October 2022

# Integration of Digital Signal Processing Block in SymbiFlow FPGA Toolchain for Artix-7 Devices

Andrew T. Hartnett
*University of Massachusetts Amherst*

# INTEGRATION OF DIGITAL SIGNAL PROCESSING BLOCK IN SYMBIFLOW FPGA TOOLCHAIN FOR ARTIX-7 DEVICES

A Thesis Presented

by

ANDREW HARTNETT

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2022

Electrical and Computer Engineering

# INTEGRATION OF DIGITAL SIGNAL PROCESSING BLOCK IN SYMBIFLOW FPGA TOOLCHAIN FOR ARTIX-7 DEVICES

A Thesis Presented

by

ANDREW HARTNETT

Approved as to style and content by:

________________________________________

Russell Tessier, Chair

________________________________________

Wayne Burleson, Member

________________________________________

Daniel Holcomb, Member

________________________________________

Christopher V. Hollot, Department Head

Electrical and Computer Engineering

## ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Russell Tessier for providing me the opportunity to continue my academic growth. Through learning from him, I believe I have been impacted for the better in understanding the resolve and determination needed to see a project through, and for that I am grateful. My appreciation extends to colleagues and friends Anurag Muttur, Tien Li Shen, and Shayan Moini as well.

Lastly, I am profoundly grateful for the endless support from my family. Without them and my partner Kelly, this work would not have seen its completion. Thank you.

# ABSTRACT

## INTEGRATION OF DIGITAL SIGNAL PROCESSING BLOCKS IN SYMBIFLOW FPGA TOOLCHAIN FOR ARTIX-7 DEVICES

SEPTEMBER 2022

ANDREW HARTNETT
B.S., UNIVERSITY OF MASSACHUSETTS AMHERST
M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier

The open-source community is a valuable resource for many hobbyists and researchers interested in collaborating and contributing towards publicly available tools. In the area of field programmable gate arrays (FPGAs) this is no exception. Contributors seek to reverse-engineer the functions of large proprietary FPGA devices. An interesting challenge for open-source FPGA engineers has been reverse-engineering the operation and bitstreams of digital signal processing (DSP) blocks located in FPGAs. SymbiFlow is an open-source FPGA toolchain designed as a free alternative to proprietary computer-aided design tools like Xilinx's Vivado. For SymbiFlow, mapping logical multipliers to DSP blocks and generating DSP block bitstreams has been left unimplemented for the Artix-7 family of FPGAs. This research seeks to rectify this shortcoming by introducing DSP information for the place and route functions into SymbiFlow. By delving into the SymbiFlow architecture definitions and creating functioning FPGA assembly code (FASM) files for Project X-Ray, a bitstream generator for Artix-7, we have been able to determine the desired output of the open-source Versatile Place & Route tool that will generate a working DSP bitstream. We diagnose and implement changes needed throughout the SymbiFlow toolchain, allowing for DSP design bitstreams to be successfully generated with open-source tools.

# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS …………………………………………………………………….. iv**

