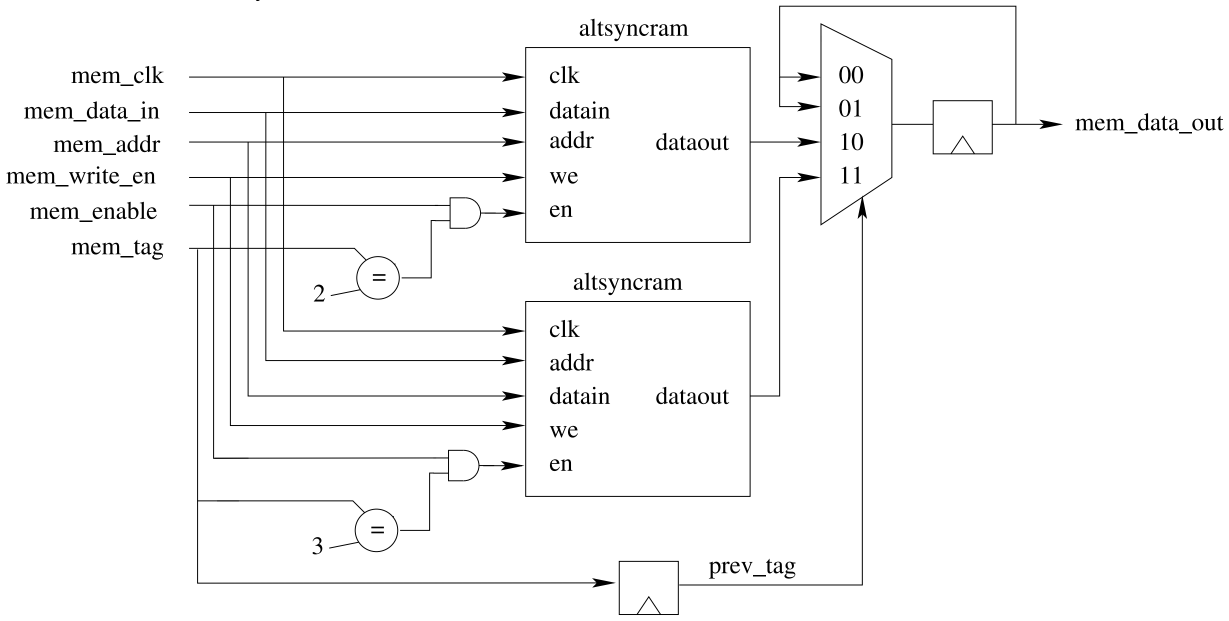
Inside the top level module the tag bits are used to steer the memory accesses to either the memory controller or the processor memory. The following figure shows memory accesses from top-level module:



All global memories are implemented in dual-ported memoires and have a 2 cycle latency.

3.5 Memory Controller

We describe the memory controller architecture below.



The memory controller is a ram composed of smaller rams. We need a memory controller to share memory between modules and to handle pointer aliasing within the same module. We need the tag bits because at compile time you may not be able to calculate exactly which pointers point to the memory and that no other pointer ever points to that piece of memory. So the memory controller is a central place to handle aliasing. The memory controller is only created if there are memories shared between functions, or if the points-to analysis fails to determine the exact memory for all pointers in the program. In the figure, mem_data_out width is the max data width of all RAMs in the memory controller. The size of pointers is currently fixed at 32 bits.

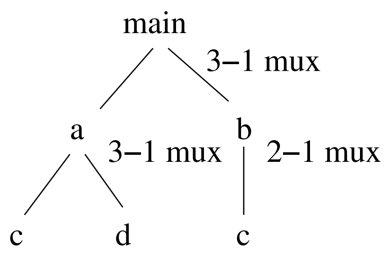
The latency of reading from a RAM is one cycle, so we must use the previous tag to determine which RAM is outputting the data requested in the previous cycle. We registered the output of the memory controller to improve Fmax as the steering mux can become large. Note that for tags 0 and 1, mem_dat_out keeps its old value.

The mem_waitrequest signal is not shown here and for the pure hardware case it is always given a 0 value. If mem_waitrequest equals 1 then the memory controller is indicating it will take longer to retrieve the memory. As long as mem_waitrequest is high the memory is not ready. After mem_waitrequest goes low then the data will be available on the next cycle. This is important for the processor memory which can take many cycles if there is a cache miss. In every state machine that legup generates the state will not change if mem_waitrequest is high.

LegUp also handles structs. In a struct, the individual elements can have non-uniform size. Also structs must be byte addressable. To handle this we need an additional 2-bit input **mem_size** which indicates the size of the struct element we are accessing. mem_size is 0 for byte, 1 for short, 2 for integer, 3 for long. For each struct a 64-bit wide ram is instantiated. Using the mem_addr and mem_size we can use the byte enable of the ram to only write the correct section of the ram. When reading data, we must steer the correct bits of the 64-bit word to the lowermost bits of mem_data_out.

If an array is initialized in the C code, we create a MIF, memory initialization file, for that RAM.

Currently, only one memory controller module may be created for a program. Each module must communicate through its parent module to get to the memory controller. Hence, there are muxes at each level of the hierarchy as shown in the figure:



For instance, in the main module we are either in the body of the main function, in 'a', or in 'b', so we need a 3-1 mux. Since we do not allow recursion, the call graph will always be a tree. Note that the further down on the call graph there is more delay to the memory controller.

3.6 Function Calls

Every function call requires two states. An initial state to set start=1 for the called function, then a second state that loops until receiving a finish=1 from the called function. Function calls are not allowed in the same state as a memory load/store.

3.7 Signed/Unsigned

In LLVM, all integers are assumed to be unsigned unless passed to a signed instruction (sdiv, srem). Since integers are unsigned, before being passed to an add operation they must be appropriately sign or zero extended. To deal with sign extension LLVM has two instructions: sign extend (sext) and zero extend (zext), which both result in an unsigned integer. However, Verilog operations such as +/- depend on the type of the operands, which can be 'signed' or 'unsigned'.

In LegUp, we declare every Verilog variable as unsigned and use the $signed() Verilog command when required by an instruction such as sdiv, srem, or sext.

3.8 Mult-dimensional Arrays

Multi-dimensional arrays are stored in row-major order, the same convention used by C. For instance given an array:

```
int array[2][2][2] = {{{0, 1}, {2, 3}}, {{4, 5}, {7, 8}}}
```

If we assign variables for the size of each dimension of the array[A][B][C] where A=2, B=2, C=2. Then to access the element array[a][b][c] the memory offset is given by:

```
offset = c + C*b + C*B*a = c + C*(b + B*a)
```

This supports storing an array of arbitrary dimension in a ram the same width as an element with A*B*C rows.

3.9 Functional Units

To keep Fmax high, we pipelined dividers/remainders and multipliers. The pipeline depth of Dividers/Remainders are equal the bit width of the operation. Multipliers have a pipeline depth of 2.

We only share dividers/modulus functional units to save area. The divider clock enable is set to 0 when the memory controller's wait_request signal is high or when we're calling a function.

3.10 Structs

Structs are supported by LegUp including pointers, arrays, structs and primitives as elements. Pointers to structs are also supported, for example linked lists can be synthesized.

LLVM's TargetData is used to specify alignment for structs. For instance for a 32-bit machine, pointers are 32-bits and 32-bit aligned. LLVM integers of type i64 are 64-bit aligned. Structs are 64-bit aligned.

3.11 Avalon Signals

Each hardware accelerator contains the following Avalon signals.

Avalon signal	Description
csi_clockreset_clk	hardware accelerator clock
csi_clockreset_reset	hardware accelerator reset

Avalon slave signals (prefixed with **avs_s1**) are used by the processor to communicate with the hardware accelerator

Avalon signal	Description
avs_s1_address	address sent from processor to hardware accelerator. Determines which accelerator argument is being written or whether the processor is giving the start signal
avs_s1_read	processor sets high to read return value from hardware accelerator
avs_s1_write	processor sets high to write an argument or start the processor.
avs_s1_readdata	accelerator sets this to the return data to send back to the processor
avs_s1_writedata	processor sets this to the value of the argument being written to the accelerator

Avalon master signals (prefixed with **avm**) which talk to the on-chip data cache. These signals correspond to the memory-mapped address of the data cache.

Avalon signal	Description
avm_ACCEL_address	points to the memory-mapped address of the data cache
avm_ACCEL_read	set high when accelerator is reading from memory
avm_ACCEL_write	set high when accelerator is writing to memory
avm_ACCEL_readdata	data returned from memory when accelerator issues a read
avm_ACCEL_writedata	**on a write, it sends the data to be written to memory, as well as the memory address and t** on a read, it sends the memory address and the size of the data (8bit, 16bit, 32bit, 64bit)
avm_ACCEL_waitrequest	asserted until the read data is received

The on-chip data cache is a write-through cache, hence when an accelerator or the processor writes to the cache, the cache controller also sends the data to the off-chip main memory.

If a memory read results in a cache miss, the cache controller will access off-chip main memory to get the data, which will be written to the cache and also returned to the accelerator.

For parallel execution which uses either a mutex or a barrier, the following Avalon signals are also created. This Avalon master is used to communicate with the mutex to either lock (pthread_mutex_lock) or unlock (pthread_mutex_unlock), and also used communicate with the barrier to initialize (pthread_barrier_init) and poll on the barrier (pthread_barrier_wait) until all threads have reached the barrier.

Avalon signal	Description
avm_API_address	points to the memory-mapped address of mutex, barrier
avm_API_read	set high when accelerator is reading from mutex, barrier
avm_API_write	set high when accelerator is writing to mutex, barrier
avm_API_readdata	data returned from mutex, barrier when accelerator issues a read
avm_API_writedata	data written to mutex, barrier when accelerator issues a read
avm_API_waitrequest	asserted until the read data is received

CHAPTER

FOUR

PROGRAMMER'S MANUAL

This is a programmer's manual for the LegUp high-level synthesis framework. The intent is to give a reader a top-level view of how LegUp is implemented, and where key pieces of the functionality reside in the codebase. LegUp is a target back-end pass to the LLVM compiler infrastructure. If you haven't used LLVM before please familiarize yourself with the LLVM Documentation. This manual assumes that you understand basic LLVM concepts. First we discuss the LegUp compiler backend pass, which receives the final optimized LLVM intermediate representation (IR) as input and produces Verilog as output. Next we discuss the LegUp frontend compiler passes, which receive LLVM IR as input and produce modified LLVM IR as output.

If you just want to dive in. Start by looking at `runOnModule()` in llvm/lib/Target/Verilog/LegupPass.cpp

4.1 LLVM Backend Pass

Most of the LegUp code is implemented as a target backend pass in the LLVM compiler framework. The top-level class is called *LegupPass*. This class gets run by the LLVM pass manager, which calls the method *runOnModule()* passing in the LLVM IR for the entire program and expecting the final Verilog code as output.

The LegUp code is logically structured according to the flow chart:

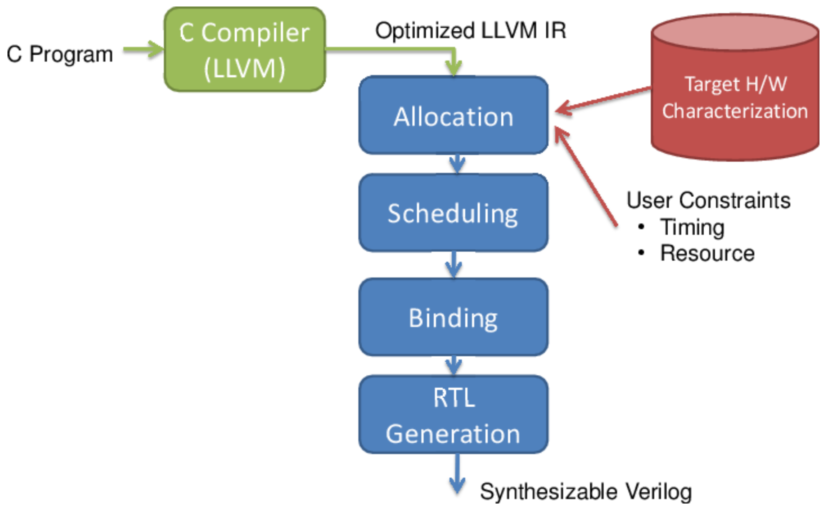


Figure 4.1: LegUp flow

There are five major logical steps performed in order: Allocation, Scheduling, Binding, RTL generation, and producing Verilog output. First, we have an *Allocation* class that reads in a user Tcl configuration script that specifies the target device, timing constraints, and HLS options. The class reads another Tcl script that contains the FPGA device specific operation delay and area characteristics. These Tcl configuration settings are stored in a global *LegupConfig* object accessable throughout the code. We pass the Allocation object to all the later stages of LegUp. This object also handles mapping LLVM instructions to unique signal names in Verilog and for ensuring these names do not overlap with reserved Verilog keywords. Global settings that should be readable from all other stages of LegUp should be stored in the Allocation class.

The next step loops over each function in the program and performs HLS scheduling. The default scheduler uses the SDC approach and is implemented in the *SDCScheduler* class. The scheduler uses the *SchedulerDAG* class, which holds all the dependencies between instructions for a function. The final function schedule is stored in a *FiniteStateMachine* object that specifies the start and end state of each LLVM instruction.

Next, we perform binding in the *BipartiteWeightedMatchingBinding* class, which performs bipartite weighted matching. We store the binding results in a datastructure that maps each LLVM instruction to the name of the hardware functional unit that the instruction should be implemented on.

In the next step, the *GenerateRTL* class loops over every LLVM instruction in the program and using the schedule and binding information, creates an *RTLModule* object (RTL Datastructure) that represents the final hardware circuit.

Finally, the *VerilogWriter* class loops over each *RTLModule* object and prints out the corresponding Verilog for the hardware module. We also print out any hard-coded testbenches and top-level modules.

4.2 LLVM Frontend Passes

In this section, we discuss portions of the LegUp code that are implemented as frontend LLVM passes. These passes receive LLVM IR as input and return modified LLVM IR as output and are run individually using the LLVM **opt** command. In the class implementing each pass, the LLVM pass manager will call the method *runOnFunction()* and provide the LLVM IR for the function and will expect the modified LLVM IR as output.

For the hybrid flow, we remove all functions from the IR that should be implemented in software in the *HwOnly* class. We remove all functions that should be implemented in hardware with the *SwOnly* class. We run these two passes on the original IR of the program to generate two new versions of the IR. We pass the HwOnly IR to the LegUp HLS backend and the SwOnly IR to the MIPS/ARM compiler backend.

Loop pipelining is performed by the *SDCModuloScheduler* class. This pass will determine the pipeline initiation interval and the scheduled start and end time of each instruction in the pipeline. This data is stored in LLVM IR metadata which can be read later by the LegUp backend.

If-conversion is performed by the class *LegUpCombineBB*, which removes simple control flow and combines basic blocks. In the class *PreLTO*, we detect LLVM built-in functions that can exist in the IR (i.e. memset, memcpy). We replace these functions with equivalent LLVM IR instructions that we can synthesize in the LegUp backend.

4.3 Execution Flow

The overall flow of execution in LegUp is as follows. First, the LLVM front end `clang-3.5` takes the .c files and compiles them into LLVM intermediate representation, stored in a byte code file (.bc). This byte code may contain LLVM intrinsic functions, which are functions that LLVM assumes exist in the C library: memcpy, memset, and memmove. These functions do not exist in hardware, so we replace them with functions that we've hand written. This is done with `opt -legup-prelto` and then linking in our versions with llvm-link lib/liblegup.bc. Now we are left solely with LLVM IR and no intrinsics. We pass this code to llvm-ld to perform link time optimizations. Finally, we pass the optimized bytecode to `llc -march=v` to run the Verilog backend (LegUp HLS). You can find such calls in the main LegUp Makefile: `examples/Makefile.common`.

The flow of the Verilog backend is as follows: First, the LLVM pass manager calls `LegupPass:runOnModule()`, which is a top-level governing function of the LegUp HLS flow. In that method, you will see many references to LEGUP_CONFIG, which is an object that holds the constraints on the HLS. For instance, if you wanted to know the number of DSPs available on this device use LEGUP_CONFIG->getMaxDPSs(). In the `runOnModule` function, you will also see references to the Allocation object, which contains global information about the circuit, such as its RTL modules, list of RAMs, and the GlobalNames object. The GlobalNames class is used to make sure each LLVM instruction has a unique name that doesn't overlap with a reserved Verilog keyword (reg, wire, etc.). Reading further in `LegupPass::runOnModule()`, you will see the method creating an RTL generator (`GenerateRTL`) instance for each function in the LLVM module being synthesized, and further down you will see the lines that invoke the HLS scheduler for each function:

```
GenerateRTL *HW = *i;
HW->scheduleOperations();
```

The purpose of scheduling is to schedule LLVM instructions into clock cycles. There are some helper classes to aid in this task: The data flow graph of LLVM instructions is represented in the `SchedulerDAG` class. Given an instruction, you can get the successors and predecessor instructions. For instance:

```
%1 = add %2, %3
%4 = add %1, %5
```

The predecessors of %1 are its operands: instructions %2, and %3. %1's successor is the instruction %4. There are also dependencies between load, store, and call instructions that can access memory. These dependencies can be detected using alias analysis performed by LLVM. Scheduling works on a function-by-function basis. There are two important parameters for each instruction:

1. The latency, how many clock cycles you must wait until the output is available, loads have latency of 2 (by default, but controllable through a tcl parameter).
2. The delay of an instruction is the other parameter, retrieved from LEGUP_CONFIG by accessing the `Operation` object for a given type of instruction (e.g. a 32-bit signed addition). Such delays are stored in a tcl characterization file for each supported device. These are found in the `boards` subdirectory of the LegUp distribution. For example, see `boards/CycloneV/CycloneV.tcl`.

The delay approximations allow the algorithm to determine how many instructions can be *chained* together in the same cycle. During scheduling, each instruction is assigned a state object that represents a state in a finite state machine stored in the FSM object. Branches, jump, and switch instructions are used to determine next state variable assignments. Each state has 3 possibilities analogous to the LLVM branch, jump, and switch instructions. First, a defaultTransition can be specified. Or a single transition variable can be set, then one or more transition conditions can be specified. If the transition variable is equal to the condition, then the associated state is the next state. In essence, the transitions are the edges in a state diagram. After we have scheduled each function, we can call generateRTL to create the RTLModule object representing the final hardware circuit for the scheduled function.

An RTLModule has a list of inputs, outputs, parameters, and RTLSignals. To understand RTLSignal, it's useful to look at the structure of the Verilog code. Each RTLSignal represents a wire or register that can be driven by other RTLSignals under different conditions. Each condition is listed as an if statement in the **always@** block devoted to that signal. The most common condition is if we are in a particular state. It is so common that there is a function to simplify this process:

```
connectSignalToDriverInState(signal, driverSignal, state, instruction, ...)
```

Here we say that during **state** we want **driverSignal** to drive **signal**. The optional **instruction** argument adds a comment above this Verilog assignment indicating the instruction that **driverSignal** was dervied from. Another option is to unconditionally drive a signal. In order to do so, use:

```
signal->connect(driver, instruction)
```

Note that this will clear away prior conditional drivers. To manually specify a conditional driver use:

```
signal->addCondition(conditionSignal, driverSignal)
```

If the **conditionSignal** is 1 then **driverSignal** drives **signal**.

To create a register or wire RTLSignal use these functions:

```
rtl->addReg(...)
rtl->addWire(...)
```

Where **rtl** is an RTLModule object, **rtl** must keep track of all signals used in order to print the variable declarations. To create a signal you must specify a name. Normally the `verilogName(instruction)` function is used, which creates a unique name for the instruction using the GlobalNames object in the allocation object discussed above. For all instructions, we follow the convention that there are 2 signals created, one wire, to represent the instruction during the state it is assigned, and one register, which the wire feeds in the assigned state only. The register is used if the instruction is used in another state. The name of the wire is `verilogName(instruction)`, the name of the register is `verilogName(instruction) + "_reg"`.

4.4 Test Suite

The test suite is built using DejaGNU (also used by GCC and LLVM). The DejaGNU test framework is launched by the `runtest` command in the `examples` subdirectory, which recursively searches all the directories in the current working directory for `dg.exp` tcl files. Every directory in `examples` that is part of the test suite has a `dg.exp` tcl file, for instance `examples/array/dg.exp`. These tcl files all load the library `examples/lib/legup.exp` and call functions like **run-test** or **run-test-gx** to run various tests.

To run the default test suite use the command:

```
cd examples
runtest
```

You should see the following output after a few minutes:

```
            ===  Summary ===

# of expected passes                476
```

Note: The number of passes you observe may differ, depending on the LegUp version your are using. The default test suite essentially takes every example and runs:

```
make
make v
```

The first command generates the Verilog for the testcase. The second command simulates that Verilog with ModelSim. The output is then parsed to ensure the *return_val* is correct and there are no Modelsim warnings or errors.

You should run the LegUp test suite regularly during development to ensure your hardware is correct. We have found that it is much easier to track down bugs this way than debugging the RTL simulations. In fact, we run our regression tests after every commit using buildbot.

Other useful variants of the `runtest` DejaGNU command are:

```
# for verbose output:
runtest -v
# only run the mips test:
runtest chstone/mips/dg.exp
```

4.5 LLVM Passes

LLVM is structured as a series of compiler passes that run in sequence on the underlying intermediate representation. The main LegUp pass is a target backend called LegupPass. Passes are normally classes inheriting from FunctionPass, which have an entry function called:

```
bool runOnFunction(Function &F);
```

When runOnFunction() is called, LLVM has already constructed the intermediate representation (IR) for the input C file. By traversing over the IR we perform the steps to generate valid Verilog RTL code. LegupPass inherits from ModulePass, which has an entry function called:

```
bool runOnModule(Module &M);
```

In LLVM, a Module has a list of Functions. A Function has a list of BasicBlocks. A BasicBlock has a list of instructions. The definition of a basic block is a straightline sequence of code with a single entry point (at the beginning) and a single exit point (at the end).