**ABSTRACT ………………………………………………………………………………v**

**TABLE OF CONTENTS ……………………………………………………………….. vi**

**LIST OF TABLES ……………………………………………………………………...viii**

**LIST OF FIGURES …………………………………………………………………... ix**

**CHAPTER 1 - INTRODUCTION …………………………………………………………1**

**1.1 Xilinx Vivado ………………………………………………………………………3**

**1.2 SymbiFlow Toolchain ……………………………………………………………... 3**

**1.3 DSP Blocks …………………………………………………………………………4**

**1.4 Thesis Outline ……………………………………………………………………....7**

**CHAPTER 2 - SYMBIFLOW ……………………………………………………………... 8**

**2.1 Yosys ………………………………………………………………………………...9**

**2.2 VPR / Genfasm ……………………………………………………………………..9**

**2.3 Project X-Ray ……………………………………………………………………….. 10**

**2.4 Using SymbiFlow …………………………………………………………………11**

**2.5 State of DSP Block Usage in SymbiFlow ………………………………………….. 13**

**CHAPTER 3 - ARCHITECTURE DEFINITIONS AND GENFASM ……………………....15**

**3.1 SymbiFlow Architecture Definitions ……………………………………………15**

**3.1.1 Architecture Primitives ……………………………………………………...** 15

**3.1.2 Generating Routing Graph and Architecture File …………………………**17

**3.1.3 Developing A Basic DSP48E1 Primitive ………………………………....** 17

**3.2 2x2 bit Multiplier Design Under Test (DUT) ……………………………………19**

**3.2.1 Symmetric Flow through SymbiFlow and Vivado ………………………....**19

**3.2.2 Differences between SymbiFlow and Vivado FASM files ……………….……...** 21

**CHAPTER 4 – COMPLETE SYMBIFLOW DSP BITSTREAM …………………………... 24**

**4.1 Diagnosing DSP Support Points ………………………………………………….... 24**

**4.2 Investigating the Vivado Routing Resources GUI ………………………………... 24**

**4.3 Yosys DSP Information into VPR ………………………………………………… 30**

**4.4 Developing DSP48E1 Primitive …………………………………………………… 32**

**4.5 Changes to Genfasm ………………………………………………………………………... 34**
**4.6 Summary of Modifications …………………………………………………………………35**
**CHAPTER 5 – EXPANDING DSP FUNCTIONALITY ……………………………………….. 36**
**5.1 Full 25x18 bit Multiplier ……………………………………………………………………36**
**5.2 Partial Pre-adder ……………………………………………………………………………39**
**5.3 Pipelined Multiplier ………………………………………………………………………… 42**
**CHAPTER 6 – CONCLUSION …………………………………………………………………...46**
**6.1 Future Work ………………………………………………………………………………… 46**
**6.1.1 Incorporating C Port for DSP48E1 ……………………………………………………...**46
**6.1.2 Investigating the D Port Congestion Issue ……………………………………………….**47
**6.1.3 Allowing Behavioral Modeling of DSP48E1 ……………………………………………..**47
**BIBLIOGRAPHY …………………………………………………………………………………48**

# LIST OF TABLES

**Table** **Page**

**1** - STEPS CALLED AND TOOLS USED WITHIN SYMBIFLOW …………………………. 13
**2** - DSP48E1 PORTS TIED TO GND NEEDED FOR WORKING 2X2 MULTIPLIER ……... 27
**3** - MODE AND INVERTED SIGNALS FOR A*B PORT MULTIPLIER ……………………29
**4** - PORTS REMOVED FROM VPR LIBRARIES BASED ON VIVADO GUI ……………... 31
**5** - MODE AND INVERTED SIGNALS FOR 25X18 BIT MULTIPLIER ……………………37
**6** - LED OUTPUT COMBINATIONS OF 25X18 BIT DSP MULTIPLIER ………………….. 38
**7** - MODE AND INVERTED SIGNALS FOR MULTIPLIER WITH PRE-ADDER ………… 40
**8** - LED OUTPUT COMBINATIONS OF MULTIPLIER WITH PRE-ADDER ……………... 42