4.6 Source Files

LegUp files inside the LLVM source tree:

- **The core of LegUp is in:**
  - `llvm/lib/Target/Verilog/`
- **Other LegUp passes that are run with opt:**
  - `llvm/lib/Transforms/LegUp/`
- **llc calls the LegupPass and has been slightly modified:**
  - `llvm/tools/llc/llc.cpp`
- **Other files with minor changes:**
  - `llvm/tools/opt/opt.cpp` (can use Tcl)
  - `llvm/autoconf/configure.ac` (add Verilog target)
  - `llvm/configure` (add Verilog target)

4.7 Important Classes

4.7.1 RTL Datastructure

The data structure that we use to represent an arbitrary circuit uses the following classes:

- `RTLModule` - a hardware module.
- `RTLSignal` - a register or wire signal in the circuit. The signal can be driven by multiple RTLSignals each predicated on a RTLSignal to form a multiplexer.
- `RTLConst` - a constant value.
- `RTLOp` - a functional unit with one, two or three operands.
- `RTLWidth` - the bit width of an RTLSignal (i.e. [31:0])

As an example let's implement the following Verilog using the RTL data structure:

```
module bitwise_AND_no_op_bitwise_OR_2to1mux_32bit
#(parameter WIDTH=32)
(
    input signed [WIDTH-1:0] data1,
    input signed [WIDTH-1:0] data2,
    input signed [WIDTH-1:0] data3,
    input signed [WIDTH-1:0] data4,
    input signed [WIDTH-1:0] data5,
    input signed [WIDTH-1:0] data6,
    input select,
    input clk,
    output reg [WIDTH-1:0] dataout
);
    reg signed [WIDTH-1:0] data1_reg;
    reg signed [WIDTH-1:0] data2_reg;
    reg signed [WIDTH-1:0] data3_reg;
    reg signed [WIDTH-1:0] data4_reg;
    reg signed [WIDTH-1:0] data5_reg;
    reg signed [WIDTH-1:0] data6_reg;
    reg signed [WIDTH-1:0] w1;
    reg signed [WIDTH-1:0] w2;
    reg signed [WIDTH-1:0] w3;

    always @ (posedge clk)
    begin
        data1_reg <= data1;
        data2_reg <= data2;
        data3_reg <= data3;
        data4_reg <= data4;
        data5_reg <= data5;
        data6_reg <= data6;

        dataout <= (w1 & w2) | w3;
    end

    always @ (*)
    begin
        if (select==0)
        begin
            w1 <= data1_reg;
            w2 <= data2_reg;
            w3 <= data3_reg;
        end
        else
        begin
            w1 <= data4_reg;
            w2 <= data5_reg;
            w3 <= data6_reg;
        end
    end

endmodule
```

The RTL data structure for the above Verilog looks like:

```
RTLModule *rtl = new
    RTLModule("bitwise_AND_no_op_bitwise_OR_2to1mux_32bit");
rtl->addIn("clk");
```

```
RTLSignal *select = rtl->addIn("select");

rtl->addParam("WIDTH", "32");
RTLWidth *width = new RTLWidth("WIDTH-1");
std::map<int, RTLSignal*> inputs;
for (int i = 1; i <=6; i++) {
    std::string name = "data" + utostr(i);
    RTLSignal *in = rtl->addIn(name, width);

    RTLSignal *reg = rtl->addReg(name + "_reg", width);
    reg->connect(in);
    inputs[i] = reg;
}
RTLSignal *dataout = rtl->addOutReg("dataout", width);

RTLOp *cond_zero = new RTLOp(RTLOp::EQ);
cond_zero->setOperand(0, select);
cond_zero->setOperand(1, new RTLConst("0"));

RTLOp *cond_one = new RTLOp(RTLOp::EQ);
cond_one->setOperand(0, select);
cond_one->setOperand(1, new RTLConst("1"));

RTLSignal *w1 = rtl->addWire("w1", width);
w1->addCondition(cond_zero, inputs[1]);
w1->addCondition(cond_one, inputs[4]);

RTLSignal *w2 = rtl->addWire("w2", width);
w2->addCondition(cond_zero, inputs[2]);
w2->addCondition(cond_one, inputs[5]);

RTLSignal *w3 = rtl->addWire("w3", width);
w3->addCondition(cond_zero, inputs[3]);
w3->addCondition(cond_one, inputs[6]);

// Note: you can pass an instruction to RTLOp's constructor
RTLOp *op_and = new RTLOp(RTLOp::And);
op_and->setOperand(0, w1);
op_and->setOperand(1, w2);

RTLOp *op_or = new RTLOp(RTLOp::Or);
op_or->setOperand(0, op_and);
op_or->setOperand(1, w3);

dataout->connect(op_or);

// to print out verilog
Allocation *allocation = new Allocation(&M);
allocation->addRTL(rtl);
VerilogWriter *writer = new VerilogWriter(Out, allocation);
writer->printRTL(rtl);
```

Signal Truncation

To get the lower 32 bits of a 64 bit signal:

```
RTLOp *lower = rtl->addOp(RTLOp::Trunc);
lower->setCastWidth(RTLWidth("32"));
lower->setOperand(0, signal_64);
```

To get the upper 32 bits of a 64 bit signal use a shift followed by the truncation above:

```
RTLOp *shift = rtl->addOp(RTLOp::Shr);
shift->setOperand(0, signal_64);
shift->setOperand(1, new RTLConst("32"));
RTLOp *upper = rtl->addOp(RTLOp::Trunc);
upper->setCastWidth(RTLWidth("32"));
upper->setOperand(0, shift);
```

Alternatively you can use the truncation operator directly:

```
RTLOp *upper = rtl->addOp(RTLOp::Trunc);
upper->setCastWidth(RTLWidth("63", "32"));
upper->setOperand(0, signal_64);
```

4.7.2 GenerateRTL

`GenerateRTL` uses the scheduling and binding algorithms to generate the final RTL data structure for the synthesized circuit.

4.7.3 VerilogWriter

`VerilogWriter` prints an `RTLModule` as Verilog, the memory controller, testbench, and required avalon signals.

4.7.4 SDC-Based Scheduling

The scheduler returns a `FiniteStateMachine` object for each LLVM function.

`FiniteStateMachine` stores `State` objects in a doubly-linked list. The `State` class stores a sequential list of instructions and the next state transitions.

The `SchedulerDAG` class creates an InstructionNode for each instruction and computes memory and data dependencies. `InstructionNodes` also store the propogation delay of the instruction. The `SchedulerMapping` class maps `InstructionNodes` to control steps.

The SDC scheduler is based on the formulation described in [Cong06]. Scheduling is formulated mathematically, as a system of equations to be solved. The formulation is a linear program (LP) that can be solved in polynomial time. SDC stands for System of Difference Constraints. All of the constraints in the LP have the form:

```
x1 - x2 REL y
```

where REL is a relational operator: EQUALS, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO. Constraints, in this form, are "difference constraints", hence the name SDC. We use the lpsolve open source linear system solver. See lpsolve.

The advantage of SDC is its flexibility: different styles of scheduling, with different types of constraints, can all be elegantly rolled into the same mathematical formulation. By using SDC-based scheduling within LegUp, we bring its scheduler closer to state-of-the-art.

Each of the supported devices (e.g. Cyclone II, Stratix III, etc) has a default clock period constraint, which we have determined as reasonable to produce good wall-clock times for a basket of programs. Chaining of operators in a cycle is permitted, within the clock period constraint limits.

A highly relevant constraint i found in the relevant device-specific file, such as `boards\CycloneII\CycloneII.tcl`. This sets the target clock period for LegUp HLS, used in SDC scheduling:

- **CLOCK_PERIOD**: Setting this parameter to a particular integer value in ns will set the clock period constraint.

The `examples/legup.tcl` file sets the following parameters which control the SDC scheduler:

- **SDC_NO_CHAINING**: Disable chaining of operations in a clock cycle. This will achieve the maximum amount of pipelining. The **CLOCK_PERIOD** parameter is useless when this is set.
- **SDC_ALAP**: Perform as-late-as-possible (ALAP) scheduling instead of as-soon-as-possible (ASAP).
- **SDC_DEBUG**: Cause debugging information to be printed from the scheduler.

Relevant source files for SDC scheduling: SDCScheduler.h and SDCScheduler.cpp. In the .cpp file, start by looking at the createMapping() method, which is the top-level method that implements the flow of SDC scheduling.

Known Issues with SDC Scheduler

- Doesn't support global scheduling across basic block boundaries
- Instructions from different basic blocks can never be in the same state

4.8 Binding

Binding uses the libhungarian-v0.1.2 library to solve bipartite weighted matching. This is the problem of finding the optimal assignment (assigning a set of jobs to a set of machines) in O(n^3), where n=max{#jobs, #machines}. Bipartite weighted matching is used to minimize the number of operations that share a functional unit.

4.8.1 Pattern Sharing Introduction

In the Legup 1.0 release, which targeted Cyclone II, Binding shared only dividers and remainders.

Binding has been modified to share other types of operations, as well as larger computational patterns. This was shown to reduce area on Stratix IV.

4.8.2 Enabling and Disabling Pattern Sharing

The `examples/legup.tcl` file sets the following parameters which control pattern sharing:

```
# Maximum chain size to consider. Setting to 0 uses Legup 1.0 original
# binding
# SET TO 0 TO DISABLE PATTERN SHARING
set_parameter PS_MAX_SIZE 10

# The settings below should all be nonzero, but can be disabled when
# debugging
# if set, these will be included in patterns and shared with 2-to-1 muxing
set_parameter PATTERN_SHARE_ADD 1
set_parameter PATTERN_SHARE_SUB 1
set_parameter PATTERN_SHARE_BITOPS 1
set_parameter PATTERN_SHARE_SHIFT 1
```

Setting PATTERN_SHARE_ADD, PATTERN_SHARE_SUB, PATTERN_SHARE_BITOPS and PATTERN_SHARE_SHIFT will share these operations when constructing computational patterns. Note that all 4 should be set when sharing for best results, but the parameters provide a means for debugging. Setting these four parameters all to 0 also results in the original LegUp Binding (equivalent to setting PS_MAX_SIZE to 0). However PS_MAX_SIZE takes precedence, so for example even if PATTERN_SHARE_ADD is set to 1, if PS_MAX_SIZE = 0 then LegUp original Binding will be active. i.e. both these examples will bind as in LegUp 1.0:

```
set_parameter PS_MAX_SIZE 0

set_parameter PATTERN_SHARE_ADD 1
set_parameter PATTERN_SHARE_SUB 1
set_parameter PATTERN_SHARE_BITOPS 1
set_parameter PATTERN_SHARE_SHIFT 1

set_parameter PS_MAX_SIZE 1

set_parameter PATTERN_SHARE_ADD 0
set_parameter PATTERN_SHARE_SUB 0
set_parameter PATTERN_SHARE_BITOPS 0
set_parameter PATTERN_SHARE_SHIFT 0
```

4.8.3 Writing Patterns to DOT and Verilog Files

Patterns found can also be written to .dot and .v files.

Setting the PS_WRITE_TO_DOT parameter to be nonzero will save all patterns of size > 1 to .dot files, and then convert these to .pdf files so that patterns may be visualized. The file name includes the pattern size and the frequency of occurrence.

The Graphviz graph visualization software can be downloaded from: http://www.graphviz.org/Download.php

Similarly, for experimental purposes, it is possible to create a verilog module for each pattern, by setting the PS_WRITE_TO_VERILOG parameter nonzero. This creates a .v file for that specific pattern with the same filename as the .dot and .pdf files.

To avoid writing patterns of any frequency to these files, the parameter FREQ_THRESHOLD lets only patterns shared with frequency greater than or equal to this threshold to be written to dot, pdf or verilog files.

The dot, pdf and verilog, files will be created in folders created for each function (given the function name).

4.8.4 Loop Pipelining

To turn on loop pipelining use the tcl command *loop_pipeline*. For working examples that use loop pipelining, look in `legup/examples/pipeline/`.

Loop pipelining supports simple array array based dependencies like:

```
loop: for (i = 1; i < N; i++) {
    a[i] = a[i-1] + 2
}
```

Loop pipelining also supports resource constraints (for instance dual port memories). To avoid being constrained by global memory ports we highly recommend turning on *LOCAL_RAMS* when loop pipelining to increase memory bandwidth.

LegUp can only pipeline loops with a single basic block. If there is control flow inside the loop then please turn on if-conversion using the *set_combine_basicblock* tcl command. Or you can manually convert the if statements into sequential code using the C ternary operator ”?:”.

Modulo scheduling rearranges the operations from one iteration of the loop into a schedule that can be repeated at a fixed interval without violating any data dependencies or resource constraints. This fixed interval between starting successive iterations of the loop is called the *initiation interval* (II) of the loop pipeline. The best pipeline performance and hardware utilization is achieved with an II of one, meaning that successive iterations of the loop begin every cycle, analogous to a MIPS processor pipeline.

If we are pipelining a loop that contains neither resource constraints or cross-iteration dependencies then the initiation interval will be one. Furthermore, in this case we can use a standard scheduling approach, which will correctly schedule the loop into a feed-forward pipeline. However, when the loop does contain constraints then the initiation interval may have to be greater than one. For instance, if two memory operations are required in the loop body but only a single memory port is available then the initiation interval must be two. In this case, modulo scheduling will be required because standard scheduling has no concept of an initiation interval. Standard scheduling assumes that operations from separate control steps do not execute in parallel when satisfying resource constraints, which is no longer true in a loop pipeline. For instance, the standard approach may schedule the first memory operation in the first time step and the second memory operation in the third time step, but if new data is entering the pipeline every two cycles then these memory operations will occur in parallel and conflict with the single memory port.

The LegUp loop pipelining implementation is in the following source files:

- `llvm/lib/Transforms/LegUp/LoopPipeline.cpp`
- `llvm/lib/Transforms/LegUp/ModuloScheduler.cpp`
- `llvm/lib/Transforms/LegUp/SDCModuloScheduler.cpp`
- `llvm/lib/Transforms/LegUp/ElementaryCycles.cpp`
- `llvm/lib/Transforms/LegUp/SDCSolver.cpp`

Please see our paper for more details on the loop pipelining algorithm:

- A. Canis, J.H. Anderson, S.D. Brown, “Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis,” IEEE International Conference on Field-Programmable Logic and Applications (FPL), Munich, Germany, September 2014.
- http://legup.eecg.utoronto.ca/pipelining_fpl_2014.pdf

After running loop pipelining please see the end of report file `pipelining.legup.rpt` for details on the final initiation interval (II) of the pipeline and the schedule. The loop pipelining pass annotates the LLVM IR with metadata.

For example, given the loop in `legup/examples/pipeline/simple/simple.c`:

```
#define N 4
loop: for (i = 0; i < N; i++) {
    printf("Loop body\n");
    printf("a[%d] = %d\n", i, a[i]);
    printf("b[%d] = %d\n", i, b[i]);
    c[i] = a[i] + b[i];
    printf("c[%d] = %d\n", i, c[i]);
}
```

After running “make”, we can look at a snippet from the bottom of `pipelining.legup.rpt`:

```
II = 1
Final Modulo Reservation Table:
FuName: main_0_a_local_mem_dual_port
time slot: 0
   issue slot: 0 instr:   %15 = load volatile i32* %scevgep4, align 4, !tbaa !3
   issue slot: 1 instr:   %11 = load volatile i32* %scevgep4, align 4, !tbaa !3
FuName: main_0_b_local_mem_dual_port
time slot: 0
   issue slot: 0 instr:   %16 = load volatile i32* %scevgep3, align 4, !tbaa !3
   issue slot: 1 instr:   %13 = load volatile i32* %scevgep3, align 4, !tbaa !3
FuName: main_0_c_local_mem_dual_port
time slot: 0
   issue slot: 0 instr:   store volatile i32 %17, i32* %scevgep2, align 4, !tbaa !3
   issue slot: 1 instr:   %18 = load volatile i32* %scevgep2, align 4, !tbaa !3
```

We see that the initiation interval is 1, which means we can start a new iteration of this loop every clock cycle. There are resource constraints caused by each dual-ported (two issue slots) local memory: a, b, c.

We can also see the final pipeline schedule:

```
Pipeline Table:
Total pipeline stages: 4
Stage:         0              1              2              3
   II:         0     |        0     |        0     |        0
 Time:         0     |        1     |        2     |        3
              %9             %9             %9             %9
       %scevgep2      %scevgep2      %scevgep2      %scevgep2
       %scevgep3      %scevgep3      %scevgep3      %scevgep3
       %scevgep4      %scevgep4      %scevgep4      %scevgep4
        <badref>       <badref>       <badref>       <badref>
             %10            %10            %10            %10
             %11            %11            %11            %11
               -            %12            %12            %12
             %13            %13            %13            %13
               -            %14            %14            %14
             %15            %15            %15            %15
             %16            %16            %16            %16
               -            %17            %17            %17
               -       <badref>       <badref>       <badref>
               -              -            %18            %18
               -              -              -            %19
             %20            %20            %20            %20
       %exitcond      %exitcond      %exitcond      %exitcond
        <badref>       <badref>       <badref>       <badref>
```

In this table, the time (clock cycle) is increasing left to right. The first column (time=0 cycles) contains every instruction from the loop body that will occur in the first clock cycle after starting the pipeline. We see that every instruction is repeated after every clock cycle (since II=1) and each successive instruction corresponds to the next iteration of the loop. However, some instructions (%19) don't start their first loop iteration until the 4th cycle of the pipeline. In this case, after 4 cycles the loop pipeline is in *steady state* and the instructions in the last column of this table will repeat for each successive iteration until the loop pipeline is done. Some instructions don't have a short label (store or calls) and are marked by <badref> in the table.

We can also look at a snippet of LLVM IR in the loop body in `simple.ll`:

```
%18 = load volatile i32* %scevgep2, align 4, !tbaa !4,
      !legup.pipeline.start_time !8, !legup.pipeline.avail_time !9,
      !legup.pipeline.stage !8

!9 = metadata !{metadata !"3"}
!8 = metadata !{metadata !"2"}
```

This load instruction is starts at cycle 2 and finishes at cycle 3 and is in pipeline stage 2 of this pipeline. This means that assuming the loop pipeline starts right away, this load will start loading after 2 cycles and will keep loading every cycle after that until the loop pipeline is finished.

The terminator instruction holds more useful information regarding the pipeline:

```
br i1 %exitcond, label %.preheader.preheader, label %legup_memset_4.exit,
    !legup.pipelined !2, !legup.II !2, !legup.totalTime !10, !legup.maxStage !9,
    !legup.tripCount !10, !legup.label !11, !legup.pipeline.start_time !3,
    !legup.pipeline.avail_time !3, !legup.pipeline.stage !3

  !2 = metadata !{metadata !"1"}
  !3 = metadata !{metadata !"0"}
  !9 = metadata !{metadata !"3"}
  !10 = metadata !{metadata !"4"}
  !11 = metadata !{metadata !"loop"}
```

This basic block is pipelined (`legup.pipelined` is 1). The initiation interval (II) is 1. The total number of timesteps in the pipeline is 4. The maximum stage is 3, so there are 4 stages (stages are indexed from 0). The tripcount (number of iterations) of the loop is 4. The label of the loop is "loop". The pipeline time step that the branch has been scheduled to is 0 and its pipeline stage is 0 (this can be ignored).

The `GenerateRTL.cpp` file handles the construction of the loop pipeline in the function `generateAllLoopPipelines()`. The FSM of the original function is modified so that there is a state waiting for the loop pipeline hardware to complete, looking at a snippet of the `scheduling.legup.rpt` file:

```
state: LEGUP_loop_pipeline_wait_loop_1_16
   %9 = phi i32 [ %20, %legup_memset_4.exit ], [ 0, %legup_memset_4.exit.preheader ], !legup.canonica
   br i1 %exitcond, label %.preheader.preheader, label %legup_memset_4.exit, !legup.pipelined !2, !le
   Transition: if (loop_1_pipeline_finish): LEGUP_F_main_BB_preheaderpreheader_17 default: LEGUP_loop
```

The state `LEGUP_loop_pipeline_wait_loop_1_16` waits until the signal `loop_1_pipeline_finish` is asserted before continuing to the next state.

The pipeline has the following control signals (all active high):

- loop_1_pipeline_start: starts the pipeline. Should not be asserted if the pipeline is running
- loop_1_pipeline_finish: the pipeline is finished and all computation is complete
- loop_1_valid_bit_*: the valid_bit signals form a shift register: `loop_1_valid_bit_1 <= loop_1_valid_bit_0` etc. Every time new data enters the pipeline a 1 is shifted into loop_1_valid_bit_0. If the valid bit is high for a time step then the input data is valid and that pipeline step can be performed
- loop_1_ii_state_*: this is a counter from 0 to II-1. This counter is only needed for pipelines with an initiation interval greater than 1
- loop_1_epilogue: no new data, the pipeline is being flushed
- loop_1_i_stage*: the value of the induction variable (i) at each pipeline stage

Most operations in the pipeline will use both the loop_1_ii_state and loop_1_valid_bit to determine when to execute. For example:

```
always @(posedge clk) begin
    if ((~(memory_controller_waitrequest) & ((loop_1_ii_state == 1'd0) & loop_1_valid_bit_3))) begin
            main_legup_memset_4exit_18_reg <= main_legup_memset_4exit_18;
    end
```

If a value calculated in one pipeline stage is used in a later pipeline stage, then LegUp will insert extra registers to store the value. We must create a register for every pipeline stage that must be crossed. These registers will be named _reg_stage* in the output Verilog. For example, if another operation needs to use `main_legup_memset_4_exit_scevgep6` in stage 2 then the register `main_legup_memset_4_exit_scevgep6_reg_stage2` would be used.

4.9 LegupConfig

`LegupConfig` is an Immutable LLVM pass that can read LegUp .tcl files. For instance, to read the functions that should be accelerated.

4.10 PreLTO

`PreLTO` pass computes the new size for memset and memcpy when applied to structs. The pass is needed because struct lengths may be different.

4.11 LLVM

4.11.1 Alias Analysis

Alias analysis, or memory disambiguation, is the problem of determining when two pointers refer to overlapping memory locations. An *alias* occurs during program execution when two or more pointers refer to the same memory location. *Points-to analysis*, a closely related problem, determines which memory locations a pointer can reference. Solving the alias and points-to analysis problems require us to know the values of all pointers at any state in the program, which makes this an undecidable problem in general.

Points-to analysis algorithms are categorized by flow-sensitivity and context-sensitivity. An approach is flow-sensitive if the control flow within the given procedure is used during analysis while a flow-insensitive approach ignores instruction execution order. Context-sensitive analysis considers the possible calling contexts of a procedure during analysis. Points-to analysis can either be confined to a single function, called intraprocedural, or applied to the whole program, called interprocedural. Points-to analysis algorithms have varying levels of accuracy and may be overly conservative, but for programs without dynamic memory, recursion, and function pointers, most pointers are resolvable at compile-time.

The compiler community has developed fast interprocedural flow-insensitive and context-insensitive algorithms. Andersen described the most accurate of these approaches, which formulates the points-to analysis problem as a set of inclusion constraints for each program variable that are then solved iteratively. Steensgaard presented a less accurate points-to analysis, which used a set of type constraints modeling program memory locations that can be solved in linear-time. In LegUp, we use the points-to analysis described by Hardekopf, which speeds up Andersen's approach by detecting and removing cycles that can occur in the inclusion constraints graph. We used the code from:

- https://code.google.com/p/addr-leaks/wiki/HowToUseThePointerAnalysis

The algorithm is described in the paper:

- Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07). New York, NY, USA, 290-299.

To aid pointer analysis, the C language now includes a pointer type qualifier keyword, *restrict*, allowing the user to assert that memory accesses by the pointer do not alias with any memory accesses by other pointers.

In LegUp, the alias analysis implementation occurs in two stages run by *Allocation::runPointsToAnalysis()*. First, we analyse the LLVM IR and set up a constraints problem in the *PADriver* class. Next we solve that constraints problem in the class *PointerAnalysis*. If there are bugs in the alias analysis they are likely within the *PADriver* class. These bugs occur when LegUp doesn't add a constraint caused by a particular LLVM instruction.

LegUp uses this alias analysis information to determine which memories a particular load/store instruction can access. If a memory is only accessed within a single function, we call that a local memory and create a block RAM inside the corresponding hardware module for that function. For alias analysis debugging information please see the *memory.legup.rpt* report file. The end of this file will contain information about which memory is local to a particular function or in global shared memory:

```
Final memory allocation:
Global Memories:
        RAM: float_exception_flags
        ROM: countLeadingZeros32_countLeadingZerosHigh
Local Memories:
        ROM: test_in Function: main
        ROM: test_out Function: main
```

You can force a memory to be global or local using the tcl commands *set_memory_global* or *set_memory_local*.

Alias analysis is also required for determining dependencies between load/store/call instructions. These instruction are not connected by a use-def chain like other LLVM instructions. In the worst case, without alias analysis, we must perform these memory instructions sequentially to avoid memory hazards.

For example:

```
store %a, 10
%b = load %a
```

This store and load have a read after write data dependency. The store must occur before the load, which means these instructions cannot be performed in parallel The LLVM `MemDep` analysis pass gives the dependencies between load/store/call instructions. Mod/Ref means modify/refer.

4.11.2 LLVM Intrinsics

The CHStone gsm benchmark requires the LLVM intrinsic function memset.i64(). By using the lowerIntrinsics function from CBackend we can turn this call into a memset() but we can't lower that. Even with -ffreestanding gcc requires: memcpy, memmove, memset, memcpy.

To handle this we create a custom intrinsic C functions defined in:

- `examples/lib/llvm/liblegup/` - source files
- `examples/lib/include/legup/intrinsics.h` - header file

These functions are compiled into an .a archive, which is linked with every Legup example.

4.12 Tips/Tricks

4.12.1 Compiling

To quickly compile only `llc` after modifying a file in `llvm/lib/Target/Verilog/`:

```
# llvm/utils must be on your path
makellvm llc
```

4.12.2 Debugging Segfaults

To debug segfaults in `llc` first make sure you have compiled a debug build. Do this by uncommenting the following line in `Makefile` and rerunning `make`:

```
#DEBUG_MODE = --disable-optimized
```

Then either update LLVM_BUILD in either `examples/Makefile.config` or your environment:

```
LLVM_BUILD=Debug+Asserts
```

Then use gdb:

```
> gdb llc
(gdb) run -march=v array.bc
```

To see DEBUG() print statements use the -debug flag:

```
llc -march=v -debug array.bc
```

4.12.3 Debugging RTL generated by LegUp

Printf

The easiest way to debug in Legup is to use C printf statements which will translate to Verilog **$display** statements which will print to the terminal when simulating the circuit in Modelsim.

make watch

To try make watch run:

```
cd array
make watch
```

If your hardware is correct ‘make watch’ will give a diff that returns nothing:

```
diff -q lli.txt sim.txt
```

`make watch` does the following:

1. Creates an annotated LLVM IR by adding a printf instruction at the end of every basic block that prints the current value of all registers modified in that basic block
2. Runs Legup on this annotated LLVM code to generate Verilog with $display statements at the end of each basic block
3. Simulates the Verilog with Modelsim, which will print out the state of registers as the program executes
4. Runs the annotated LLVM with the LLVM interpreter (`lli`)
5. diffs the two outputs to verify that the values of the registers are the same between software and hardware

Presently, the order the basic blocks are executed is identical when running in software or hardware, in the future this will change as basic blocks start to run in parallel. This will break this debugging method as the order of the basic block execution will be non-deterministic.

One caveat, registers that contain addresses to memory are not compared, because the software version of the code will have different addresses than the hardware. In some cases LLVM will cast a pointer to an integer, making it hard to identify that the register actually stores an address and this will lead to a false mismatch.

4.13 LegUp Quality of Results

To determine the LegUp quality of results we use the CHStone benchmark suite and Dhrystone. These are tracked on our quality of results page.

The horizontal axis shows the git revision, the rightmost being the latest revision. Click on a graph to zoom in, and click on a particular revision to view the git log message for that revision. Latency metrics are from a functional simulation using Modelsim. Area and fmax is provided from Quartus after place and route.

CHAPTER

FIVE

DEBUGGING

LegUp includes a source-level debugger. This allows for debugging of the produced RTL design using a software-like debugger that supports single-stepping, breakpoints, and variable inspection.

The debugger supports the following modes of execution:

1. **In-system interactive**: The design is implemented and running on the FPGA, and the debugger application is connected to the FPGA board. As the user single-steps through execution, the design on the FPGA is executed in lock-step by enabling and disabling the clock to the circuit.
2. **In-system record and replay**. In this mode, the user can run the design at full speed, and the circuit execution is captured into on-chip memories. Once the circuit is stopped, the execution history can be retrieved and debugging can be performed on a replay of the execution. (This is similar to an embedded logic analyzer, except that debugging is performed using a software-like debugger, rather than a waveform.) Compression techniques are used, tailored to each design, to obtain long execution traces. (See [*1*] and [*2*])
3. **Simulation**. This mode supports the same software-like debug experience as above, but instead of being connected to an FPGA, execution is performed using Modelsim.
4. **C/RTL Verification**. In this mode the C code is executed using gdb, and the RTL is simulated in Modelsim. The tool automatically detects discrepancies, and reports them to the user. (See [*3*])

More information can be found in included papers [*References*].

Presently, the debugger only works with the hardware-only flow; it does not support the hybrid flow where a processor is present.

5.1 Installation

Install the necessary packages in Ubuntu:

```
sudo apt-get install python3-pyqt5 \
mysql-server libmysqlclient-dev python3-mysql.connector
```

During MySQL installation you may be prompted for a database username/password. Set these to:

```
username: root
password: letmein
```

If you are using a different username/password, you will need to update the `DEBUG_DB_USER` and `DEBUG_DB_PASSWORD` options in `examples/legup.tcl`.

Next you need to install pyserial. You can try the `python3-serial` package in Ubuntu, but at one point it was broken. To inst

1. Download pyserial-2.7.tar.gz from https://pypi.python.org/pypi/pyserial

2. Extract
3. sudo python3 setup.py install

5.2 Configuring your Design

There are example designs already configured for debugging. These are located in the examples/debug directory. The following describes how to configure a new design for debugging.

Add the following to your Makefile:

```
DEBUG_G_FLAG = 1
```

This will ensure the debug flag (`-g`) is enabled during Clang compilation.

If you want to perform in-system debugging, you need to add the following option:

```
DEBUGGER = 1
```

This will ensure that when you create a Quartus project (`make p`), the necessary debugger RTL files are also included in the project.

If you want to perform debugging without any optimizations, as presented in [*1*], include the following in your Makefile. If you are debugging with optimizations, as described in [*2*], skip this step.

```
NO_OPT = 1
NO_INLINE = 1
DEBUG_KEEP_VARS_IN_MEM = 1
```

If you do not already have a `config.tcl` file for the design, you will need to create one, and add the following to your Makefile:

```
LOCAL_CONFIG = -legup-config=config.tcl
```

Edit your `config.tcl`, and add the following to the top of the file (if not already included):

```
source ../legup.tcl
set_project CycloneII DE2 Tiger_SDRAM
```

This will configure the project for the DE2 board. Next, add the following to your `config.tcl` to enable debugging. This is required for any mode of debugging (simultion or in-system):

```
set_parameter DEBUG_FILL_DATABASE 1
set_parameter DEBUG_DB_SCRIPT_FILE <your_legup_dir>/examples/createDebugDB.sql
```

If you want to debug in-system, you will also need to add the following option, which will instruct LegUp to automatically instrument the RTL file with necessary debug logic.

```
set_parameter DEBUG_INSERT_DEBUG_RTL 1
```

If you want to perform in-system debug, with visiblility into variables located in datapath registers, as explained in [*2*], add the following option:

```
set_parameter DEBUG_CORE_TRACE_REGS 1
```

The following trace buffer optimizations are also available, as described in [*2*]:

```
set_parameter DEBUG_CORE_TRACE_REGS_DELAY_WORST 1
set_parameter DEBUG_CORE_TRACE_REGS_DELAY_ALL 1
set_parameter DEBUG_CORE_TRACE_REGS_DUAL_PORT 1
```

5.3 Compiling your Design

Once configured for debugging your design can be compiled as usual:

```
make
```

If you want to perform in-system debug, you will also need to generate a bitstream:

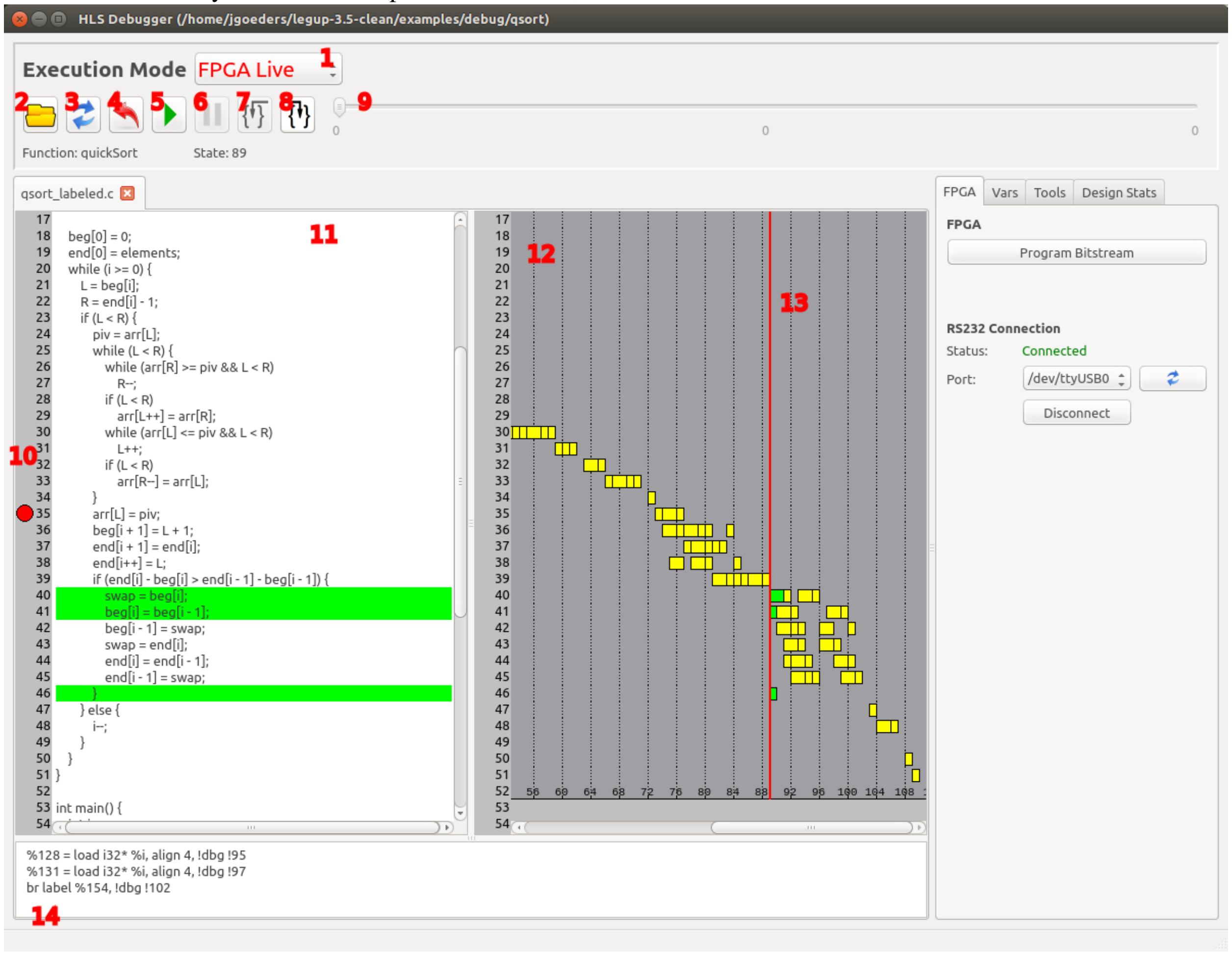
```
make p
make f
```

5.4 Using the Debugger

1. *(In-system only)* Connect the DE2 board to the computer using the RS232 port.
2. Launch the debugger from <legup-3.0>/dbg/debugger/src:

   ```
   python3 main.py
   ```

3. Once launched, open the folder containing your design.
4. *(In-system only)* On the FPGA tab, choose the /dev/ttyUSB0 port (or whatever port your serial is) and click 'Connect'. If you have RS232 permission issues, see here.



Execution Mode: FPGA Live

Function: quickSort   State: 89

```
    beg[0] = 0;
    end[0] = elements;
    while (i >= 0) {
      L = beg[i];
      R = end[i] - 1;
      if (L < R) {
        piv = arr[L];
        while (L < R) {
          while (arr[R] >= piv && L < R)
            R--;
          if (L < R)
            arr[L++] = arr[R];
          while (arr[L] <= piv && L < R)
            L++;
          if (L < R)
            arr[R--] = arr[L];
        }
        arr[L] = piv;
        beg[i + 1] = L + 1;
        end[i + 1] = end[i];
        end[i++] = L;
        if (end[i] - beg[i] > end[i - 1] - beg[i - 1]) {
          swap = beg[i];
          beg[i] = beg[i - 1];
          beg[i - 1] = swap;
          swap = end[i];
          end[i] = end[i - 1];
          end[i - 1] = swap;
        }
      } else {
        i--;
      }
    }
}

int main() {
```

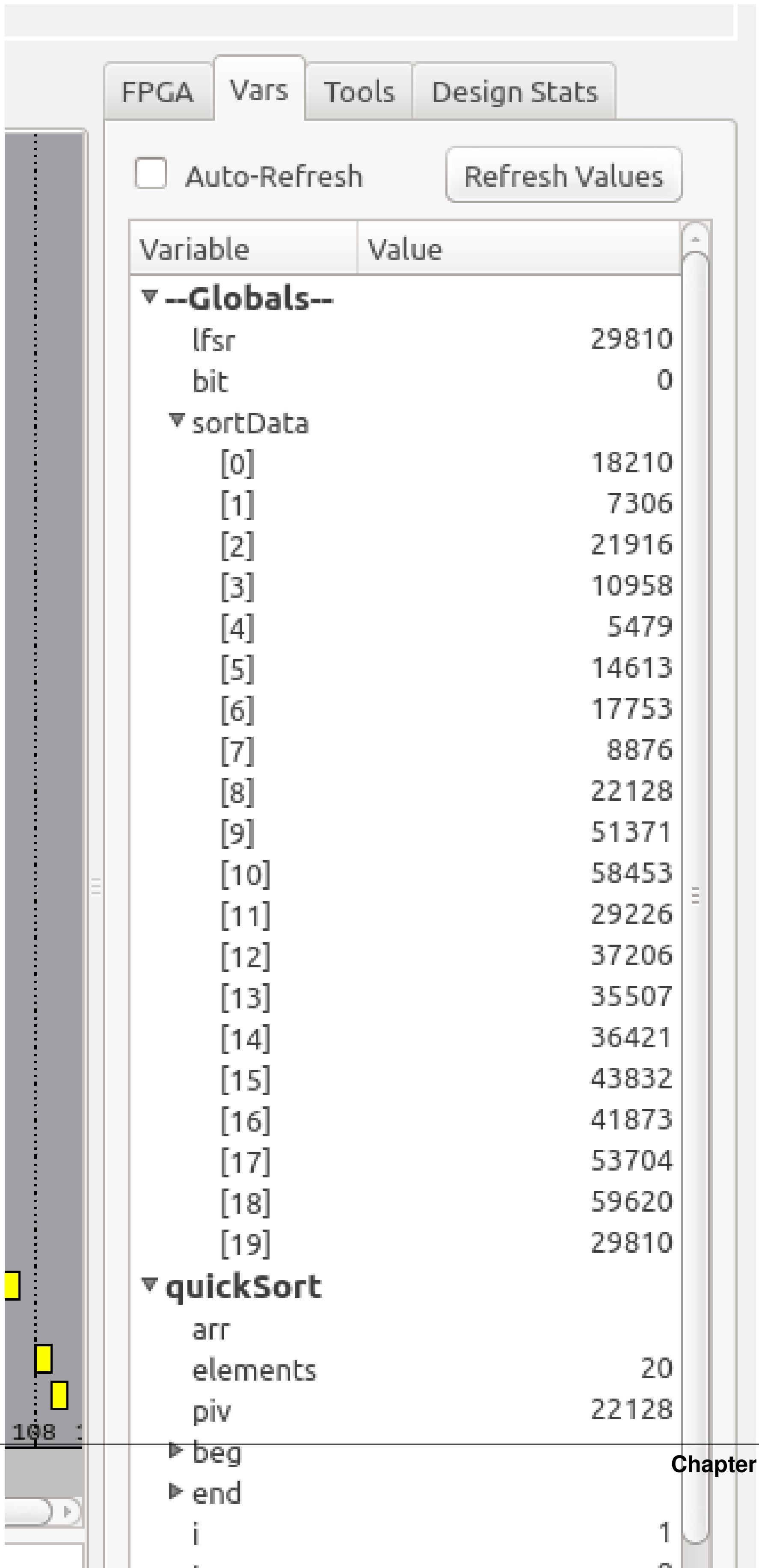
```
%128 = load i32* %i, align 4, !dbg !95
%131 = load i32* %i, align 4, !dbg !97
br label %154, !dbg !102
```

Status: Connected

Port: /dev/ttyUSB0

Item #	Name	Description
1	Mode	• **FPGA Live**: In-system Interactive debugger. Single stepping will execute the design on the FPGA in lock-step by enabling/disabling clock. • **FPGA Replay**: In-system Replay. Switching to this mode will retrieve the saved execution history from *FPGA Live* mode. Due to on-chip memory constraints, this may not contain the entire execution history. • **Simulation**: Debugging is performed without a connected FPGA. Modelsim is used to simulate circuit execution.
2	Open Design	Opens a new design. Make sure you compile (make) before opening.
3	Refresh	Refreshes the current state. This can be useful after reprogramming the bitstream.
4	Reset	Resets the design by asserting the reset signal. *Note: If the design relies upon memory initialization values, resetting will not be sufficient.*
5	Play	Run the design indefinitely (until completion or a breakpoint). *Note: Once a design completes execution it returns to its initial state. So if you hit run without a breakpoint the design will completely execute, but you won't see any visible change to the debugger.*
6	Pause	Pauses execution. (Mostly useful during simulation)
7	Step Back	Steps back one clock cycle. This is available during *FPGA Replay* or *Simulation*.
8	Step Forward	Steps forward one clock cycle.
9	Slider	This slider quickly moves through execution history. This is available during *FPGA Replay* or *Simulation*.
10	Breakpoints	Double click in this area to add a breakpoint. Breakpoints must be on lines with corresponding IR instructions. Breakpoints have a 1 cycle delay. *Note: The breakpoint is associated with the first FSM state of all IR instructions at that line of C code. If a line of C code has multiple IR instructions, and only a later instruction is executed, the breakpoint will not be triggered.*
11	Source code	Source code. All lines of source code which have an underlying IR instruction being executed will be high-

5.5 Reading Variables



Variable	Value
▼ **--Globals--**	
lfsr	29810
bit	0
▼ sortData	
[0]	18210
[1]	7306
[2]	21916
[3]	10958
[4]	5479
[5]	14613
[6]	17753
[7]	8876
[8]	22128
[9]	51371
[10]	58453
[11]	29226
[12]	37206
[13]	35507
[14]	36421
[15]	43832
[16]	41873
[17]	53704
[18]	59620
[19]	29810
▼ **quickSort**	
arr	
elements	20
piv	22128
▶ beg	
▶ end	
i	1

This pane provides information on the source-code variables. The availability of variable values depends on the mode of operation, and how the variable is mapped to FPGA resources (memory vs. register).

If all compiler optimizations are disabled, all variables will be located in on-chip memory.

Once compiler optimizations are enabled, some variables are optimized to datapath registers, some are replaced with constants, some remain in memory, and others are completely optimized away. (See [2]). The value of variables optimized away can never be obtained, while those that are optimized to constants are always available to the user. Variables in on-chip memory and datapath registers are available under certain circumstances (see table below).

Mode	Variable Resource	Variable Availability
FPGA Live	On-chip memory	Always available
	Datapath register	Not available
FPGA Replay	On-chip memory	If the variable is updated (written to) during the replay period, the value is available after the first write (obtained from execution trace). If the variable is never updated, the value is always available (obtained by reading directly from FPGA memory).
	Datapath register	The variable is available after the first time it is updated.
Simulation	On-chip memory	The variable is available after the first time it is read/written.
	Datapath register	Not available (has not been implemented yet)

When reading the variable value, one of the following will be displayed.

Message	Description
Numeric value	The variable value was obtained successfully, and the decimal representation is given.
<N/A>	The variable exists in the RTL, but is not available (See table above).
<Optimized Away>	The variable has been optimized out of the RTL, due to compilzer optimizations.
<Unknown>	The variable is tracked, but at this point in the replay the variable has not been updated yet (See table above).
<Undefined>	The variable value is undefined. This occurs for uninitialized variables.

The GUI supports displaying simple integer variables (char, short, int, etc.), and provides an interface for viewing struct and array members. Other types of variable inspection (pointer dereferencing, etc) are not yet supported.

By default, when connected to an FPGA performing interactive debugging, variables are not automatically updated. This is because the serial connection to the board is relatively slow, and loading the variables each cycle can slow down the speed at which you can single-step. The *Auto-Refresh* checkbox is provided to override this behaviour. When working with simulation or replay data the serial connection speed is irrelevant, and variables will be automatically refreshed after each single-step.

5.6 SW/RTL Discrepancy Checking

The debugger supports SW/RTL discrepancy checking, as described in [3]. Discrepancy checking can be performed by clicking the *Run* button on the *Tools* tab. Discrepancy checking simulates the RTL in Modelsim executes the software using gdb, and then compares the execution traces for differences. A report of any differences can be viewed by clicking the *View Report* button.

To use discrepancy checking, ensure the following conditions are met:

1. The following options must be added to the config.tcl for the design:

```
set_parameter INSPECT_DEBUG 1
set_parameter NO_ROMS 1
```

2. Compile a binary of the design using gcc, ensuring the name of the executable is <design_name>.out:

   ```
   gcc -g array.c -o array.out
   ```

3. Ensure that RTL debug instrumentation is not enabled. This interferes with the RTL simulation. The DEBUG_INSERT_DEBUG_RTL option should not be enabled.

4. The discrepancy checking uses a MySQL debug database that is separate from the MySQL database used by the rest of the debugging tools. Unfortunately this database only holds one design at a time. Each time you compile a design with the INSPECT_DEBUG option enabled the database is erased and repopulated. Make sure you do not compile any other designs between compilation and discrepancy detection of the design of interest.

5. The discrepancy detection only works with unoptimized designs. Make sure these options are included in the Makefile of the design:

   ```
   NO_OPT = 1
   NO_INLINE = 1
   DEBUG_KEEP_VARS_IN_MEM = 1
   ```

5.7 References

[1] Jeffrey Goeders and Steven Wilton. Effective FPGA Debug for High-Level Synthesis Generated Circuits. *In International Conference on Field Programmable Logic and Applications*, September 2014. [http] [2] Jeffrey Goeders and Steven Wilton. Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs. *In International Symposium on Field-Programmable Custom Computing Machines*, pages 127-134, May 2015. [3] Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. Source-Level Debugging for FPGA High-Level Synthesis. *In International Conference on Field Programmable Logic and Applications*, September 2014. [http]

CHAPTER

SIX

HYBRID PARTITION ANALYSIS

LegUp provides not only pure hardware flow, but also hardware/software hybrid flow. This section is to introduce one of the steps in hybrid design flow, suggesting candidate functions to be accelerated. At the end of this step, users can find software cycle counts and estimated hybrid cycle counts of the functions.

6.1 Software Cycles

Software cycles can be gathered by running the program on a soft MIPS processor with a hardware profiler monitoring the execution. Other than actually executing the C program on the soft processor on FPGA, we also provide a simulation flow to gather the profiling data. Note that currently both the on-board-execution and simulation flows support only DE2 board, so please make sure the variable *FPGA_board* in example/Makefile.config is set to *DE2* instead of *DE4*. To run the flows for other devices, some minor changes are required.

6.1.1 Profile on FPGA

To profile a C program's actual execution on the soft processor, the make target to use is *profile_tiger_on_board*. To eliminate the print functions (printf), run the make target with variable *PRINTF_OFF* defined, e.g.,

```
make profile_tiger_on_board PRINTF_OFF=elimatating_print_function.
```

```
profile_tiger_on_board: tiger_with_profiler
        make pre_on_board_profiling
        make on_board_profiling
        make post_on_board_profiling
```

The *tiger_with_profiler* make target compiles the C programs to executable and generates byte code to be downloaded to sdram on the FPGA board. It also generates the hash parameters that are used to configure the profiler, by calling another make target *gen_hash_for_profiler*.