# LIST OF FIGURES

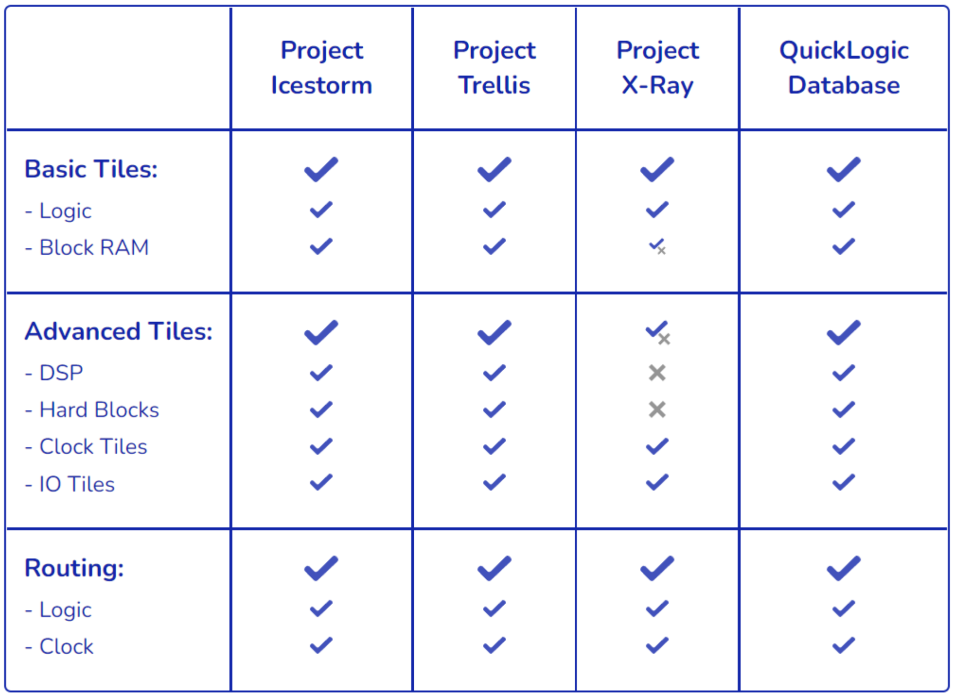
**Figure** **Page**

**1** - SUPPORTED ARCHITECTURE FOR VARIOUS POPULAR FPGAS …………………… 2
**2** - VISUALIZATION OF TOOLS INVOLVED IN SYMBIFLOW TOOLCHAIN …………... 4
**3** - INTERNAL ARCHITECTURE OF 7 SERIES DSP48E1 SLICE …………………………... 5
**4** - DSP48 MACRO PROVIDED BY VIVADO FOR DSP BLOCK CREATION …………….. 6
**5** - DETAILED SYMBIFLOW TOOLCHAIN WITH ARCHITECTURE DEFINITIONS AND INTERMEDIATE FILES …………………………………………………………….. 8
**6** - COUNTER_TEST.V, PART OF SYMBIFLOW-EXAMPLES ……………………………11
**7** - COMMON.MK MAKEFILE USED TO CALL STEPS WITH SYMBIFLOW ……………12
**8** - SNAPSHOT OF RAMB18E1.MODEL.XM …………………………………………….. 16
**9** - SNAPSHOT OF BRAM.PB_TYPE.XML SHOWING RAMB18E1 INTERCONNECTS .. 16
**10** - PRELIMINARY DSP48E1 PRIMITIVE USED FOR PLACE AND ROUTE ……………18
**11** - CUSTOM 2X2 BIT UNSIGNED MULTIPLIER USING DSP48E1 …………………….. 19
**12** - SYMMETRIC FLOW BETWEEN SYMBIFLOW AND VIVADO ……………………... 20
**13 -** FASM FILE INFORMATION COMPARISON ……………………………………….. 22
**14 -** ASSISTED WORKING 2X2 BIT MULTIPLIER FLOW THROUGH SYMBIFLOW …..24
**15** - VIVADO ROUTED FLOORPLAN FOR 2X2 MULTIPLIER ……………………………25
**16 -** SWITCHBOX INTERCONNECT FASM FILE WITH GUI …………………………….. 26
**17** - ROUTING RESOURCE GUI VIEW OF DSP BLOCK ………………………………….. 28
**18** - 2X2 BIT MULTIPLIER RTL CODE WITH PARAMETERS …………………………… 29
**19 -** PART OF CELLS_MAP.V DEFINITION OF A DSP48E1_VPR CELL ………………... 32
**20** - "FASM_PARAMS" SECTION OF DSP48E1 PRIMITIVE ……………………………… 33
**21** - RTL ASSIGNMENT OF INPUT SWITCHES ONTO A AND B INPUT SIGNALS …… 38
**22** - DSP48E1 INTERNALS WITH PRE-ADDER INTO 25X18 BIT MULTIPLIER ……….. 39
**23** - RTL ASSIGNMENT OF INPUT SWITCHES ONTO A, B, AND D INPUT SIGNALS .. 41
**24** - PIPELINED REGISTER FLOW WITHIN THE DSP48E1 ……………………………… 43
**25** - RTL INSTANTIATION OF A PIPELINED 25X18 BIT MULTIPLIER DSP48E1 ……... 44
**26** - CLOCK DIVIDER USED TO DRIVE DSP48E1 ………………………………………… 44

# CHAPTER 1 - INTRODUCTION

While Application Specific Integrated Circuits (ASICs) have been used as the primary hardware for digital logic implementation over the past few decades, the recent growth in popularity of Field Programmable Gate Arrays (FPGAs) now provides them with competition. FPGAs offer the reconfigurability of hardware post-manufacturing, which leads to more inexpensive implementation of digital circuitry. This flexibility is provided at the cost of optimality and efficiency towards a specific task which is a strong benefit of ASICs. Nevertheless, work has continued with FPGAs to leverage their reprogrammable digital hardware. Creative new applications for the FPGA include implementing the devices in data centers [1] and in hybrid-form with CPUs [2]. For data centers, work has been done to demonstrate the feasibility of allowing users to access portions of the device through the cloud and dividing the same chip amongst multiple tenants [1]. Further research has been conducted in combining the FPGA architecture with that of the existing CPU in modern computers. Doing so would allow for data transfer between the two devices, where FPGA reconfigurability and CPU efficiency can both be utilized [2].