Then the *pre_on_board_profiling* make target prepares the files (*.dat) in on_board directory. These files contains the program's instruction byte code as well as the hash parameters. These contents will be downloaded to the FPGA board (SDRAM or the FPGA Chip) from your host computer.

Next, the *on_board_profiling* make target programs the FPGA and uses *system-console* to run a tcl file named *profile.tcl*. This tcl file downloads the program's instruction and hash parameters to FPGA, starts program's execution and retrieves profiling data when program finishes exection. The profiling data retrieved from profiler are saved in on_board directory. The profiler can be configured with 2 profiling types and 5 profiling targets. The profiling types can be hierarchical or flat. The profiling targets can be 1) number of instructions, 2) execution cycles, 3) stall cycles, 4) instruction stall cycles and 5) data stall cycles. Currently *on_board_profiling* profiles the program with all the 10 configurations (2*5). Please refer to *Makefile.common* for more detail.

Finally, *post_on_board_profiling* organizes the profiling data into a report file named *$(NAME).profiling.rpt* in the on_board directory.

The programming bit stream used in this flow is pre-compiled and targetting the Cyclone-II FPGA chip (EP2C35F672C6). The programming bit stream is generated by compiling the Quartus project located at *$LEGUP_DIR/tiger/processor/tiger_DE2*. The pre-compiled bit stream is located at *$LEGUP_DIR//tiger/tool_source/profiling_tools/tiger_leap.sof*.

6.1.2 Profile through Simulation

To simulation the program's execution, run *make tigersim PROFILER_ON=TRUE*. Define varaible *PRINTF_OFF* to eliminate the print functions. Define varaible *SHOW_PERCENTAGE* to display percentage information in generated report files (It does not make sence to show percentage in flat profiling). To configure the profiling type, define *DO_HIER=1* to enable hierarchical profiling and *DO_HIER=0* to do flat profiling. The default profiling type is hierarchical profiling. To configure the profiling target, define the variable *CNT_INC* with corresponding values,

```
CNT_INC=00001    # number of instructions
CNT_INC=00010    # execution cycles (DEFAULT)
CNT_INC=00100    # stall cycles (sum of next two)
CNT_INC=01000    # stall cycles spent on fetching instruction
CNT_INC=10000    # stall cycles spent on fetching data
```

For example, the command

```
make tigersim PROFILER_ON=true PRINTF_OFF=true DO_HIER=1 CNT_INC=00100 SHOW_PERCENTAGE=true
```

hierarchically profiles the program's total cache stall cycles and adds percentage information in the generated report file.

6.2 Hybrid Cycles

6.2.1 Methodology

The runtime (hybrid cycles) of an accelerator in hybrid system consists of 4 major components, software wrapper cycles, hardware cycles, cycles to load data from global memory, and cycles to store data to global memory. Refer to slide.

6.2.2 Detail Implementation

This section goes through the hybrid cycles prediction flow for a function. The corresponding make target of this flow is *predictHybridCycle*:

```
# predict hybrid cycle of the first function specified in config.tcl
predictHybridCycle:
        $(MAKE) \
        printf_off PRINTF_OFF=eliminating_print_function \
        hybridFrontend \
        \
        predictHwOnlyRenamePass \
        hybridHWloweringLinking \
        predictNonAcceleratedOnlyPass \
        predictLoweringLinking \
        predictEmulTrace \
        predictInsertTrack \
        \
        hybridSwOnlyPass \
        hybridCompileCwrapper \
        hybridSWloweringLinking \
        hybridMIPSbackend \
        hybridMIPSbinUtils \
        \
        hybridVerilogbackend \
        \
        printf_on \
        predictStateTrace \
        predictDataCycle \
        predictInstrCycle \
        predictCollectData
```

Each make target used in *predictHybridCycle* is described below:

```
printf_off:
ifdef PRINTF_OFF
        cp $(LEVEL)/../tiger/tool_source/lib/no_uart_h $(LEVEL)/../tiger/tool_source/lib/uart.h
else
        cp $(LEVEL)/../tiger/tool_source/lib/uart_h $(LEVEL)/../tiger/tool_source/lib/uart.h
endif
```

printf_off replace library file uart.h with no_uart_h in order to remove all printf functions. It allows more accurate profiling and prediction by eliminating all printf functions.

```
#run front end to produce LLVM IR
hybridFrontend:
        # produces pre-link time optimization binary bitcode: $(NAME).prelto.bc
        $(FRONT_END) $(NAME).c -emit-llvm -pthread -c $(CFLAG) -mllvm -inline-threshold=-100 -o $(NAM
        $(LLVM_HOME)llvm-dis $(NAME).prelto.1.bc
        $(LLVM_HOME)opt -legup-config=config.tcl $(OPT_FLAGS) -legup-parallel-api < $(NAME).prelto.1
        $(LLVM_HOME)llvm-dis $(NAME).prelto.bc
```

hybridFrontend runs LLVM front end to produce pre link time optimization IR `$(NAME).prelto.bc`. This make target is also used as the first step in hybrid flow (make target *hybrid*).

```
# run hw-only-rename pass for hybrid
# this pass renames the descendents of accelerating function and set linkage for global variables
predictHwOnlyRenamePass: hybridHwOnlyPass
        $(LLVM_HOME)opt -legup-config=config.tcl $(OPT_FLAGS) -legup-hw-only-rename < $(NAME).prelto
        mv $(NAME).prelto.hw_rename.bc $(NAME).prelto.hw.bc
        $(LLVM_HOME)llvm-dis $(NAME).prelto.hw.bc
```

predictHwOnlyRenamePass runs *hybridHwOnlyPass* as the first step. *hybridHwOnlyPass* strips away non-accelerated functions in `$(NAME).prelto.bc` and saves the new IR in `$(NAME).prelto.hw.bc`. The second step uses LLVM opt command to run a pass called `-legup-hw-only-rename`. This pass

- sets the linkages of all the global variables to `LinkOnceAnyLinkage` so that the linking between HW-side and SW-side can be performed in *predictLoweringLinking*.
- renames the descendents of the accelerating function to maintain 2 sets of function definition in the IR `$(NAME).sw.bc` (produced in *predictLoweringLinking*). This ensures the correct counting of basic block executions (*predictStateTrace*).

The pass is implemented in `RenameHwOnly.cpp`. Also, `$(NAME).prelto.hw.bc` is replaced by `$(NAME).prelto.hw_rename.bc` at the last step.

```
#Lower HW IR and link
hybridHWloweringLinking:
        # HW part
        # performs intrinsic lowering so that the linker may be optimized
        $(LLVM_HOME)opt $(OPT_FLAGS) -legup-prelto < $(NAME).prelto.hw.bc > $(NAME).hw.lowered.bc
        # produces $(NAME).bc binary bitcode and a.out shell script: lli $(NAME).bc
        $(LLVM_HOME)llvm-ld $(LDFLAG) $(NAME).hw.lowered.bc $(LEVEL)/lib/llvm/liblegup.a $(MIPS_LIB)/
        $(LLVM_HOME)llvm-dis $(NAME).hw.bc
```

hybridHWloweringLinking is also used in regular hybrid flow. In prediction flow, this target produces a new IR `$(NAME).hw.bc` based on the renamed version of HW-side IR `$(NAME).prelto.hw.bc`. This step makes sure the IR `$(NAME).hw.bc` in prediction flow has the same implementation as that in regular hybrid flow. `$(NAME).hw.bc` will be used to 1) link back with SW-side IR in *predictLoweringLinking* and 2) generate HW logic (Verilog) in *hybridVerilogbackend*.

```
# run non-accelerated-only pass for hybrid
# this pass remove only the accelerated function from $(NAME).prelto.bc
predictNonAcceleratedOnlyPass:
        $(LLVM_HOME)opt -legup-config=config.tcl $(OPT_FLAGS) -legup-non-accelerated-only < $(NAME).p
        $(LLVM_HOME)llvm-dis $(NAME).prelto.sw.bc
```

predictNonAcceleratedOnlyPass uses LLVM opt command to run a pass called `-legup-non-accelerated-only`. This pass is similar to the pass `-legup-sw-only` which is used in regular hybrid flow (*hybridSwOnlyPass*). It simply removes the accelerating function without substitute in a wrapper function. The produced IR is saved in `$(NAME).prelto.sw.bc` which is the so called SW-side IR.

```
#lower the SW IR(w/o wrapper) and link with \*.hw.bc for prediction purpose
predictLoweringLinking:
        $(LLVM_HOME)opt $(OPT_FLAGS) -legup-prelto $(NAME).prelto.sw.bc -o $(NAME).sw.lowered.bc
        # link with hw part
        $(LLVM_HOME)llvm-ld -disable-inlining -disable-opt $(NAME).sw.lowered.bc $(NAME).hw.bc $(LEVE
        $(LLVM_HOME)llvm-dis $(NAME).sw.bc
```

predictLoweringLinking links back HW-side IR `$(NAME).hw.bc` and SW-side IR `$(NAME).sw.lowered.bc`. The generated IR `$(NAME).sw.bc` is a complete but manipulated implementation of the original program.

```
# generates executable of the IR generated by predictLoweringLinking
# execute the program by gxemul and direct gxemul output to $(ACCELERATOR_NAME).raw.trace
# this raw trace will be used in predictDataCycle and predictInstrCycle
predictEmulTrace:
        # pass -legup-num-params reports # of arguments of each function
        $(LLVM_HOME)opt $(OPT_FLAGS) -legup-num-params < $(NAME).sw.bc > /dev/null
        grep $(ACCELERATOR_NAME) num_params.legup.rpt > $(ACCELERATOR_NAME).num_params.rpt
        $(LLVM_HOME)llvm-ld $(LDFLAG) -disable-inlining -disable-opt $(NAME).sw.bc $(LEVEL)/lib/llvm/
        $(LLVM_HOME)llc $(NAME).emul.bc -march=mipsel -relocation-model=static -mips-ssection-thresho
        $(LLVM_HOME)llvm-dis $(NAME).emul.bc
        $(MIPS_PREFIX)as $(NAME).s -mips1 -mabi=32 -o $(NAME).o -EL
        $(MIPS_PREFIX)ld -T $(MIPS_LIB)/prog_link_emul.ld -e main $(NAME).o -o $(NAME).emul.elf -EL -
        $(MIPS_PREFIX)objdump -D $(NAME).emul.elf > $(NAME).emul.src
        ##############################################
        ##   Please type "quit" to end simulation   ##
        ##############################################
        gxemul -E oldtestmips -e R3000 $(NAME).emul.elf -p `$(MIPS_LIB)/../find_ra $(NAME).emul.src`
```

predictEmulTrace

- runs a pass -legup-num-params to report # of arguments of each function
- produces the executable `$(NAME).emul.elf` for IR `$(NAME).sw.bc`

- emulates the program using GXemul and direct emulation output to `$(ACCELERATOR_NAME).raw.trace`

The executable `$(NAME).emul.elf` and the "raw" trace `$(ACCELERATOR_NAME).raw.trace` will be used later in *predictDataCycle* and *predictInstrCycle*.

```
# insert print statement at the end of each BB in the IR generated by predictLoweringLinking
predictInsertTrack:
        $(LLVM_HOME)opt $(OPT_FLAGS) -legup-track-bb < $(NAME).sw.bc > $(NAME).track_bb.bc
        $(LLVM_HOME)llvm-dis $(NAME).track_bb.bc
```

predictInsertTrack runs a pass `-legup-track-bb` on top of the complete but maniputated IR `$(NAME).sw.bc`. In the new IR `$(NAME).track_bb.bc`, the print statements for tracking purpose, are inserted in front of all the call, return instructions, and at end of the basic blocks.

```
\
hybridSwOnlyPass \
hybridCompileCwrapper \
hybridSWloweringLinking \
hybridMIPSbackend \
hybridMIPSbinUtils \
```

hybridSwOnlyPass, *hybridCompileCwrapper*, *hybridSWloweringLinking*, *hybridMIPSbackend* and *hybridMIPSbinUtils* are used in regular hybrid flow. In prediction flow, we run these five targets to get the same SW-side executable `$(NAME).elf` in regular hybrid flow. As same as `$(NAME).emul.elf`, `$(NAME).elf` will also be used in *predictDataCycle* and *predictInstrCycle*.

```
#compile HW IR to Verilog backend for hybrid
hybridVerilogbackend:
        export LEGUP_ACCELERATOR_FILENAME=$(NAME); \
        $(LLVM_HOME)llc -legup-config=config.tcl -legup-config=parallelaccels.tcl $(LLC_FLAGS) -marc
        cp $(NAME).v $(PWD)/tiger/
        #only move .mif files if it exists
        find . -maxdepth 1 -name "\*.mif" -print0 | xargs -0 -I {} mv {} ./tiger
```

hybridVerilogbackend is also used in regular hybrid flow to run backend HLS for HW-side IR. The purpose of running this target in prediction flow is to gather scheduling information of the hardware. Here, the scheduing information is generated based on the renamed version of HW-side IR and saved in `scheduling.legup.rpt`.

```
# after the accelerating function is synthesize to HW, this target combine scheduling information an
predictStateTrace:
        # interpret IR that is generated by predictInsertTrack
        $(LLVM_HOME)lli $(NAME).track_bb.bc | grep 'Track@' | sed 's/Track@//' > $(ACCELERATOR_NAME)
        # combine the BB trace and scheduling information of BBs in order to get HW cycle
        perl $(PROF_TOOLS)/../partition_analysis/get_hw_cycle.pl $(ACCELERATOR_NAME).lli_bb.trace $(
        cat $(ACCELERATOR_NAME).acel_cycle.rpt
```

predictStateTrace is used to estimate HW cycle. It uses LLVM lli command to interpret `($NAME).track_bb.bc` which has print statements inserted. Output is redirected to `$(ACCELERATOR_NAME).lli_bb.trace`. Then a perl script `get_hw_cycle.pl` takes the trace as input, based on the scheduled lengthes of BBs in `legup.scheduling.rpt` and reports HW cycles of accelrating function. Result is saved in `$(ACCELERATOR_NAME).acel_cycle.rpt`.

```
# predict number of cycles spent on load/store from/to global memory
predictDataCycle:
        # get the global variables that will be accessed by accelerator
        grep '^@' $(NAME).hw.ll | sed 's/^@//' | sed 's/ .*//' > $(ACCELERATOR_NAME).hw_accessed_gv.
        # get the names, sizes and starting addresses of global variables from the pure SW src code
        $(MIPS_PREFIX)objdump -t $(NAME).emul.elf | grep '\s\.scommon\s\|\s\.rodata\s\|\s\.bss\s\|\s
        # get the names, sizes and starting addresses of global variables from the hybrid src code
        $(MIPS_PREFIX)objdump -t $(NAME).elf      | grep '\s\.scommon\s\|\s\.rodata\s\|\s\.bss\s\|\s
        # extract the traces of loads and stores from raw trace
        perl $(PROF_TOOLS)/../partition_analysis/extract_trace.pl $(NAME).emul.src $(ACCELERATOR_NAM
        # convert loading address from the gxemul src code to tiger src code
        perl $(PROF_TOOLS)/../partition_analysis/convert_ld_addr.pl $(ACCELERATOR_NAME).hw_accessed_
        # remove the addresses of Global CONST and local stack
        perl $(PROF_TOOLS)/../partition_analysis/gen_ld_addr.pl $(ACCELERATOR_NAME) $(ACCELERATOR_NA
        # run cache simulation
        ../$(LEVEL)/tiger/cache_simulator/cache_sim -file $(ACCELERATOR_NAME).ld_addr.trace -cachesi
        # report store cycle
        perl $(PROF_TOOLS)/../partition_analysis/get_store_cycle.pl $(ACCELERATOR_NAME) $(ACCELERATO
```

predictDataCycle is used to estimate number of cycles spent on load/store from/to global memory. The first line simply greps names of all the global variables from HW-side IR and save them in `$(ACCELERATOR_NAME).hw_accessed_gv.src`. (These GV are visible by HW-side IR. In other words, if a global variable is only accessed by passing-in address pointer as argument of the accelerating function, this global variable will not appear in `hw_accessed_gv` file. In our example testbenches, some global constants are accessed by passing-in address pointers as function arguments. These memory accesses should not be eliminated although they are global constants.) Also, the actual addresses of global variables in hybrid system are dumped out from `$(NAME).elf` and saved in `$(ACCELERATOR_NAME).gv_table.src`. The script `extract_trace.pl` extracts out loads and stores from the previously generated "raw" trace, calculates values of sp in processor when functions are just called, and saves traces seperately in `$(ACCELERATOR_NAME).extracted.store.trace` and `$(ACCELERATOR_NAME).extracted.load.trace`. To estimate load cycles, `convert_ld_addr.pl` firstly converts loading address from "GXemul" version to the actual "Tiger" version. In this step, all the loads to the global constants that are visible by HW-side IR will be marked with `<CONST>`. By elimating the loads of global constants and local stacks, the script `gen_ld_addr.pl` generates an address trace which can be treated as a complete trace of all the loads from data cache by both processor and accelerator in actual hybrid system. In this final verion of load trace `$(ACCELERATOR_NAME).ld_addr.trace`, the accelerator sections are marked with `<Accelerator Started>` at the begining and `<Accelerator Finished>` at the end. A cache simulator `cache_sim` reads the trace and reports the numbers of hits and misses by accelerator. Also, based on the extraced trace, the script `get_store_cycle.pl` ignores the stores towards the address space on local stack, and reports the number of stores towards global memory. (Since the data cache in current hybrid system adopts write-through policy, all the data stores will need to access global memory, and update `but not replace` the content in data cache.)

```
# predict number of cycles spent on fetching instructions
predictInstrCycle:
        # get the addresses of function from the pure SW src code
        $(MIPS_PREFIX)objdump -t $(NAME).emul.elf | grep '\s\.text\s' | grep -v '\.text$$' | sort -k
        # get the addresses of function from the hybrid src code
        $(MIPS_PREFIX)objdump -t $(NAME).elf      | grep '\s\.text\s' | grep -v '\.text$$' | sort -k
        # convert the instruction address and replace the accelerating function with wrapper functio
        perl $(PROF_TOOLS)/../partition_analysis/gen_instr_trace.pl $(ACCELERATOR_NAME) $(ACCELERATO
        # run cache simulation
        ../$(LEVEL)/tiger/cache_simulator/cache_sim -file $(ACCELERATOR_NAME).instr_addr.trace -cache
```

predictInstrCycle is to find out the number cycles spent on fetching instructions of wrapper function (Since the instructions in wrapper functions are generally simple, most of the cycles spent in wrapper function are to fetch instruction and to store arguments to accelerator). The first two lines dump out the function names and starting addresses from both "Tiger" and "GXemul" versions of exetables. `gen_instr_trace.pl` takes the "raw" trace and both function tables as input, translates instruction addresses from "GXemul" version to "Tiger" version (actual hybrid system version), substitutes the accelerated section with the instruction addresses of the wrapper function, and marks the accelerator section. The produced instruction trace `$(ACCELERATOR_NAME).instr_addr.trace` is expected to be the same as that in the actual hybrid system. Now, a cache simulator can be used to estimate the number of instruction hits and misses when executing wrapper function.

```
# collect prediction data
predictCollectData:
        perl $(PROF_TOOLS)/../partition_analysis/collect_prediction_data.pl $(NAME) $(ACCELERATOR_NAM
```

predictCollectData will parse out all parts of prediction data from previously generated report files and save the final prediction result in `$(NAME).hybrid_ prediction.csv` and `$(NAME).hybrid_prediction.rpt`. Both files will be either created or appended.

To run a complete partition analysis of a program, `predictAll` is created to profile the C program's execution in pure SW mode and estimate the hybrid cycles of the functions that take more than 5% of hierarchical runtime (includeing runtime of descendant functions) in SW.

CHAPTER

SEVEN

LEGUP OVER PCIE

The pcie/ directory contains working code for a new hybrid flow that uses a hard processor (x86) connected to a DE4 FPGA. The communication is done via the PCIe interface protocol. Similar to LegUp's MIPS hybrid flow, the PCIe flow allows the user to specify functions to compile into hardware accelerators.

7.1 User information

We recommend that you complete the Altera DE4 tutorial at http://www.altera.com/education/univ/materials/boards/de4/unv-de4-board.html with a DE4 to gain a better understanding on how the PCIe interface works and to physically set up the PCIe connection. You may use the code in driver/, as it is a compatible driver. The driver and initial pcie_tutorial/ design were modified from this tutorial.

Note: the following is still ongoing work, so it may not fully work as advertised. 'make pcie' requires pre-written C wrapper functions and a QSYS script to describe the system. We try to automatically generate the C wrapper and the QSYS script, but currently this is only functional for single-threaded functions that only pass by value. To compile with the generated wrappers, use 'make pcieAuto'. The examples directories contains some examples of C wrapers and QSYS scripts that allow pass by reference and multi-threaded functions, but automating these examples is still ongoing.

Once you have completed the tutorial, take a look at the vector_add example, which adds two arrays into a third array. In the vector_add example, config.tcl is used to specify that vector_add will get accelerated and synthesized onto the FPGA. Our flow supports passing in basic data types such as int len, used to pass in the array length, as well as pointers, which are used to pass the 3 arrays. The FPGA and hard processor have physically separate memory spaces, so the program must synchronize the memory to allow the accelerator to operate on a chunk of memory. We have provided the following API for memory synchronization:

```
#include "legup_mem.h"

// Allocate a block of memory on the FPGA and return a pointer to the FPGA memory
void * malloc_shared(size_t size, void *default_ptr); // ignore default_ptr for now
// Free an allocated pointer to FPGA memory
void * free_shared(void *ptr);

// Copy data to the FPGA from the host processor
memcpy_to_shared(void *dst_fpga_ptr, void *src_host_ptr, size_t size);
// Copy data from the FPGA to the host processor
memcpy_from_shared(void *dst_host_ptr, void *src_fpga_ptr, size_t size);
```

Note: the return value of malloc_shared() and the fpga_ptr arguments to the memcpy() functions are all pointers to FPGA memory, and do not make any sense to use on the host processor. We recommend making this explicit in the variable name by preceding with `SHARED_MEM_` or a similar prefix.

Once you understand what the code does, you can compile it. Type "make pcieSWAuto", which will create software executables. This produces three executables: vector_add.elf, vector_add_sw, and vector_add_memtest. vector_add.elf is the final program that will communicate with hardware accelerators. If you run this executable now, it should fail because it "Could not open device". This is because the DE4 is not yet programmed. vector_add_sw_memtest is a software-only implementation of the benchmark. It simulates the memory copying of the "legup_mem.h" functions, and its execution should mirror that of the hybrid executable. vector_add_sw is software-only simulation that does not do any copying. malloc_shared simply returns the default_ptr and the other functions do nothing. This executable is also compiled with "#define SW_ONLY" if pointer offsets need to be corrected for this version. For our work, the sw_memtest executable was very helpful to make sure our shared memory synchronization was correct and the sw executable was used as a fair software-only benchmark for performance comparison.

To generate the entire system, type "make pcie", which will also compile the software executables (ie calling "make pcieSWAuto

- Compile the software executables (make pcieSWAuto)
- Generate Verilog for the hardware accelerators
- Create the necessary software and hardware wrappers
- Link the software executables
- Generate the hardware system of accelerators, memory, and PCIe interface (make pcieVerilogbackend)
- Compile the Quartus project (make pcieQuartus)
- Program the Quartus project (make pcieProgram)

Once this is complete, you will need to restart your computer (ensure the DE4 remains powered on). Then, install the PCIe device driver (make pcieDriver) and then you should be able to run vector_add.elf and view the output.

7.2 Future work

We plan to support multiple threads by allowing each hardware accelerator to be duplicated, and each accessed by a single thread on the host processor program. We will add an extra configuration to specify the # of threads to match the number of generated hardware accelerators, and a method to specify a thread_id from the host processor so that it may be mapped to a unique hardware accelerator.

CHAPTER

EIGHT

LEGUP ON XILINX FPGAS

LegUp includes beta support for the Xilinx toolchain. At this time support is **very limited**, and includes only a subset of the full features in LegUp. Supported features include:

1. Support for generating Verilog that is compatible with the Xilinx tools.
2. Basic support for the Virtex 6 ML605 board.
3. Limited Makefile support targeting the Xilinx ISE toolchain.
4. Support only for the hardware-only flow (no processor support)
5. Using the Python debugger on the ML605 board (see *Debugging*).

Note: This is a bug in the LegUp 4.0 release (virtual machine and source tar) which prevents the Xilinx tools from loading memory initialization files. To fix this issue, please download the patch `here` and apply it using the following command:

```
legup@legup-vm:~/legup-4.0$ patch -p1 < ~/Downloads/legup-4.0-to-319115.patch
```

8.1 Options

To generate Verilog that is compatiable with Xilinx tools, the following options need to be set in a Tcl file:

1. RAM Format

   ```
   set_parameter INFERRED_RAM_FORMAT "xilinx"
   ```

This ensures that for inferred dual-ported RAMs, the behaviour of both ports of the RAM are defined in the same *always* block (versus Altera, where the ports are in separate *always* blocks)

2. Generic dividers

   ```
   set_parameter DIVIDER_MODULE "generic"
   ```

This uses generic divider modules, instead of the Altera primitives.

8.2 Makefile Support

To enable Makefile support for the Xilinx tool chain, the following needs to be added to the *Makefile* for the design:

```
XILINX = 1
```

Very basic Makefile support has been added, tested using ISE 14.7. To use these makefile targets, the ISE binaries will need to be included in the *PATH*. The following makefile targets are supported:

```
make
make p
make q
make f
```

8.3 Devices

Currently LegUp includes partial support for one Xilinx FPGA board, the Virtex-6 ML605 board. This board can be targeted by modifying the Tcl file to include:

```
set_project Virtex6 ML605 hw_only
```

Note: Device characterization has not been performed for this board. This includes delay information for basic operations (add, sub, multiply), which is used by the LegUp scheduler. The included delay values have been copied from another characterization file for a different FPGA. Thus, the generated circuit will not be optimally scheduled for this board. Better fmax may be obtained by updating the characterization information in:

```
<legup>/boards/Virtex6/ML605/ML605.tcl
```

To target a different Xilinx FPGA board, the user may either:

1. Modify LegUp to include support for another board (look at the ML605 files for reference)
2. Manually use the generated Verilog in a user created Xilinx project.

8.4 Examples

An example design with Xilinx ML605 support is included here:

```
<legup>/examples/xilinx/qsort
```

Note that the options listed above have been included in the *config.tcl* and *Makefile* files.

Further examples with debugging support enabled (see *Debugging*) are included here:

```
<legup>/examples/debug/xilinx
```

CHAPTER

NINE

FREQUENTLY ASKED QUESTIONS

How is LegUp different from other high-level synthesis (HLS) tools?

The source code is available for research purposes and we include a test suite to verify circuit correctness using simulations

Why should I use LegUp versus some other HLS tool? If you may have a need to modify or experiment with the underlying HLS algorithms, then you need access to the HLS tool source code. The reason you may need to modify such algorithms is: 1) you are an HLS researcher, or 2) you are curious how they work, or 3) you wish to build HLS support for a hardware platform that is presently unsupported by any existing commercial HLS tool

What is the latest version of the LegUp release? When did the project start? 4.0 is the latest release. We started the project at the University of Toronto in 2009 and made our first public release in 2011

Is LegUp free to use for commercial purposes? No, it is free only for research/academic purposes. Please see the license agreement when you download LegUp

Should I use the Virtual Machine or install LegUp from scratch? We highly recommend using the VM, as it is a turn-key solution that requires no messy library installation.

What are the goals of the LegUp project?

- To make FPGAs easier to program
- To help researchers develop new high-level synthesis algorithms

What is the input high-level lanagage?

ANSI C without recursive functions, or dynamic memory. Functions, arrays, structs, global variables, floating point, and pointers are supported

What is the output?

Verilog that can be simulated with ModelSim and synthesized using Altera Quartus II The synthesized circuits have been verified in hardware using an Altera DE2 (Cyclone II FPGA), Altera DE4 (Stratix IV FPGA), Altera DE5 (Stratis V FPGA), or the Altera DE1-SoC (Cyclone V-SoC FPGA).

Does LegUp support software/hardware partitioning?

Yes. We call this the LegUp hybrid flow. You can specify a list of functions to synthesize into hardware accelerators. The rest of the program is left running on a soft MIPS processor or ARM processor and hardware/software communication interfaces are generated automatically. The MIPS soft processor can be used on any of the supported FPGAs; the ARM processor is only avaialble on the Cyclone V-SoC FPGA (DE1-SoC board).

What high-level synthesis algorithms are supported?

- SDC scheduling with operator chaining and pipelined functional units
- Binding using bipartite weighted matching

- Pattern-based resource sharing
- Loop pipelining
- If-conversion (beta)

How are the quality of results?

- Hardware metrics are given on our quality of results page.
- We've found that the area-delay product over our benchmarks is compariable to eXCite, a commercial high-level synthesis tool.
- Quality of loop pipelining results are equal to (or better than) a state-of-the-art commercial tool (2014)

Do you support VHDL output?

No. We only support Verilog.

Do you support Xilinx FPGAs?

No, not officially. However, it is possible to generate Verilog in the pure hardware flow (no processor) that is generic and can be targeted to Xilinx FPGAs (at least for integer programs that require no floating point). See the Constraints Manual for how to configure LegUp to use generic dividers. However, bear in mind that LegUp uses delay estimation models during scheduling to make decisions about operator chaining. Presently, the estimation models are only developed for Altera devices.

Do you support having a NiosII or Microblaze processor?

No. Swapping the Tiger MIPS processor with a Microblaze/NiosII processor would be non-trivial.

How can I see the CDFG from legup? Can you display a gantt chart?

Yes! There is a "bare bones" GUI that you can use to see the CDFG and schedule in a Gantt-style form. To find out how to use the GUI, do Tutorial #1, which is found on the Tutorials section of the website.

Does legup support scheduling constraints? e.g., the number of operators, the time a certain operation should be used?

Yes, to some extent. You can control the number of specific types of each hardware resource to be used. See the constraints manual for how to use TCL constraints to do this.

How often do you release?

Roughly every year.

Why use the LLVM compiler infrastructure over GCC?

When we compared LLVM to GCC we found that the benefits outweighed the disadvantages.

GCC Pros:

- Mature and very popular
- Supports auto-vectorization
- Compiles faster code than LLVM (5-10%)
- Support for adding new optimization passes using a shared library (plug-in)

GCC Cons:

- Very little documentation
- Large complex C codebase with heavy use of globals and macros.
- Only have access to single static assignment form (GIMPLE) in the optimization phase

LLVM Pros:

- Great Documentation
- Used by Apple, NVIDIA, Xilinx, Altera, and others
- Very modular C++ design. Easy to add compiler passes and targets
- Code is very easy to work with and understand
- Access to SSA in every stage of the compiler
- Permissive BSD license
- Mature state-of-the-art compiler

Why did you write a new high-level synthesis tool when there are so many out there?

None of the existing high-level synthesis tools have source code available for researchers. GAUT claims to be open-source but the code is not available for download. xPilot from UCLA is an advanced research tool but only the binary is available and it hasn't been updated since 2007. ROCCC provides an open source eclipse plugin based on SUIF and LLVM but only supports small C programs. Standard C code must be rewritten to work with ROCCC because all function parameters must be structs. Trident uses a very old version of LLVM to interface with an extensive amount of Java code, but unfortunately no longer compiles with the latest version of LLVM.

CHAPTER

TEN

CONSTRAINTS MANUAL

High-level synthesis (HLS) tools typically accept user-provided constraints that impact the automatically generated hardware. This guide explains the constraints available for LegUp HLS.

Constraints are primarily represented in a TCL file(s) that are read by LegUp HLS on execution. The defaults are mainly to be found in `examples/legup.tcl`. While it is possible to directly modify `examples/legup.tcl`, for design-specific constraints, we recommend you create a `config.tcl` file in the local design directory that includes `examples/legup.tcl` then overrides default constraint values, as appropriate. See `examples/mult/config.tcl` for an example of the recommended methodology.

10.1 Most Commonly Used Constraints

10.1.1 CASE_FSM

This parameter controls whether the finite state machine (FSM) in the Verilog output by LegUp is implemented with a `case` statement or `if-else` statements. Although both options are functionally equivalent; some back-end RTL synthesis tools may be sensitive to the RTL coding style.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter CASE_FSM 1
```

10.1.2 CLOCK_PERIOD

This is a widely used constraint that allows the user to set the target clock period for a design. The clock period is specified in nanoseconds.

It has a significant impact on scheduling: the scheduler will schedule operators into clock cycles using delay estimates for each operator, such that the specified clock period is honored. In other words, operators will be chained together combinationally to the extent allowed by the value of the CLOCK_PERIOD parameter.

LegUp has a default CLOCK_PERIOD value for each device family that is supported. That default value was chosen to minimize the wall-clock time for a basket of benchmark programs (mainly the CHStone benchmark circuits).

If the parameter SDC_NO_CHAINING is 1, then the CLOCK_PERIOD parameter has no effect.

Category

HLS Constraints

Value Type

Integer represent a value in nanoseconds

Valid Values

Integer

Default Value

Depends on the target device

Location Where Default is Specified

`boards/CycloneII/CycloneII.tcl`

`boards/StratixIV/StratixIV.tcl`

and so on...

Dependencies

SDC_NO_CHAINING: If this parameter is set to 1, then CLOCK_PERIOD does nothing.

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter CLOCK_PERIOD 15
```

10.1.3 GROUP_RAMS

This parameter group all arrays in the global memory controller into four RAMs (one for each bitwidth: 8, 16, 32, 64). This saves M9K blocks by avoiding having a small array taking up an entire M9K block.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter GROUP_RAMS 1
```

10.1.4 GROUP_RAMS_SIMPLE_OFFSET

When GROUP_RAMS is on, this option simplifies the address offset calculation. Calculate the offset for each array into the shared RAM to minimize addition. The offset must be a multiple of the size of the array in bytes (to allow an OR instead of an ADD):

> before: addr = baseaddr + offset after: addr = baseaddr OR offset

the idea is that none of the lower bits of baseaddr should overlap with any bits of offset. This improves area and fmax (less adders) but at the cost of wasted empty memory inside the shared RAM

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter GROUP_RAMS_SIMPLE_OFFSET 1
```

10.1.5 LOCAL_RAMS

This parameter turns on alias analysis to determine when an array is only used in one function. These arrays can be placed in a block ram inside that hardware module instead of in global memory. This increases performance because local rams can be accessed in parallel while global memory is limited to two ports.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter LOCAL_RAMS 1
```

10.1.6 MB_MINIMIZE_HW

This parameter toggles whether the reduced bitwidths analyzed by the bitwidth minimization pass will be used in generating the Verilog design.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Related parameters: *MB_RANGE_FILE*, *MB_MAX_BACK_PASSES*, *MB_PRINT_STATS*

Applicable Flows

All devices and flows

Test Status

Prototype functionality

Examples

```
set_parameter MB_MINIMIZE_HW 1
```

10.1.7 NO_INLINE

This is a Makefile parameter that can disable the LLVM compiler from inlining functions. Note that all compiler optimizations will be turned off when `NO_INLINE` is enabled. This parameter can be set in `examples/Makefile.config` or in a local Makefile.

Category

LLVM

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
NO_INLINE=1
```

10.1.8 NO_OPT

This is a Makefile parameter that disables LLVM optimizations, which is equivalent to the `-O0` flag. This parameter can be set in `examples/Makefile.config` or in a local Makefile.

Category

LLVM

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
NO_OPT=1
```

10.1.9 set_accelerator_function

This sets the C function to be accelerated to HW in the hybrid flow. It can be used on one or more functions.

Category

HLS Constraints

Value Type

String

Valid Values

Name of the function

Default Value

NULL

Location Where Default is Specified

N/A

Dependencies

None

Applicable Flows

Hybrid flow

Test Status

Actively in-use

Examples

```
set_accelerator_function "add"
```

```
set_accelerator_function "div"
```

10.1.10 UNROLL

This is a Makefile parameter that allows user to specify additional flags related to the unroll transformation in LLVM compiler. This parameter can be set in `examples/Makefile.config` or in a local Makefile. Please see example settings in `examples/Makefile.config`.

Category

LLVM

Value Type

string

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
UNROLL = -unroll-allow-partial -unroll-threshold=1000
```

10.1.11 VSIM_NO_ASSERT

When set to 1, this constrain cause assertions to be disabled in the Verilog output used to debug LegUp. This is useful to speed-up long simulation.

Category

Simulation

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Untested

Examples

```
set_parameter VSIM_NO_ASSERT 1
```

10.1.12 loop_pipeline

This parameter enables pipelining for a given loop in the code. Loop pipelining allows a new iteration of the loop to begin before the current one has completed, achieving higher throughput. In a loop nest, only the innermost loop can be pipelined. Optional arguments:

Parameter	Description
-ii	Force a specific pipeline initiation interval
-ignore-mem-deps	Ignore loop carried dependencies for all memory accesses in the loop

Category

HLS Constraints

Value Type

Parameter	Value Type
loop_pipeline	String
-ii	Integer
-ignore-mem-deps	None

Valid Values

See Examples

Default Value

N/A

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
loop_pipeline "loop1"
```

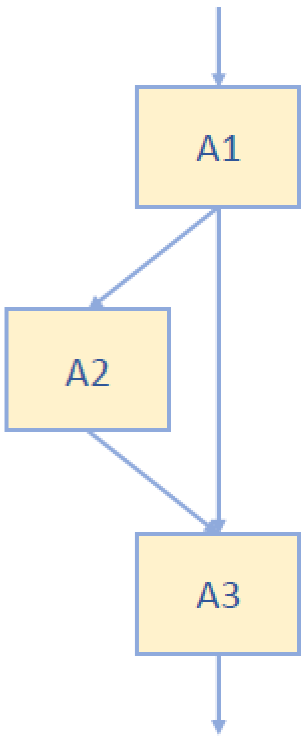
```
loop_pipeline "loop2" -ii 1
```

```
loop_pipeline "loop3" -ignore-mem-deps
```

10.1.13 set_combine_basicblock

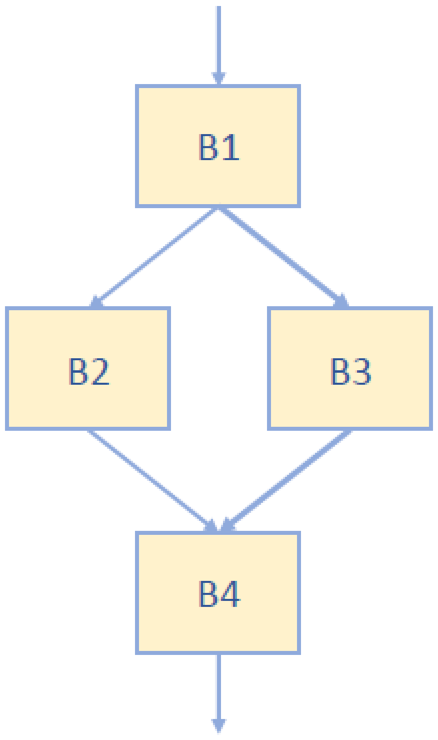
This parameter allows for basic block merging within the LLVM IR which potentially reduces the number of cycles of execution. There are two modes of operation: merge patterns throughout program, merge patterns only within loops. Currently, only 2 patterns are supported:

Pattern A:



A1, A2, A3 are basicblocks.

Pattern B:



B1, B2, B3, B4 are basicblocks.

Category

HLS Constraint

Value Type

Integer

Valid Values

0, 1, 2

Default Value

unset (off by default)

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

None

Note: May require *LOCAL_RAMS* and *GLOBAL_RAMS* to be turned off.

Applicable Flows

All devices for pure hardware flow

Test Status

Prototype functionality

Examples

```
set_combine_basicblock 1
```

10.1.14 set_operation_latency

This parameter sets the latency of a given operation when compiled in LegUp. Latency refers to the number of clock cycles required to complete the computation; an operation with latency one requires one cycle, while zero-latency operations are completely combinational, meaning multiple such operations can be chained together in a single clock cycle.

Category

HLS Constraints

Value Type

<operation> integer

Valid Values

See Default and Examples Note: operator name should match the device family operation database file: boards/StratixIV/StratixIV.tcl or boards/CycloneII/CycloneII.tcl

Default Values

```
altfp_add 14
altfp_subtract 14
altfp_multiply 11
altfp_divide_32 33
altfp_divide_64 61
altfp_truncate_64 3
altfp_extend_32 2
altfp_fptosi 6
altfp_sitofp 6
signed_comp_o 1
signed_comp_u 1
reg 2
mem_dual_port 2
local_mem_dual_port 1
multiply 1
```

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

`set_operation_latency altfp_add_32 18`

`set_operation_latency multiply 0`

10.1.15 set_project

This parameter sets the default target project, or device, used. Changing the project also updates the associated family and board parameters.

Category

Board and Device Specification

Value Type

String

Valid Values

See Examples

Default Value

`CycloneV DE1-SoC Tiger_SDRAM`

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_project CycloneV DE1-SoC Tiger_SDRAM
```

```
set_project CycloneV DE1-SoC ARM_Simple_Hybrid_System
```

```
set_project CycloneV SoCKit ARM_Simple_Hybrid_System
```

```
set_project CycloneIV DE2-115 Tiger_SDRAM
```

```
set_project CycloneII DE2 Tiger_SDRAM
```

```
set_project CycloneII CycloneIIAuto Tiger_SDRAM
```

```
set_project StratixV DE5-Net Tiger_DDR3
```

```
set_project StratixIV DE4-230 Tiger_DDR2
```

```
set_project StratixIV DE4-530 Tiger_DDR2
```

10.1.16 set_resource_constraint

This parameter constrains the number of times a given operation can occur in a cycle.

Note: A constraint on "signed_add" will apply to:

- signed_add_8
- signed_add_16
- signed_add_32
- signed_add_64
- unsigned_add_8
- unsigned_add_16
- unsigned_add_32
- unsigned_add_64

Category

HLS Constraints

Value Type

<operation> integer

Valid Values

See Default and Examples Note: operator name should match the device family operation database file: boards/StratixIV/StratixIV.tcl or boards/CycloneII/CycloneII.tcl

Default Values

```
mem_dual_port 2
divide 1
modulus 1
multiply 2
altfp_add 1
altfp_subtract 1
altfp_multiply 1
altfp_divide 1
altfp 1
```

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

`set_resource_constraint signed_divide_16 3`

`set_resource_constraint signed_divide 2`

`set_resource_constraint divide 1`

10.2 Other Advanced Constraints

10.2.1 CASEX

This parameter indicates whether `casex(...)` statement should be used instead of `case(...)` statement in the `case` style FSM verilog. This parameter is ignored when `CASE_FSM` is disabled.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

CASE_FSM

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter CASEX 1
```

10.2.2 DEBUG_DB_HOST

Hostname for MySQL debug database. Used by both Inspect and Python Debugger.

Category

Debugging

Value Type

String

Default Value

localhost

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

INSPECT_DEBUG for Inspect debugger. None for Python debugger.

Related parameters: *DEBUG_DB_USER*, *DEBUG_DB_PASSWORD* For Python debugger: *DEBUG_DB_NAME*, *DEBUG_DB_SCRIPT_FILE* For Inspect debugger: *INSPECT_DEBUG_DB_NAME*, *INSPECT_DEBUG_DB_SCRIPT_FILE*

Applicable Flows

Pure hardware, Inspect debugger for pure hardware

Examples

```
set_parameter DEBUG_DB_HOST localhost
```

10.2.3 DEBUG_DB_NAME

MySQL debug database name. Used by Python debugger.

Category

Debugging

Value Type

String

Default Value

legupDebug

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

Related parameters: *DEBUG_DB_HOST*, *DEBUG_DB_USER*, *DEBUG_DB_PASSWORD*, *DEBUG_DB_SCRIPT_FILE*

Applicable Flows

Pure hardware

Test Status

Prototype functionality

Examples

```
set_parameter DEBUG_DB_NAME legupDebug
```

10.2.4 DEBUG_DB_PASSWORD

Password for MySQL debug database. Used by both Inspect and Python Debugger.

Category

Debugging

Value Type

String

Default Value

letmein

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

INSPECT_DEBUG for Inspect debugger. None for Python debugger.

Related parameters: *DEBUG_DB_HOST*, *DEBUG_DB_USER* For Python debugger: *DEBUG_DB_NAME*, *DEBUG_DB_SCRIPT_FILE* For Inspect debugger: *INSPECT_DEBUG_DB_NAME*, *INSPECT_DEBUG_DB_SCRIPT_FILE*

Applicable Flows

Pure hardware, Inspect debugger for pure hardware

Examples

```
set_parameter DEBUG_DB_PASSWORD letmein
```

10.2.5 DEBUG_DB_SCRIPT_FILE

Path to script file for creating MySQL debug database used by Python debugger.

Category

Debugging

Value Type

String

Default Value

examples/createDebugDB.sql

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

Related parameters: *DEBUG_DB_HOST*, *DEBUG_DB_USER*, *DEBUG_DB_PASSWORD*, *DEBUG_DB_NAME*

Applicable Flows

Pure hardware

Test Status

Prototype functionality

Examples

```
set_parameter DEBUG_DB_SCRIPT_FILE $::CURRENT_PATH/createDebugDB.sql
```

10.2.6 DEBUG_DB_USER

Username for MySQL debug database. Used by both Inspect and Python Debugger.

Category

Debugging

Value Type

String

Default Value

root

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

INSPECT_DEBUG for Inspect debugger. None for Python debugger.

Related parameters: *DEBUG_DB_HOST*, *DEBUG_DB_PASSWORD* For Python debugger: *DEBUG_DB_NAME*, *DEBUG_DB_SCRIPT_FILE* For Inspect debugger: *INSPECT_DEBUG_DB_NAME*, *INSPECT_DEBUG_DB_SCRIPT_FILE*

Applicable Flows

Pure hardware, Inspect debugger for pure hardware

Examples

```
set_parameter DEBUG_DB_USER root
```

10.2.7 DEBUG_SDC_CONSTRAINTS

This parameter prints all the SDC constraints during modulo scheduling into the pipelining.legup.rpt file.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter DEBUG_SDC_CONSTRAINTS 1
```

10.2.8 DFG_SHOW_DUMMY_CALLS

Show dummy calls such as printf in the dataflow dot graphs.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0, dummy calls ignored)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

NO_DFG_DOT_FILES should be disabled

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
    set_parameter DFG_SHOW_DUMMY_CALLS 1
```

10.2.9 DISABLE_REG_SHARING

Disables register sharing based on live variable analysis.

Category

HLS Constraint

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0 - disabled)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter DISABLE_REG_SHARING 1
```

10.2.10 DONT_CHAIN_GET_ELEM_PTR

By default, LegUp assumes that get element pointer instructions take zero time in hardware and will therefore chain consecutive get element pointer instructions into a single cycle. This default behaviour can be disabled by setting the DONT_CHAIN_GET_ELEM_PTR constraint to zero. This will split dependent get element pointer instructions into seperate clock cycles.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter DONT_CHAIN_GET_ELEM_PTR 0
```

10.2.11 DUAL_PORT_BINDING

Enabling this parameter results in use of the dual-ported on-chip memories, allowing up to two memory accesses per clock cycle.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1 (On)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Example

```
set_parameter DUAL_PORT_BINDING 1
```

10.2.12 ENABLE_PATTERN_SHARING

This parameter enables resource sharing for patterns of computational operators in a program's dataflow graph. The idea is that, in a given program, there may be commonly occurring patterns of operators that could be shared in the hardware, by putting multiplexers on the inputs and steering the right data in at the right time (based on the FSM state). This may save area in certain cases. The approach is described in this paper:

Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter ENABLE_PATTERN_SHARING 1
```

10.2.13 EXPLICIT_LPM_MULTS

This parameter explicitly instantiate all multipliers as Altera lpm_mult modules. When this is off (default) multiplies are instantiated using the Verilog multiply operator.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter EXPLICIT_LPM_MULTS 1
```

10.2.14 FREQ_THRESHOLD

Minimum pattern frequency written to dot/v file

Category

Miscellaneous

Value Type

Integer

Valid Values

Positive integer

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

At least one of these parameters: *PS_WRITE_TO_DOT* or *PS_WRITE_TO_VERILOG* needs to be enabled.

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
    set_parameter FREQ_THRESHOLD 1
```

10.2.15 INCLUDE_INST_IN_FSM_DOT_GRAPH

This constraint enables including instruction labels in the state node labels in the FSM's generated dot graph.

Category

Miscellaneous

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Untested

Examples

```
set_parameter INCLUDE_INST_IN_FSM_DOT_GRAPH 1
```

10.2.16 INCREMENTAL_SDC

This parameter solve the SDC problem during loop pipelining incrementally by detecting negative cycles in the constraint graph. This is marginally faster than the default method of rerunning the LP solver.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter INCREMENTAL_SDC 1
```

10.2.17 INFERRED_RAMS

Use Verilog to infer RAMs instead of instantiating an Altera altsyncram module. Note: inferred RAMs don't support structs (no byte-enable) so having structs in your program forces this parameter to turn off

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter INFERRED_RAMS 1
```

10.2.18 INSPECT_DEBUG

Enables Inspect Debugger functionality and populates database. Creates full debugging information for hardware during code generation for the Inspect Debugger.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Related parameters: *DEBUG_DB_HOST*, *DEBUG_DB_USER*, *DEBUG_DB_PASSWORD*, *INSPECT_DEBUG_DB_NAME*, *INSPECT_DEBUG_DB_SCRIPT_FILE*

Applicable Flows

Inspect debugger for pure hardware

Test Status

Prototype functionality

Examples

```
set_parameter INSPECT_DEBUG 1
```

10.2.19 INSPECT_DEBUG_DB_NAME

MySQL debug database name. Used by Inspect debugger.

Category

Debugging

Value Type

String

Default Value

inspect_db

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

INSPECT_DEBUG

Related parameters: *INSPECT_ONCHIP_BUG_DETECT_DEBUG*, *DEBUG_DB_HOST*, *DEBUG_DB_USER*, *DEBUG_DB_PASSWORD*, *INSPECT_DEBUG_DB_SCRIPT_FILE*

Applicable Flows

Inspect debugger for pure hardware

Test Status

Prototype functionality

Examples

```
set_parameter INSPECT_DEBUG_DB_NAME inspect_db
```

10.2.20 INSPECT_DEBUG_DB_SCRIPT_FILE

Path to script file for creating MySQL debug database used by Inspect debugger.

Category

Debugging

Value Type

String

Default Value

examples/inspect_db.sql

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

INSPECT_DEBUG

Related parameters: *INSPECT_ONCHIP_BUG_DETECT_DEBUG*, *DEBUG_DB_HOST*, *DEBUG_DB_USER*, *DEBUG_DB_PASSWORD*, *INSPECT_DEBUG_DB_NAME*

Applicable Flows

Inspect debugger for pure hardware

Test Status

Prototype functionality

Examples

```
set_parameter INSPECT_DEBUG_DB_SCRIPT_FILE $::CURRENT_PATH/inspect_db.sql
```

10.2.21 INSPECT_ONCHIP_BUG_DETECT_DEBUG

Creates partial debugging information for Inspect Debugger. This parameter should only be set when Inspect ON_CHIP_BUG_DETECT mode is required. Requires INSPECT_DEBUG parameter to be enabled.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

INSPECT_DEBUG

Applicable Flows

Inspect debugger for pure hardware

Test Status

Prototype functionality

Examples

```
set_parameter INSPECT_ONCHIP_BUG_DETECT_DEBUG 1
```

10.2.22 KEEP_SIGNALS_WITH_NO_FANOUT

If this parameter is enabled, all signals will be printed to the output Verilog file, even if they don't drive any outputs.

Category

HLS Constraint

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter KEEP_SIGNALS_WITH_NO_FANOUT 1
```

10.2.23 LLVM_PROFILE

Enabling this parameter will allow the scheduler to modify scheduling based on the llvm profiling information. To enable this option, `LLVM_PROFILE=1` should be set in `Makefile.config` and `set_parameter LLVM_PROFILE 1` should be set in `config.tcl` or `legup.tcl`.

With `LLVM_PROFILE` enabled, the schedules will be modified for the infrequently used basic blocks, which the basic blocks executed at or below the frequency threshold `LLVM_PROFILE_MAX_BB_FREQ_TO_ALTER`. This value can be set via

```
set_parameter LLVM_PROFILE_MAX_BB_FREQ_TO_ALTER $(USER_DEFINED_FREQUENCY).
```

Two options are provided to further specify how the scheduling should be modified. The first option is to extend paths in infrequently executed BBs by a fixed number of states, which can be set with

```
set_parameter LLVM_PROFILE_EXTRA_CYCLES $(USER_DEFINED_CYCLES).
```

Note that this change can only be applied to multi-cycle paths thus the parameter `MULTI_CYCLE_REMOVE_REG` must be set.

The second option is to adjust the target period for the infrequently executed basic blocks. As an alternative to `LLVM_PROFILE_EXTRA_CYCLES` which simply extends paths in infrequent basic blocks by a fixed amount, `LLVM_PROFILE_PERIOD_<DEVICE>` can be used to change the target period for paths in infrequent blocks (e.g. a 13ns constraint on Cyclone II might be the best, but infrequent blocks can use a 6ns constraint instead). This target frequency parameter is set via a FPGA device-dependent parameter, either `LLVM_PROFILE_PERIOD_CII` (for Cyclone II) or `LLVM_PROFILE_PERIOD_SIV` (for Stratix IV).

Category

HLS Constraints.

Value Type

Integer

Valid Values

`LLVM_PROFILE`: 0, 1
`LLVM_PROFILE_MAX_BB_FREQ_TO_ALTER`: positive integers
`LLVM_PROFILE_EXTRA_CYCLES`: positive integers
`LLVM_PROFILE_PERIOD_CII`: positive integers
`LLVM_PROFILE_PERIOD_SIV`: positive integers

Default Value

0 for LLVM_PROFILE.

Location Where Default is Specified

`examples/legup.tcl` `examples/Makefile.config`

Dependencies

`MULTI_CYCLE_REMOVE_REG`

Applicable Flows

All devices and flows

Test Status

Prototype functionality

Examples

Enable scheduling modification based on profiling information, `set_parameter PRINT_BB_STATS LLVM_PROFILE` in `config.tcl` `LLVM_PROFILE=1` in `Makefile.config`

Specify the maximum execution frequency for infrequent basic blocks, `set LLVM_PROFILE_MAX_BB_FREQ_TO_ALTER 100`

With `set_parameter MULTI_CYCLE_REMOVE_REG 1` enabled, paths in infrequent executed basic block may be,

1. extended for 2 cycles,

```
set LVM_PROFILE_EXTRA_CYCLES 2
```

2) or be extended to match a more constrainted clock period, say 13ns for Cyclone II or 6ns for Stratix IV.

```
set LLVM_PROFILE_PERIOD_CII 13
set LLVM_PROFILE_PERIOD_SIV 6
```

10.2.24 MB_MAX_BACK_PASSES

The bitwidth minimization pass repeatedly traverses the CDFG to reduce the number of bits used in each variable. Every forward traversal of the CDFG is followed by a backward traversal. This parameter specifies the maximum number of times the analysis should traverse backwards on the CDFG.

Setting this parameter to -1 instructs the analysis to continually traverse the CDFG until no further bitwidth reduction can be achieved.

Category

HLS Constraints

Value Type

Integer

Valid Values

-1, 0 to INT_MAX

Default Value

-1

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

Need to set *MB_MINIMIZE_HW* to 1 for this parameter to take effect on the generated Verilog.

Related parameters: *MB_RANGE_FILE*, :ref: *MB_PRINT_STATS*

Applicable Flows

All devices and flows

Test Status

Prototype functionality

Examples

```
set_parameter MB_MAX_BACK_PASSES -1
```

10.2.25 MB_PRINT_STATS

This parameter toggles whether the bitiwdth minimization pass prints a report showing the number of bits reduced. The reported numbers are an over-estimation of the savings since Quartus performs some bitwidth optimization internally already.

Since the bitwidth minimization anlysis is always performed, this report reflects the actual number of bits saved only if *MB_MINIMIZE_HW* is turned on.

Category

Miscellaneous

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

Need to set *MB_MINIMIZE_HW* to 1 for the report to reflect the actual number of bits saved.

Related parameters: *MB_RANGE_FILE*, :ref: *MB_MAX_BACK_PASSES*

Applicable Flows

All devices and flows

Test Status

Prototype functionality

Examples

```
set_parameter MB_PRINT_STATS 1
```

10.2.26 MB_RANGE_FILE

The bitwidth minimization pass in LegUp is capable of reading in data value ranges based on profiling results. This parameter specifies the filename from which to read the initial data ranges used in the analysis. If this parameter is not set, no initial ranges are assumed.

Category

HLS Constraints

Value Type

String

Valid Values

Alphanumeric file name

Default Value

N/A

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

Need to set *MB_MINIMIZE_HW* to 1 for this to take effect

Related parameters: *MB_PRINT_STATS*, :ref: *MB_MAX_BACK_PASSES*

Applicable Flows

All devices and flows

Test Status

Prototype functionality

Examples

```
set_parameter MB_RANGE_FILE "range.profile"
```

10.2.27 MODULO_DEBUG

This parameter show some high level debugging information from the loop pipelining (modulo) scheduler. For instance, the final initiation interval (II) and how many backtracking attempts were made.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter MODULO_DEBUG 1
```

10.2.28 MODULO_SCHEDULER

This parameter specifies the type of modulo scheduler to use for loop pipelining. There are three options:

SDC_BACKTRACKING Default. Used SDC modulo scheduling with a backtracking mechanism to resolve conflicting resource and recurrence constraints

SDC_GREEDY SDC modulo scheduling using a greedy approach to resolving resource constraints. This method may not achieve minimum II II for loops with resource constraints and cross-iteration dependencies.

ITERATIVE Classic iterative modulo scheduler approach using a list scheduler (no operator chaining support)

Category

HLS Constraints

Value Type

String

Valid Values

SDC_BACKTRACKING, SDC_GREEDY, ITERATIVE

Default Value

SDC_BACKTRACKING

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

`set_parameter MODULO_SCHEDULER SDC_BACKTRACKING` `set_parameter MODULO_SCHEDULER SDC_GREEDY` `set_parameter MODULO_SCHEDULER ITERATIVE`

10.2.29 MULTIPLIER_NO_CHAIN

This parameter tells the LegUp scheduler not to chain multipliers. If this parameter is on then every multiplier will be scheduled into a separate clock cycle.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter MULTIPLIER_NO_CHAIN 1
```

10.2.30 MULTIPUMPING

This parameter controls whether LegUp multi-pumps multipliers to save DSP blocks and achieve better performance. For details see: Andrew Canis, Stephen D. Brown, and Jason H. Anderson, "Multi-Pumping for Resource Reduction in FPGA High-Level Synthesis," Design, Automation, and Test in Europe (DATE). Grenoble, France, March, 2013. (PDF)

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use (examples/multipump)

Examples

```
set_parameter MULTIPUMPING 1
```

10.2.31 MULTI_CYCLE_ADD_THROUGH_CONSTRAINTS

This parameter allows LegUp to use -through constraints to create different multi-cycle slack on different multi-cycle paths that have different latencies but the same source and destination register.

Category

Quartus

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

MULTI_CYCLE_REMOVE_REG

Related parameters: *MULTI_CYCLE_DEBUG*, *MULTI_CYCLE_REMOVE_REG_DIVIDERS*, *MULTI_CYCLE_DUPLICATE_LOAD_REG*, *MULTI_CYCLE_REMOVE_CMP_REG*, *MULTI_CYCLE_DISABLE_REG_MERGING*

Applicable Flows

Pure hardware flow with no multipumping and no loop pipelining.

Test Status

Actively in-use

Examples

```
set_parameter MULTI_CYCLE_ADD_THROUGH_CONSTRAINTS 1
```

10.2.32 MULTI_CYCLE_DEBUG

Enabling this parameter prints out useful debugging messages for multi-cycling.

Category

Miscellaneous

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

MULTI_CYCLE_REMOVE_REG

Related parameters: *MULTI_CYCLE_ADD_THROUGH_CONSTRAINTS*, *MULTI_CYCLE_REMOVE_REG_DIVIDERS*, *MULTI_CYCLE_DUPLICATE_LOAD_REG*, *MULTI_CYCLE_REMOVE_CMP_REG*, *MULTI_CYCLE_DISABLE_REG_MERGING*

Applicable Flows

Pure hardware flow with no multipumping and no loop pipelining.

Test Status

Actively in-use

Examples

```
set_parameter MULTI_CYCLE_DEBUG 1
```

10.2.33 MULTI_CYCLE_DISABLE_REG_MERGING

Enabling this will print qsf constraints to instruct Quartus synthesis to not merge registers that have multi-cycle constraints.

Category

Quartus

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

MULTI_CYCLE_REMOVE_REG

Related parameters: *MULTI_CYCLE_ADD_THROUGH_CONSTRAINTS*, *MULTI_CYCLE_DEBUG*, *MULTI_CYCLE_REMOVE_REG_DIVIDERS*, *MULTI_CYCLE_DUPLICATE_LOAD_REG*, *MULTI_CYCLE_REMOVE_CMP_REG*

Applicable Flows

Pure hardware flow with no multipumping and no loop pipelining.

Test Status

Actively in-use

Examples

```
set_parameter MULTI_CYCLE_DISABLE_REG_MERGING 1
```

10.2.34 MULTI_CYCLE_DUPLICATE_LOAD_REG

Enabling this parameter will duplicate load registers. This is used to allow loads from global memory to be incorporated into multi-cycle paths.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

MULTI_CYCLE_REMOVE_REG

Related parameters: *MULTI_CYCLE_ADD_THROUGH_CONSTRAINTS*, *MULTI_CYCLE_DEBUG*, i *MULTI_CYCLE_REMOVE_REG_DIVIDERS*, *MULTI_CYCLE_REMOVE_CMP_REG*, *MULTI_CYCLE_DISABLE_REG_MERGING*

Applicable Flows

Pure hardware flow with no multipumping and no loop pipelining.

Test Status

Actively in-use

Examples

```
set_parameter MULTI_CYCLE_DUPLICATE_LOAD_REG 1
```

10.2.35 MULTI_CYCLE_REMOVE_CMP_REG

Setting this to 1 removes the unnecessary output register associated with compare instructions.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

MULTI_CYCLE_REMOVE_REG

Related parameters: *MULTI_CYCLE_ADD_THROUGH_CONSTRAINTS*, *MULTI_CYCLE_DEBUG*, *MULTI_CYCLE_REMOVE_REG_DIVIDERS*, *MULTI_CYCLE_DUPLICATE_LOAD_REG*, *MULTI_CYCLE_DISABLE_REG_MERGING*

Applicable Flows

Pure hardware flow with no multipumping and no loop pipelining.

Test Status

Actively in-use

Examples

```
set_parameter MULTI_CYCLE_REMOVE_REG 1
```

10.2.36 MULTI_CYCLE_REMOVE_REG

This parameter instructs LegUp to multi-cycle combinational data paths in infrequently used basic blocks, in an attempt to increase the circuit's FMax. Doing this will remove registers on the multi-cycle paths and generate multi-cycle sdc constraints.

Multi-cycle paths are only enabled for the pure hardware flow with no multipumping and no loop pipelining.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

None

Related parameters: *MULTI_CYCLE_ADD_THROUGH_CONSTRAINTS*, *MULTI_CYCLE_DEBUG*, *MULTI_CYCLE_REMOVE_REG_DIVIDERS*, *MULTI_CYCLE_DUPLICATE_LOAD_REG*, *MULTI_CYCLE_REMOVE_CMP_REG*, *MULTI_CYCLE_DISABLE_REG_MERGING*

Applicable Flows

Pure hardware flow with no multipumping and no loop pipelining.

Test Status

Actively in-use

Examples

```
set_parameter MULTI_CYCLE_REMOVE_REG 1
```

10.2.37 MULTI_CYCLE_REMOVE_REG_DIVIDERS

Enabling this parameter will multi-cycle dividers. This requires *MULTI_CYCLE_REMOVE_REG* to be set to 1.

Multi-cycle paths are only enabled for the pure hardware flow with no multipumping and no pipelined resources.

Category

Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

MULTI_CYCLE_REMOVE_REG

Related parameters: *MB_RANGE_FILE*, *MB_MAX_BACK_PASSES*, *MB_PRINT_STATS*

Applicable Flows

Pure hardware flow with no multipumping and no pipelined resources.

Test Status

Actively in-use

Examples

```
set_parameter MULTI_CYCLE_REMOVE_REG_DIVIDERS 1
```

10.2.38 MULT_BY_CONST_INFER_DSP

This parameter assumes that all

LegUp detects whether multiply-by-constant operations will infer DSPs. For instance, x * 10 = x * (2^3 + 2) = (x << 3) + (x << 1) Therefore, the final circuit will not require a DSP block for this multiplier.

In general, Quartus will not infer if a multiply by constant can be replaced by shifts (which are free) and at most one addition to see if this is possible: 1) calculate the closest power of two 2) can get there by adding/subtracting x or (x << n)?

This impacts binding because we don't want to share a multiply operation that requires DSP blocks with a multiply that can be done with shifts/adds.

This parameter turns off this detection and assumes at all multiply-by-constant operations will infer DSP blocks.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Untested

Examples

```
set_parameter MULT_BY_CONST_INFER_DSP 1
```

10.2.39 NO_DFG_DOT_FILES

By default, LegUp generates data flow graph dot files for every basic block. This constraint can disable the generation of those dot files.

Category

Debugging

Value Type

String

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Untested

Examples

```
set_parameter NO_DFG_DOT_FILES 1
```

10.2.40 NO_LOOP_PIPELINING

This parameter turns off loop pipelining.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter NO_LOOP_PIPELINING 1
```

10.2.41 NO_ROMS

This parameter places constant arrays into RAMs instead of the default read-only memory.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter NO_ROMS 1
```

10.2.42 PATTERN_SHARE_ADD

Select which instructions to share in pattern sharing. Choices are: Adders / Subtractors Bitwise operations (AND, OR, XOR) Shifts (logical shift Left/Right and arithmetic shift Right)

If set, these instructions will be included in patterns and shared with 2-1 muxing. Note that multipliers, dividers and remainders are not shared in patterns because they should be shared with more than 2-1 muxing (if at all). The bipartite binding algorithm is used for those instructions while pattern sharing is used for the smaller instructions above.

This particular parameter pertains to sharing add.

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

0 or 1

Default Value

0 (for Cyclone II); 1 (for all other archs with 6-LUTs)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PATTERN_SHARE_ADD 1’‘

10.2.43 PATTERN_SHARE_BITOPS

Select which instructions to share in pattern sharing. Choices are: Adders / Subtractors Bitwise operations (AND, OR, XOR) Shifts (logical shift Left/Right and arithmetic shift Right)

If set, these instructions will be included in patterns and shared with 2-1 muxing. Note that multipliers, dividers and remainders are not shared in patterns because they should be shared with more than 2-1 muxing (if at all). The bipartite binding algorithm is used for those instructions while pattern sharing is used for the smaller instructions above.

This particular parameter pertains to sharing bitwise operators (XOR, AND, etc)

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, “Impact of FPGA architecture on resource sharing in high-level synthesis,” FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

0 or 1

Default Value

0 (for Cyclone II); 1 (for all other archs with 6-LUTs)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PATTERN_SHARE_BITOPS 1’‘

10.2.44 PATTERN_SHARE_SHIFT

Select which instructions to share in pattern sharing. Choices are: Adders / Subtractors Bitwise operations (AND, OR, XOR) Shifts (logical shift Left/Right and arithmetic shift Right)

If set, these instructions will be included in patterns and shared with 2-1 muxing. Note that multipliers, dividers and remainders are not shared in patterns because they should be shared with more than 2-1 muxing (if at all). The bipartite binding algorithm is used for those instructions while pattern sharing is used for the smaller instructions above.

This particular parameter pertains to sharing shifts.

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, “Impact of FPGA architecture on resource sharing in high-level synthesis,” FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

0 or 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PATTERN_SHARE_SHIFT 1’‘

10.2.45 PATTERN_SHARE_SUB

Select which instructions to share in pattern sharing. Choices are: Adders / Subtractors Bitwise operations (AND, OR, XOR) Shifts (logical shift Left/Right and arithmetic shift Right)

If set, these instructions will be included in patterns and shared with 2-1 muxing. Note that multipliers, dividers and remainders are not shared in patterns because they should be shared with more than 2-1 muxing (if at all). The bipartite binding algorithm is used for those instructions while pattern sharing is used for the smaller instructions above.

This particular parameter pertains to sharing subtract.

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, “Impact of FPGA architecture on resource sharing in high-level synthesis,” FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

0 or 1

Default Value

0 (for Cyclone II); 1 (for all other archs with 6-LUTs)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PATTERN_SHARE_SUB 1’‘

10.2.46 PIPELINE_ALL

This parameter tells LegUp to try to pipeline every loop in the program regardless of the loop label. Loops will only be pipelined if they are only one basic block.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
    set_parameter PIPELINE_ALL 1
```

10.2.47 PIPELINE_RESOURCE_SHARING

This parameter turns on resource sharing inside a loop pipeline.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter PIPELINE_RESOURCE_SHARING 1
```

10.2.48 PIPELINE_SAVE_REG

This parameter saves registers in the loop pipeline by only using registers on the pipeline stage boundaries. Instead of using a register for every single pipeline time step.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter PIPELINE_SAVE_REG 1
```

10.2.49 PRINTF_CYCLES

Enabling this parameter will result in prepending each printf function with the number of cycles elapsed to be displayed in ModelSim simulations.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter PRINTF_CYCLES 1
```

10.2.50 PRINT_BB_STATS

Enabling this parameter will result in printing of statistics on the number of instructions, memory instructions and basic blocks in each loop or function.

Category

Miscellaneous

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter PRINT_BB_STATS 1
```

10.2.51 PRINT_FUNCTION_START_FINISH

When this parameter is enabled, additional simulation messages will be displayed to show the functions' starting cycles and finish cycles.

Category

Simulation

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter PRINT_FUNCTION_START_FINISH 1
```

10.2.52 PRINT_STATES

When this parameter is enabled, the cur_state of each function will be displayed at every clock cycle in simulation.

Category

Simulation

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter PRINT_STATES 1
```

10.2.53 PS_BIT_DIFF_THRESHOLD

Two operations will only be shared if the difference of their true bit widths is below this threshold: e.g. an 8-bit adder will not be shared with a 32-bit adder unless BIT_DIFF_THRESHOLD >= 24

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

Any integer

Default Value

10

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PS_BIT_DIFF_THRESHOLD 10’‘

10.2.54 PS_MAX_SIZE

Maximum pattern size to share. Setting to 0 will also disable pattern sharing. Setting this to a high value may incur significant runtime.

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, “Impact of FPGA architecture on resource sharing in high-level synthesis,” FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

Any integer

Default Value

10

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PS_MAX_SIZE 10’‘

10.2.55 PS_MIN_SIZE

Minimum pattern size to share. This is used because sharing is more beneficial for larger patterns (larger patterns have a smaller mux:instruction ratio) and sometimes sharing is only beneficial for patterns of a certain size or greater. For example, in Cyclone II sharing small patterns (e.g. 2 adds) does not reduce area.

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, “Impact of FPGA architecture on resource sharing in high-level synthesis,” FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

Any integer

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PS_MIN_SIZE 1’‘

10.2.56 PS_MIN_WIDTH

The minimum bit width of an instruction to consider (e.g. don’t bother sharing 1 bit adders)

This parameter does NOTHING unless ENABLE_PATTERN_SHARING is 1.

Paper: Stefan Hadjis, Andrew Canis, Jason Helge Anderson, Jongsok Choi, Kevin Nam, Stephen Dean Brown, Tomasz S. Czajkowski, “Impact of FPGA architecture on resource sharing in high-level synthesis,” FPGA 2012: 111-114

Category

HLS Constraints

Value Type

Integer

Valid Values

Any integer

Default Value

2

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

ENABLE_PATTERN_SHARING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

> ‘‘set_parameter PS_MIN_WIDTH 2’‘

10.2.57 RESTRICT_TO_MAXDSP

This parameter turns on multiplier sharing (binding) when the number of multipliers required by the design is going to exceed the maximum DSP blocks available on the target FPGA. Turning this on prevents the final circuit from implementing multiplication with soft logic (high area cost) after all DSP blocks are used, instead sharing the available DSP blocks using multiplexing.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Untested recently

Examples

```
set_parameter RESTRICT_TO_MAXDSP 1
```

10.2.58 SDC_BACKTRACKING_PRIORITY

The same as SDC_PRIORITY but for backtracking SDC modulo scheduling. A priority function isn't strictly necessary due to backtracking but a decent scheduling order can really speed up scheduling by reducing the amount of backtracking. Without this on you might have to increase the SDC_BACKTRACKING_BUDGET_RATIO parameter to allow more backtracking.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter SDC_BACKTRACKING_PRIORITY 1
```

10.2.59 SDC_MULTIPUMP

This parameter schedules more multipliers into the same state for multi-pumping. If two multipliers are scheduled into the same state then we can replace them with a multi-pumped DSP and save DSP blocks (with multi-pumping on)

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

MULTIPUMPING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter SDC_MULTIPUMP 1
```

10.2.60 SDC_NO_CHAINING

This is an HLS scheduling constraint that can have a significant impact on speed performance. Chaining is a concept in HLS that allows operators to be stitched together combinationally in a single clock cycle, provided a target clock period constraint is met. When this parameter is set to 1, chaining is disabled and every operator will be scheduled in its own FSM state. This will generally increase the number of cycles in the overall schedule (bad), but it may help the circuit Fmax (good).

When this parameter is set to 1, then the CLOCK_PERIOD constraint is irrelevant. The reason for this is that the CLOCK_PERIOD constraint aims to allow chaining to the extent possible such that the specified period is met.

In general, this parameter should remain 0; however, there may be some utility in setting it to 1 for research purposes.

Category

HLS Constraints

Value Type

Integer

Valid Values

0 or 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter SDC_NO_CHAINING 1
```

10.2.61 SDC_ONLY_CHAIN_CRITICAL

By default during modulo scheduling we allow chaining to occur anywhere. This parameter turns off chaining for any operation not on a loop recurrence for maximum pipelining while still not impacting II. This parameter only works when SDC_BACKTRACKING is on.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

0

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

set_parameter MODULO_SCHEDULER SDC_BACKTRACKING

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter SDC_ONLY_CHAIN_CRITICAL 1
```

10.2.62 SDC_PRIORITY

In greedy SDC modulo scheduling the order of scheduling instructions uses a priority function based on perturbation. Perturbation is calculated for each instruction by:

1. adding a GE constraint to the SDC formulation for that instruction
2. resolving the SDC schedule
3. counting how many many other instructions are displaced from their prior schedule

Instruction with higher perturbation are scheduled with higher priority. If this is off, instructions are scheduled randomly.

Category

HLS Constraints

Value Type

Integer

Valid Values

0, 1

Default Value

1

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter SDC_PRIORITY 1
```

10.2.63 SYSTEM_BARRIER_BASE_ADDRESS

This parameter specifies the base address of the barrier in the QSys system.

Category

Board and Device Specification

Value Type

Integer

Valid Values

Any valid base address.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Applicable Flows

Software and hybrid flows

Test Status

Not used

Examples

`set_parameter SYSTEM_BARRIER_BASE_ADDRESS 0x00800000`

10.2.64 SYSTEM_CLOCK_INTERFACE

This parameter specifies the 'Clock Output' interface of the QSys module that provides the system clock.

Category

Board and Device Specification

Value Type

String

Valid Values

The 'Clock Output' interface of the QSys module that supplies the system clock.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_CLOCK_MODULE*

Applicable Flows

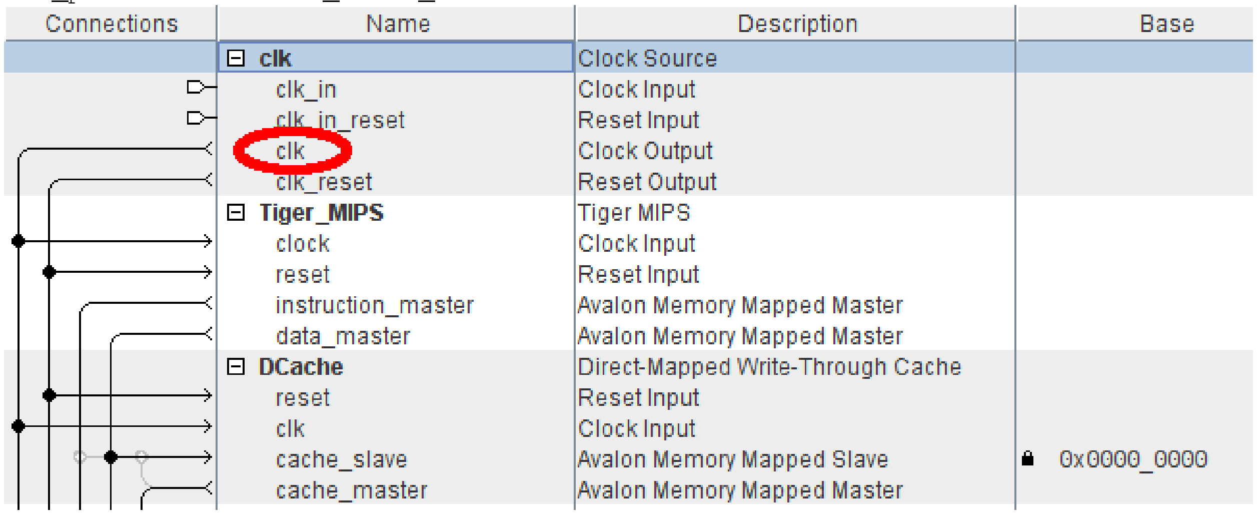
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_CLOCK_INTERFACE clk
```



Connections	Name	Description	Base
	⊟ **clk**	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ **Tiger_MIPS**	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	⊟ **DCache**	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.65 SYSTEM_CLOCK_MODULE

This parameter specifies the QSys module that supplies the system clock.

Category

Board and Device Specification

Value Type

String

Valid Values

The name of the QSys module that supplies the system clock.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_CLOCK_INTERFACE*

Applicable Flows

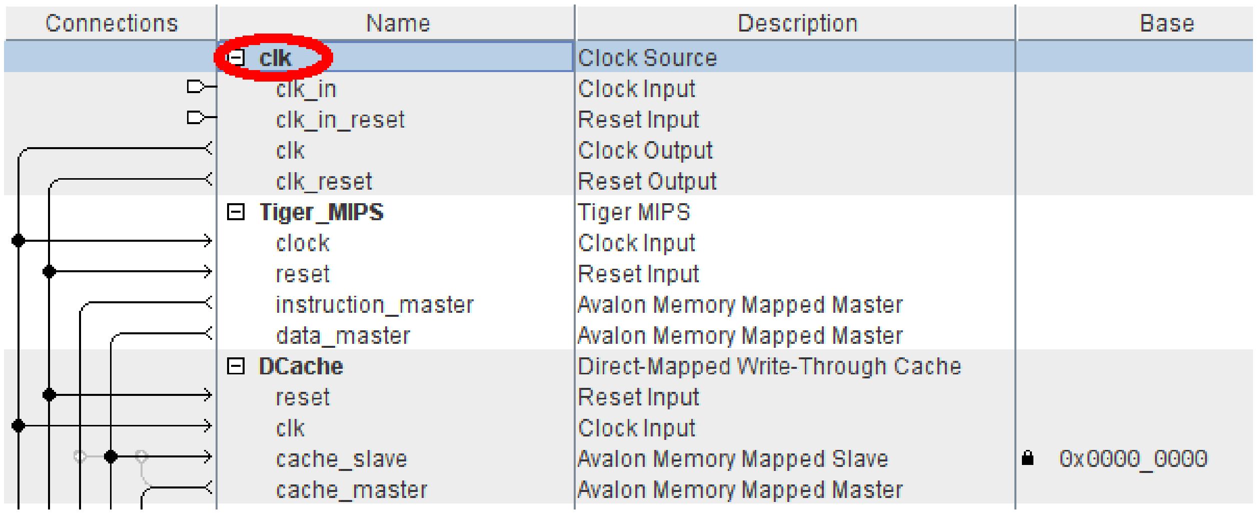
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_CLOCK_MODULE clk
```



Connections	Name	Description	Base
	⊟ **clk**	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ **Tiger_MIPS**	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	⊟ **DCache**	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.66 SYSTEM_DATA_CACHE_TYPE

This parameter specifies the type of cache used in the QSys system.

Category

Board and Device Specification

Value Type

String

Valid Values

Any QSys component that functions as a cache.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_DATA_CACHE_TYPE legup_dm_wt_cache
```

10.2.67 SYSTEM_MEMORY_BASE_ADDRESS

This parameter specifies the system memory base address. This is the address at which the program will be linked. It should be the same as the memory base address in QSys.

Category

Board and Device Specification

Value Type

Integer

Valid Values

The base address of the memory in QSys.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_MEMORY_BASE_ADDRESS 0x40000000
```

10.2.68 SYSTEM_MEMORY_INTERFACE

This parameter specifies the 'Avalon Memory Mapped Slave' interface for the system memory.

Category

Board and Device Specification

Value Type

String

Valid Values

The 'Avalon Memory Mapped Slave' interface of the QSys module that supplies the system memory.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_MEMORY_MODULE*

Applicable Flows

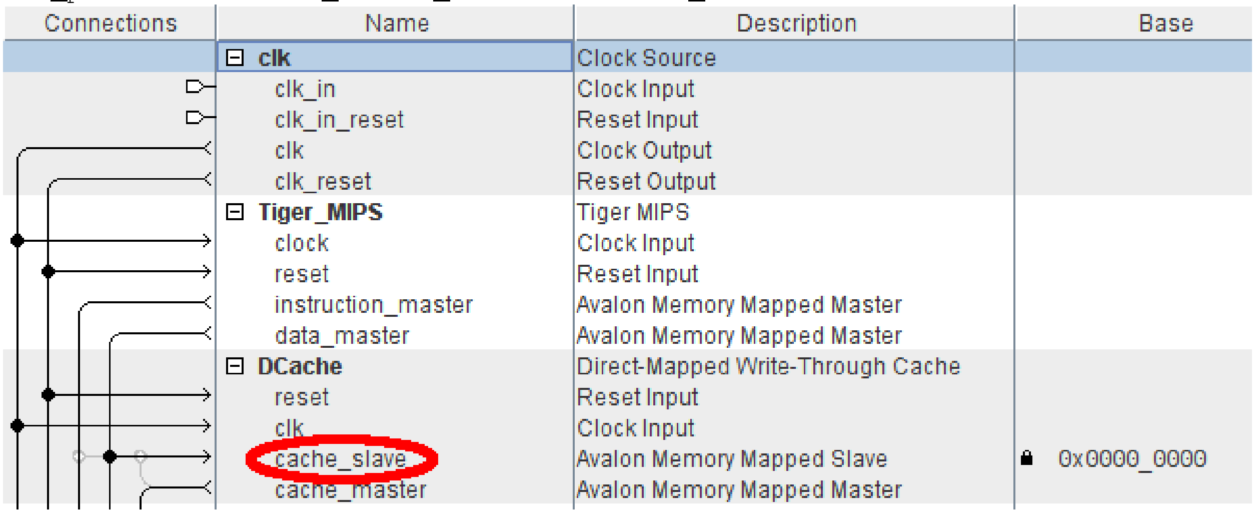
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_MEMORY_INTERFACE cache_slave
```



Connections	Name	Description	Base
	⊟ **clk**	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ **Tiger_MIPS**	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	⊟ **DCache**	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.69 SYSTEM_MEMORY_MODULE

This parameter specifies the QSys module that supplies the system memory.

Category

Board and Device Specification

Value Type

String

Valid Values

The name of the QSys module that supplies the system memory.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_MEMORY_INTERFACE*

Applicable Flows

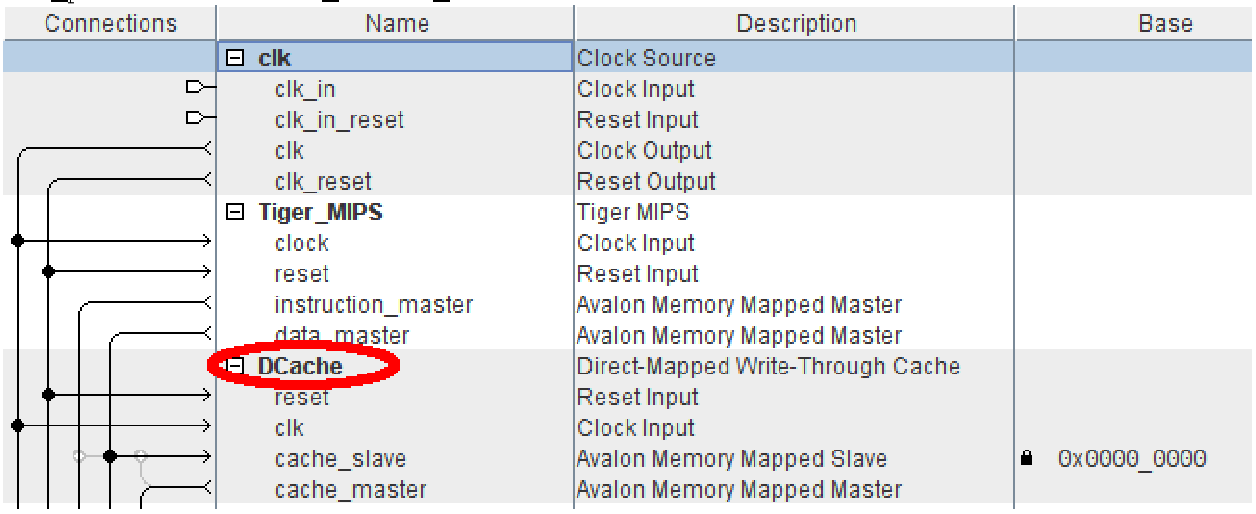
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_MEMORY_MODULE DCache
```



Connections	Name	Description	Base
	⊟ clk	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ Tiger_MIPS	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	⊟ DCache	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.70 SYSTEM_MEMORY_SIM_INIT_FILE_NAME

This parameter specifies the .dat file to be used to initialize the memory for hybrid system simulation.

Category

Board and Device Specification

Value Type

String

Valid Values

Any valid .dat file.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_MEMORY_SIM_INIT_FILE_TYPE*

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_MEMORY_SIM_INIT_FILE_NAME altera_sdram_partner_module.dat
```

10.2.71 SYSTEM_MEMORY_SIM_INIT_FILE_TYPE

This parameter specifies the file format of the simulation initialization file specified by *SYSTEM_MEMORY_SIM_INIT_FILE_NAME*.

Category

Board and Device Specification

Value Type

String

Valid Values

DAT

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_MEMORY_SIM_INIT_FILE_NAME*

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use.

Examples

```
set_parameter SYSTEM_MEMORY_SIM_INIT_FILE_TYPE DAT
```

10.2.72 SYSTEM_MEMORY_SIZE

This parameter specifies the size of the memory in the QSys system.

Category

Board and Device Specification

Value Type

Integer

Valid Values

The size of the system memory, in bytes.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_MEMORY_BASE_ADDRESS SYSTEM_MEMORY_WIDTH*

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_MEMORY_SIZE 0x04000000
```

10.2.73 SYSTEM_MEMORY_WIDTH

This parameter specifies the width of the memory component specified by *SYSTEM_MEMORY_MODULE*‘

Category

Board and Device Specification

Value Type

Integer

Valid Values

1, 2, 4, 8, etc.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_MEMORY_SIZE SYSTEM_MEMORY_BASE_ADDRESS*

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_MEMORY_WIDTH 2
```

10.2.74 SYSTEM_MUTEX_BASE_ADDRESS

This parameter specifies the base address of the mutex module in the QSys system.

Category

Board and Device Specification

Value Type

Integer

Valid Values

The base address of the Mutex module in QSys.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Applicable Flows

Software and hybrid flows

Test Status

Not used.

Examples

```
set_parameter SYSTEM_MUTEX_BASE_ADDRESS 0x00800000
```

10.2.75 SYSTEM_PROCESSOR_ARCHITECTURE

This parameter specifies the architecture of the processor in the QSys system.

Category

Board and Device Specification

Value Type

String

Valid Values

MIPSI, ARMA9

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use

Examples

`set_parameter SYSTEM_PROCESSOR_ARCHITECTURE MIPSI`

10.2.76 SYSTEM_PROCESSOR_DATA_MASTER

This parameter specifies the 'Avalon Memory Mapped Master' for the data interface of the system processor.

Category

Board and Device Specification

Value Type

String

Valid Values

The 'Avalon Memory Mapped Master' used for data by the QSys module that supplies the system processor.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_PROCESSOR_NAME*

Applicable Flows

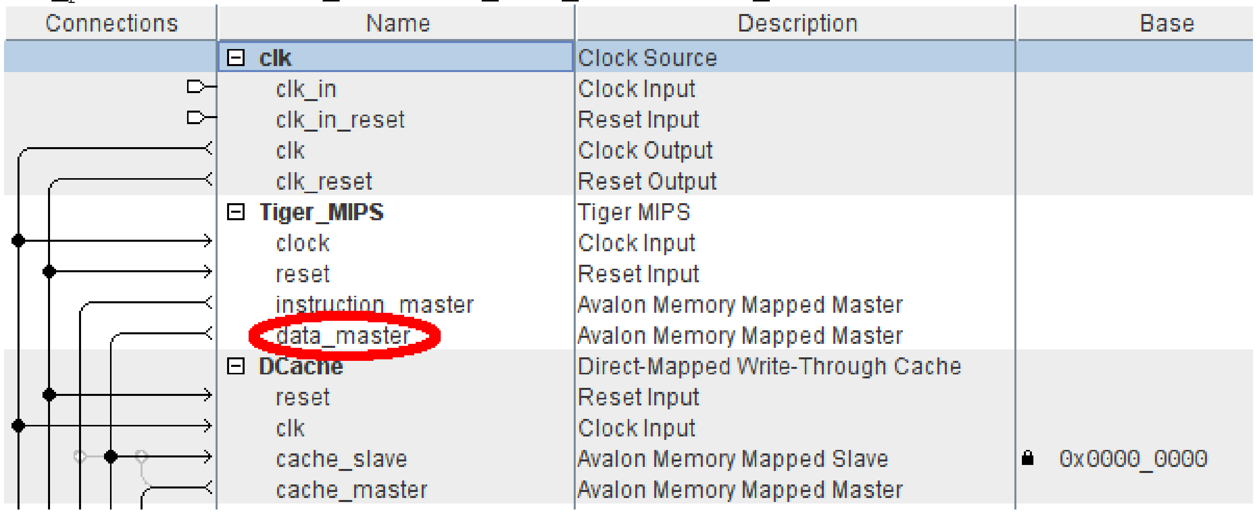
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_PROCESSOR_DATA_MASTER data_master
```



Connections	Name	Description	Base
	⊟ **clk**	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ **Tiger_MIPS**	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	⊟ **DCache**	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.77 SYSTEM_PROCESSOR_NAME

This parameter specifies the QSys module that supplies the system processor.

Category

Board and Device Specification

Value Type

String

Valid Values

The name of the QSys processor module.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Applicable Flows

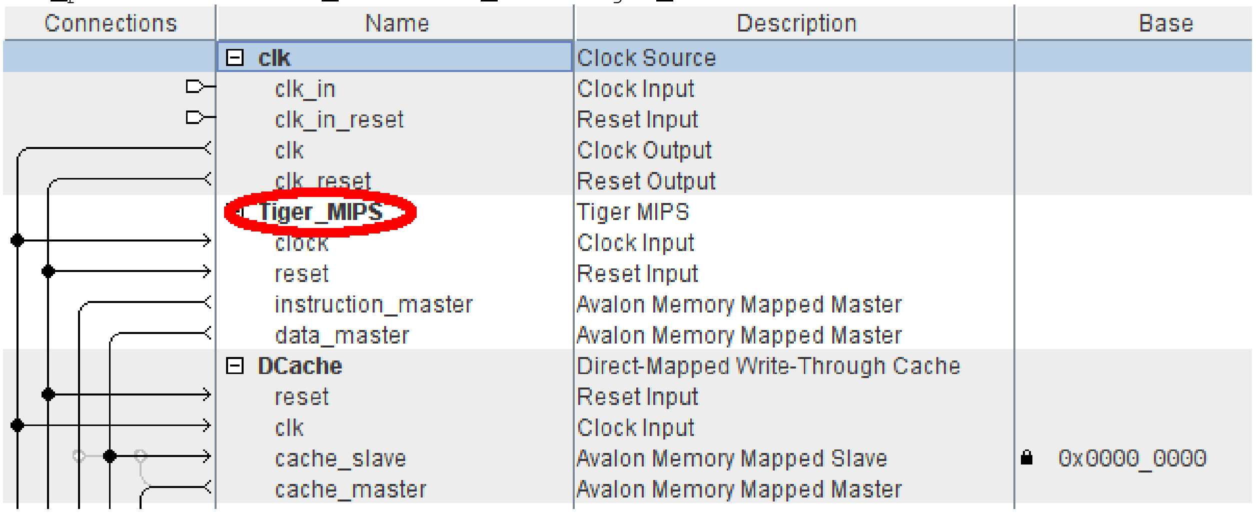
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_PROCESSOR_NAME Tiger_MIPS
```



Connections	Name	Description	Base
	⊟ clk	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	Tiger_MIPS	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	⊟ DCache	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.78 SYSTEM_PROJECT_NAME

The name of the directory containing the Quartus II project and Qsys system to be used by the software and hybrid makefile targets. The directory must also include a legup.tcl file that describes the Qsys system. This constraint is set in the set_project function and should not be set independently.

Category

Board and Device Specification

Value Type

String

Valid Values

Any valid directory name

Default Value

Tiger_SDRAM

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_PROJECT_NAME Tiger_SDRAM
```

10.2.79 SYSTEM_RESET_INTERFACE

This parameter specifies the 'Reset Output' interface of the QSys module that supplies the system reset.

Category

Board and Device Specification

Value Type

String

Valid Values

The 'Reset Output' interface of the QSys module that supplies the system reset.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_RESET_MODULE*

Applicable Flows

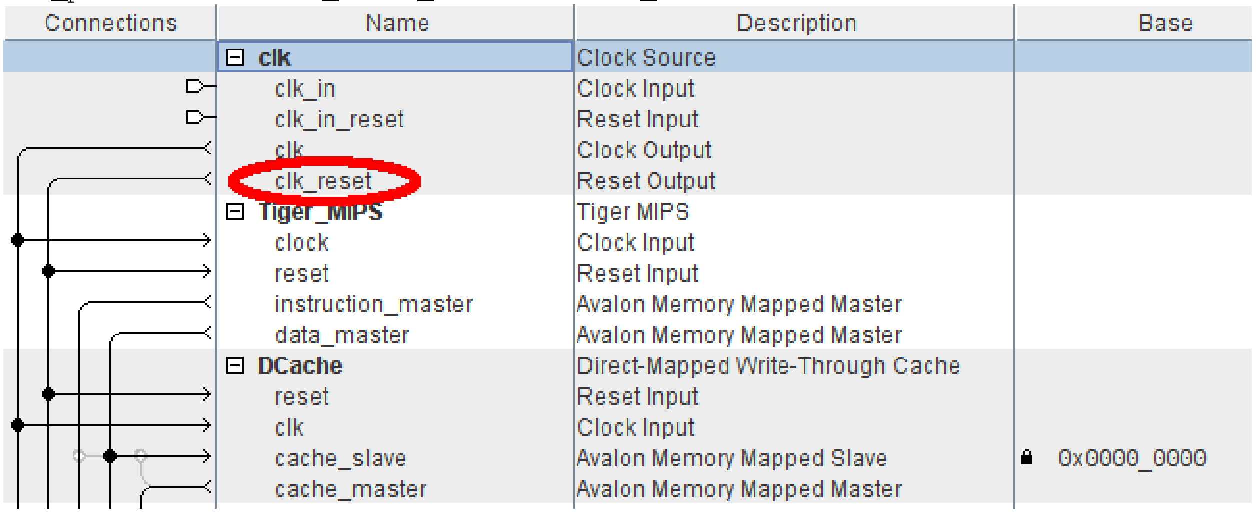
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_RESET_INTERFACE clk_reset
```



Connections	Name	Description	Base
	⊟ **clk**	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	⊟ **Tiger_MIPS**	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	⊟ **DCache**	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.80 SYSTEM_RESET_MODULE

This parameter specifies the QSys module that supplies the system reset. Probably the same as *SYSTEM_CLOCK_INTERFACE*.

Category

Board and Device Specification

Value Type

String

Valid Values

The name of the QSys module that supplies the system reset.

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<system>/legup.tcl`

e.g. `boards/CycloneV/DE1-SoC/Tiger_SDRAM/legup.tcl`

Dependencies

None

Related parameters: *SYSTEM_RESET_INTERFACE*

Applicable Flows

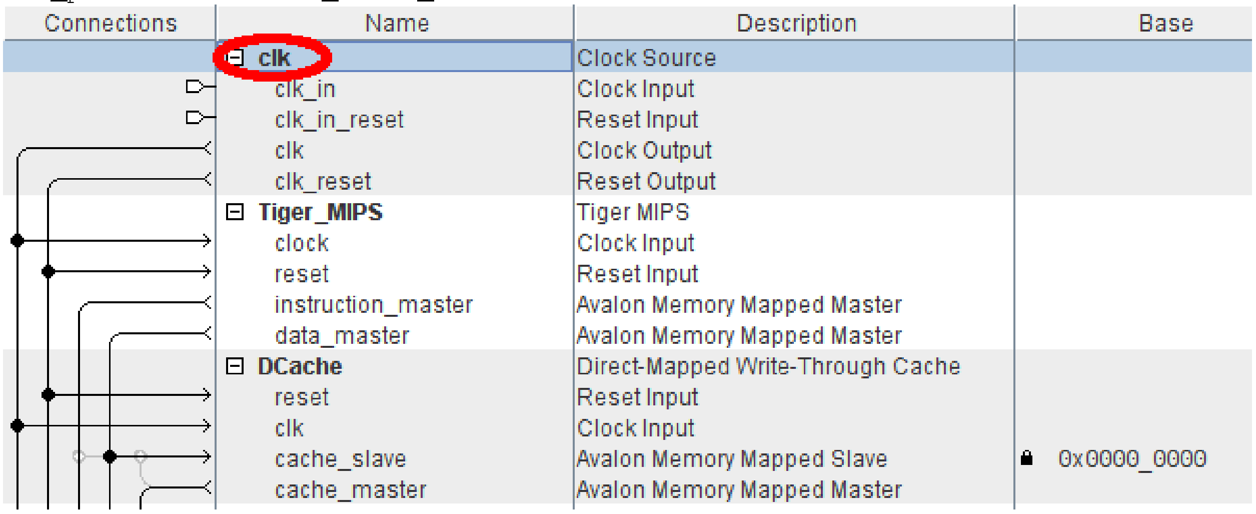
Software and hybrid flows

Test Status

Actively in-use

Examples

```
set_parameter SYSTEM_RESET_MODULE clk
```



Connections	Name	Description	Base
	clk	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	
	clk	Clock Output	
	clk_reset	Reset Output	
	Tiger_MIPS	Tiger MIPS	
	clock	Clock Input	
	reset	Reset Input	
	instruction_master	Avalon Memory Mapped Master	
	data_master	Avalon Memory Mapped Master	
	DCache	Direct-Mapped Write-Through Cache	
	reset	Reset Input	
	clk	Clock Input	
	cache_slave	Avalon Memory Mapped Slave	0x0000_0000
	cache_master	Avalon Memory Mapped Master	

10.2.81 TEST_WAITREQUEST

Test handling of the waitrequest signal by setting waitrequest high of 5 cycles at the beginning of every transfer. This should have no impact on circuit functionality but just make the total cycle count about 5x higher.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Untested

Examples

```
set_parameter TEST_WAITREQUEST 1
```

10.2.82 TIMING_NO_IGNORE_GETELEMENTPTR_AND_STORE

LLVM IR `store` and `getelementptr` instructions take an entire clock cycle in the Verilog generated by LegUp. By default these instructions are not printed in the timing reports `timingReport.overall.legup.rpt` and `timingReport.legup.rpt`. Setting this parameter to 1 will cause `store` and `getelementptr` instructions to be printed in the timing reports.

Category

Miscellaneous

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None.

Related parameters: *TIMING_NUM_PATHS*:

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter TIMING_NO_IGNORE_GETELEMENTPTR_AND_STORE 1
```

10.2.83 TIMING_NUM_PATHS

This parameter defines the number of paths to be printed in the timing reports `timingReport.legup.rpt` and `timingReport.overall.legup.rpt` when LegUp HLS is run. The timing reports will contain the TIMING_NUM_PATHS longest paths.

Category

Miscellaneous

Value Type

Integer

Valid Values

Positive integers

Default Value

10

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_parameter TIMING_NUM_PATHS 20
```

10.2.84 debugger_capture_mode

This parameter controls whether Blair's debugger is in 'Capture Mode'. See *use_debugger*.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

N/A

Dependencies

use_debugger

Applicable Flows

Pure HW and Hybrid flows

Test Status

Functional

Examples

```
debugger_capture_mode 0
```

```
debugger_capture_mode 1
```

10.2.85 set_custom_test_bench_module

This TCL command is to overwrite the name of testbench module to be elaborated in simulation.

Category

Simulation

Value Type

String

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
    set_custom_test_bench_module "custom_tb"
```

10.2.86 set_custom_top_level_module

This TCL command allows user to overwrite the name of top level entity in Quartus compilation.

Category

Quartus

Value Type

string

Dependencies

NONE

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_custom_top_level_module foo
```

10.2.87 set_custom_verilog_file

This TCL command is to specify the path of Verilog file that contains the definition of custom function.

Category

HLS Constraints

Value Type

String

Dependencies

set_custom_verilog_function

Applicable Flows

All devices and flows

Test Status

Prototype functionality

Examples

```
set_custom_verilog_file "stdio.v"
```

10.2.88 set_custom_verilog_function

This TCL command is to specify the name and module interface for a function with custom Verilog. LegUp will not generate Verilog for the functions specified by the command, but will still instantiate the corresponding module based on the specified interface and connect it with the rest of HLS-generated Verilog. This command requires multiple arguments, 1) function name, 2) memory access requirement and 3) direction, name and bit-width for each module port. The Verilog definition of the custom function should be provided via the `set_custom_verilog_file` command.

Category

HLS Constraints

Valid Values

Function name: string
Memory access requirement: one of the two strings, ‘memory’ or ‘noMemory’
Module port directory: one of the two strings, ‘input’ or ‘output’
Module port bit-width: a string in format, ‘high_bit:low_bit’, where high_bit and low_bit are integers
Module port name: a string with no whitespace.

Dependencies

set_custom_verilog_file

Applicable Flows

All devices and flows

Test Status

Prototype functionality

Examples

```
set_custom_verilog_function "boardGetChar" noMemory
    input 7:0 UART_BYTE_IN \
    input 0:0 UART_START_RECEIVE \
    input 1:0 UART_RESPONSE \
    output 17:2 LEDR
```

10.2.89 set_device_specs

This option is used to define certain device-specific parameters. All parameters are required. The parameters are described in the following table:

Parameter	Description
-Family	The device family
-Device	The device number
-MaxALMs	The total number of ALMs in the device
-MaxM4Ks	The total number of M4K memory blocks in the device
-MaxRAMBits	The total number of RAM bits that can be stored in the device
-MaxDSPs	The total number of DSP blocks in the device

You should not need to use `set_device_specs` unless you are adding support for a new board or device.

Category

Board and Device Specification

Value Type

Parameter	Value type
-Family	String
-Device	String
-MaxALMs	Integer
-MaxM4Ks	Integer
-MaxRAMBits	Integer
-MaxDSPs	Integer

Valid Values

See Examples

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<board>/<board>.tcl`

e.g.: `boards/CycloneV/DE1-SoC/DE1-SoC.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_device_specs -Family CycloneV -Device 5CSEMA5F31C6 -MaxALMs 32075 -MaxM4Ks
397 -MaxRAMBits 4065280 -MaxDSPs 87
```

```
set_device_specs -Family StratixV -Device 5SGXEA7N2F45C2 -MaxALMs 234720
-MaxM4Ks 2560 -MaxRAMBits 520428800 -MaxDSPs 256
```

10.2.90 set_legup_config_board

The name of the board being targetted by LegUp. This constraint is set in the set_project function and should not be set independently.

Category

Board and Device Specification

Value Type

String

Valid Values

Any valid directory name

Default Value

DE1-SoC

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_legup_config_board DE1-SoC
```

10.2.91 set_legup_output_path

This parameter sets the output path for all LegUp generated files. Only subdirectories of the current path are allowed. The makefile will create the full path, while the clean target will remove it.

Category

Miscellaneous

Value Type

String

Valid Values

All alphanumeric strings

Default Value

`output`

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_legup_output_path output
```

10.2.92 set_memory_global

This parameter can be used to force a particular array to be placed in global memory instead of local memory.

Category

HLS Constraints

Value Type

String

Valid Values

Array name

Default Value

None

Location Where Default is Specified

`examples/legup.tcl`

Dependencies

LOCAL_RAMS

Applicable Flows

All devices and flows

Test Status

No longer in-use

Examples

```
set_memory_global JpegFileBuf
```

10.2.93 set_memory_local

This parameter can be used to force a particular array to be placed in local memory instead of global memory.

Category

HLS Constraints

Value Type

String

Valid Values

Array name

Default Value

None

Location Where Default is Specified

```
examples/legup.tcl
```

Dependencies

LOCAL_RAMS

Applicable Flows

All devices and flows

Test Status

No longer in-use

Examples

```
set_memory_local JpegFileBuf
```

10.2.94 set_operation_attributes

This option is used to define certain device-specific parameters for a given operation. For example, `set_operation_attributes` can be used to set the fmax, latency, and resource usage for a pipelined divider on a Cyclone V device. All parameters are required. The parameters are described in the following table:

Parameter	Description
-Name	Operation name
-Fmax	Maximum frequency for operation
-CritDelay	Input to output delay for operation
-StaticPower	Static power for operation
-DynamicPower	Dynamic power for operation
-LUTs	Number of LUTs used by operation
-Registers	Number of registers used by operation
-LEs	Number of logic elements used by operation
-DSP	Number of DSP blocks used by operation
-Latency	Cycles of latency for operation

Note: You should not have to use `set_operation_attributes` unless you are adding support for a new board or operation.

Category

Board and Device Specification

Value Type

Parameter	Value Type
-Name	String
-Fmax	Real
-CritDelay	Real
-StaticPower	Real
-DynamicPower	Real
-LUTs	Integer
-Registers	Integer
-LEs	Integer
-DSP	Integer
-Latency	Integer

Valid Values

See Examples

Default Value

N/A

Location Where Default is Specified

`boards/<device_family>/<device_family>.tcl` e.g.: `boards/CycloneV/CycloneV.tcl`

Dependencies

None

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_operation_attributes -Name signed_add_16 -Fmax 471.48 -CritDelay 2.121
-StaticPower 0.00 -DynamicPower 0.00 -LUTs 16 -Registers 48 -LEs 24 -DSP 0
-Latency 0
```

```
set_operation_attributes -Name signed_multiply_64 -Fmax 112.3 -CritDelay 8.905
-StaticPower 0.00 -DynamicPower 0.00 -LUTs 62 -Registers 192 -LEs 100 -DSP 6
-Latency 0
```

```
set_operation_attributes -Name unsigned_multiply_pipelined_4_64 -Fmax 114.5
-CritDelay 8.734 -StaticPower 0.00 -DynamicPower 0.00 -LUTs 57 -Registers 512
-LEs 260 -DSP 6 -Latency 4
```

10.2.95 set_operation_sharing

This parameter allows operation sharing to be turned on or off for a given operation. Operation sharing is on by default for all operations.

Operation sharing adds multiplexing before and after an operation (e.g. divider) so that it can be used for multiple operations, instead of creating duplicate hardware.

Note: A constraint on "signed_add" will apply to:

- **signed_add**_8
- **signed_add**_16
- **signed_add**_32
- **signed_add**_64
- un**signed_add**_8
- un**signed_add**_16
- un**signed_add**_32
- un**signed_add**_64
- ...

Category

Constraints

Value Type

String

Valid Values

-on <operation>, -off <operation>

Default Value

unset ('-on' by default for all operations)

Location Where Default is Specified

By default, operation sharing is enbled for all operations in the LegUp source code.

A few operations are cheaper than multiplexing, so operation sharing is turned off for them in the file:

`examples/legup.tcl`

Dependencies

None

Related parameters: *RESTRICT_TO_MAXDSP*

Applicable Flows

All devices and flows

Test Status

Actively in-use

Examples

```
set_operation_sharing -off signed_add
```

```
set_operation_sharing -off signed_subtract
```

10.2.96 use_debugger

This parameter controls whether Blair's debugger is added to the Verilog generated by LegUp.

The debugger monitors all memory transactions and checks against a working golden data set to track down where the memory access messes up. The debugger is turned on using a tcl parameter, use_debugger, set in config.tcl. Another tcl parameter, *debugger_capture_mode*, is used to set the mode of the debugger. When it is set to 1 (which should be used on the working version of hardware), the simulation dumps out all memory accesses (address and data for port a and b on a read/write) to .dat files. Once this golden data set is created, set debugger_capture_mode to 0, then re-run legup and simulation on the non-working HW. This time the simulation compares the memory accesses in the non-working HW with the data in .dat files. If any memory access is different from the gold data set in .dat files, it prints out what the differences are, and stops the simulation right away.

Category

Debugging

Value Type

Integer

Valid Values

0, 1

Default Value

unset (0)

Location Where Default is Specified

N/A

Dependencies

None

Applicable Flows

Pure HW and Hybrid flows

Test Status

Functional

Examples

```
use_debugger 1
```

CHAPTER

ELEVEN

RELEASE NOTES

11.1 LegUp 4.0

11.1.1 Major new features

- Support for ARM processor in the hybrid flow (on the DE1-SoC board).
- Generic Verilog in the pure hardware flow that can be targeted to ANY FPGA vendor (e.g. Xilinx) or even ASICs. This is enabled through the use of generic dividers, RAM blocks and multipliers.
- Support for pthreads and OpenMP in the pure hardware flow. A processor is no longer necessary in the system for the HLS of software threads.
- Improved loop pipelining for loops with recurrences and resource constraints, as described in FPL 2014 paper (Canis et al.).
- Local and grouped RAMs. Pointer analysis is used to analyze when RAMs can be kept local to a Verilog module. LegUp is also able to group multiple arrays into a single RAM in the hardware.
- QSys compatibility so LegUp hybrid implementations can be compiled by the latest versions of Altera Quartus II.
- LLVM 3.5: LegUp 4.0 is built within the most recent version of LLVM.
- Device support: Altera Cyclone II, Cyclone IV, Cyclone V, Stratix IV, Stratix V.
- Custom Verilog support which gives the user the ability to tell LegUp to NOT perform HLS for a given function, for which they intend to provide their own Verilog.

11.1.2 Minor new features

- Constraints manual explaining all TCL constraints and their function.
- FSM implementation using case statement instead of if-else.

11.1.3 Beta features

- Bitwidth minimization, as described in ASP-DAC 2013 paper (Gort et al.).
- Multi-cycling support, as described in DATE 2015 paper (Hadjis et al.).
- HLS debugging support, as described in FPL 2014 paper (Calagar et al., Goeders et al.).
- If-conversion that flattens the control-flow graph in certain cases, which may improve performance and enable loop pipelining opportunities.

11.1.4 Contributors to LegUp 4.0

- Andrew Canis
- Jongsok (James) Choi
- Blair Fort
- Ruo Long (Lanny) Lian
- Bain Syrowik
- Nazanin Calagar
- Jeffrey Goeders
- Hsuan (Julie) Hsiao
- Yu Ting (Joy) Chen
- Mathew Hall
- Stefan Hadjis
- Marcel Gort
- Tomasz Czajkowski
- Stephen Brown
- Jason Anderson

11.2 LegUp 3.0

11.2.1 Major new features

- Loop pipelining for simple loops using iterative modulo scheduling
- Floating point support
- Multi-pumping of DSP-block multipliers tested on the DE4
- Hardware Profiler of hybrid flow working and fully scripted
- Multi-ported caches using LVT memories and multi-pumping memories
- Support for dual-port memories
- Support for resource constraints in SDC scheduler
- New minimize bitwidth pass
- Scheduling cycle prediction engine for hybrid and pure-hardware flow

11.2.2 Minor new features

- Improvements to the design of the internal memory controller
- Alias analysis for load/store scheduling
- Scripts to set multi-cycle timing constraints based on LegUp schedule
- New benchmarks to test multi-pumped multipliers with loop unrolling

- Tests for floating point operations
- Allow “local” rams that don’t use shared memory controller if no pointers alias
- Support for independent “local” rams to be accessed in parallel
- Can set metadata in LLVM IR for loop pipelining
- Multiplier resource sharing now only shares multipliers that infer DSPs
- Can specify the multiplier pipeline stages
- Support for C loop labels for loop pipelining
- Support for DDR2 memory for the Tiger MIPS processor on the Altera DE4 board
- Can use Edmonds matching algorithm to group multipliers into the same cycle
- Latex gantt chart generated for schedule
- Scripts to generate characterization for DSP-block multiplier
- Script to find the path between two signals in LegUp generated Verilog
- Code refactoring and removal of dead code

11.2.3 Beta features

- GUI for viewing scheduling report
- PCIe support for DE4
- Support for parallel accelerators using Pthreads and OpenMP

11.3 LegUp 2.0

11.3.1 Major new features

- SDC scheduler (see [Cong06])
- Pattern-based resource sharing (paper to appear in FPGA 2012)
- Added an online demo version of LegUp to the website
- LLVM version updated to 2.9
- Compiler front-end updated to clang (llvm-gcc is deprecated)
- Support for Stratix IV (DE4 board) with device characterization
- Added: Polly, CLooG, isl. These libraries support polyhedral loop dependency analysis
- New documentation with pdf version
- Cache simulator for TigerMIPS
- Memory access profiler for extracting parallel functions
- Added bit width minimization analysis used for pattern sharing
- Added live variable analysis pass used for binding/pattern sharing
- Significant code refactoring, both for clarity, modifiability, and also removal of dead code
- Tcl interface to control LegUp parameters: see examples/legup.tcl

- Supported Quartus version is now 10.1sp1

11.3.2 Minor new features

- New datastructure to represent the output circuit as Cell, Pin, and Net objects
- Register sharing for mul/div/rem functional units
- Binding restricts multiplier usage to DSPs available on FPGA
- Test suite examples now return non-zero values and print "RESULT: PASS" when successful
- Two new C example benchmarks: 1) 16-bit FFT, 2) 32-bit 16 tap FIR filter
- Connected signals are now verified to have equal bit width
- All signals that don't drive primary outputs are removed
- New log file: scheduling.legup.rpt, which lists the LLVM instructions assigned to each state
- New log file: binding.legup.rpt, which lists the patterns found and shared during binding
- Divider functional units now use clock enable instead of a counter
- Most classes no longer inherit from LLVM's FunctionPass to avoid LLVM PassManager issues
- LegupTcl and LegupConfig files moved into llvm/lib/Target/Verilog directory
- Combinational loops are detected and avoided in binding
- Verilog variable names now include the LLVM register name, basic block name, and C function name
- State names are now appended with the actual state number
- Makefile now supports parallel make ie. make -j4
- benchmark.pl parser now supports StratixIV and TimeQuest
- Makefile can support linking multiple C files

11.3.3 Bug Fixes

- Altsyncrams now have correct intended_device_family
- Fixed memory leaks using valgrind
- No more Quartus warnings when synthesizing Verilog output
- Fix to Verilog output "if" statements that reduced ALM count
- Added warning for uninitialized variables
- Fixed varXXXX variable postfix changing between runs
- Combinational always blocks now use blocking assignment
- Removed inferred latches

11.3.4 Improvements to the hybrid flow

- Designed a new memory controller for hardware accelerators to control memory accesses between global and local memory
- Added a test suite to accelerate each function in all benchmarks. All functions return the correct result.

- Added burst capability and pipeline bridges to the processor
- Combined 3 avalon ports from accelerator into 1 port for stability and reduced area
- One new C example benchmark, memory_access_test, to test different memory access patterns
- Fixed simulation path and SOPC generation issues which were causing problems for certain users
- Fixed minor bugs in data cache

11.3.5 Beta features

- Hardware profiler for TigerMIPS soft processor
- Loop pipelining using Iterative Modulo Scheduling

11.4 LegUp 1.0

11.4.1 Features

- C to Verilog high-level synthesis tool. Tested on Linux 32/64-bit.
- Supports CHStone benchmark suite and dhrystone benchmarks
- Tiger MIPS processor from the University of Cambridge
- ASAP/ALAP scheduling with operator chaining and pipelined functional units
- Binding for multipliers and dividers using bipartite weighted matching
- Quality of results for Cyclone II are given in [Canis11]. We've found that the area-delay product over our benchmarks is compariable to eXCite, a commercial high-level synthesis tool.

[Cong06] J. Cong and Z. Zhang, “An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation,” Proceedings of the 2006 Design Automation Conference, San Francisco, CA, pp. 433-438, July 2006.

[Canis11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, T. Czajkowski, “LegUp: High-level synthesis for FPGA-based processor/accelerator systems,” ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pp. 33-36, Monterey, CA, February 2011.

T

test suite, 32