Currently, popular FPGA vendors include Xilinx (AMD) [3], Altera (Intel) [4], and Lattice [5], among others. This work focuses on the Xilinx Artix-7 family of FPGAs and its Vivado Design Suite [6], [7] due to the popularity of the vendor and tools. Vivado offers the ability to generate bitstreams for their FPGAs from code written in a hardware description language (HDL). A commonly used HDL for this task is called Verilog. Using Verilog, one can describe the desired hardware configuration needed at register-transfer level (RTL), and then use Vivado to conduct the full flow of synthesis, place & route, and bitstream generation. As an alternative to proprietary tools like Vivado, the open-source synthesis toolchain SymbiFlow [8] seeks to emulate the same functions of synthesis, place & route, and bitstream generation for a variety of FPGA families. An important feature present in Vivado yet missing in SymbiFlow at the time of this writing is the processing of designs that use digital signal processing (DSP) blocks. These blocks are commonly used for bit multiplication and pattern detection in digital circuitry meant to emulate analog processes. One example of their use is in applications needing double-precision floating-point operations, presenting a large step forward in reducing necessary hardware resources [9]. With DSP block support missing in the SymbiFlow toolchain, designs which could use DSP blocks are not able to successfully generate bitstreams for Artix-7 devices. The SymbiFlow authors demonstrate this in a table of implemented architectural features, copied here as Figure 1 [8]. The figure includes information for four different bitstream generation tools, each focused on FPGAs from a unique vendor. Project Icestorm [10] and Project Trellis [11] provide bitstream documentation for the Lattice iCE40 and Lattice ECP5 [5] architectures, respectively. Project X-Ray [12] documents the architecture of Xilinx 7-Series devices and is used in this work. Lastly, the QuickLogic [13] database documents bitstreams of the EOS-S3 and QLF-K4N8 FPGAs [14].



| | Project Icestorm | Project Trellis | Project X-Ray | QuickLogic Database |
|---|---|---|---|---|
| **Basic Tiles:** | ✓ | ✓ | ✓ | ✓ |
| - Logic | ✓ | ✓ | ✓ | ✓ |
| - Block RAM | ✓ | ✓ | ✓✗ | ✓ |
| **Advanced Tiles:** | ✓ | ✓ | ✓✗ | ✓ |
| - DSP | ✓ | ✓ | ✗ | ✓ |
| - Hard Blocks | ✓ | ✓ | ✗ | ✓ |
| - Clock Tiles | ✓ | ✓ | ✓ | ✓ |
| - IO Tiles | ✓ | ✓ | ✓ | ✓ |
| **Routing:** | ✓ | ✓ | ✓ | ✓ |
| - Logic | ✓ | ✓ | ✓ | ✓ |
| - Clock | ✓ | ✓ | ✓ | ✓ |

**Figure 1** - Supported architecture for various popular FPGAs. Project X-Ray is the bitstream generation tool for the Xilinx Artix-7 family of boards (from [8]**)**

Through this work, we diagnose points within the SymbiFlow toolchain that need modification to offer DSP block support. Completing these changes allows us to generate the first DSP design bitstream in SymbiFlow for Artix-7 FPGAs, enhancing the abilities of the open-source toolchain. From here, we broaden the functionality of the block to include support for the full 25x18 bit multiplier, a partial pre-adder, and pipelined multiplier support. The continuation of this introduction will provide a comparison of the Vivado Design Suite and the current state of the SymbiFlow toolchain. Then, an overview of the Artix-7 digital signal processing (DSP) block, referred to as the DSP48E1, will be provided.

## 1.1 Xilinx Vivado

Xilinx Vivado [7] is a proprietary computer-aided design tool used in FPGA development to create bitstreams. Its steps can be broken down into synthesis, implementation, and bitstream generation. Following synthesis, the user is provided with a gate-level netlist of the input RTL design along with resource utilization statistics, including the percentage of I/O ports and other logic gates used by the FPGA. Warnings and errors are shown if design rule checks fail during this step. The physical design step performs place and route of cells onto the target FPGA architecture. A visual representation of the routing network is made available via a graphical user interface (GUI). All mapped cells and the routed logic between them can be seen via this user interface. Finally, the bitstream generation step creates an FPGA-specific bitstream file that can be loaded onto an FPGA. Each of these functions provides information to the user in the form of log files and customization options. However, the proprietary nature of Vivado prevents source-code-level configurability and access to algorithms used in each step.

## 1.2 SymbiFlow Toolchain

The purpose of the SymbiFlow toolchain [8] is to provide a free and open-source alternative to proprietary tools like Vivado for creating FPGA designs and bitstreams. In combining tools for synthesis, place & route, and bitstream generation, SymbiFlow allows users to take an RTL design and generate bitstreams functioning on FPGAs from several vendors. These FPGA vendors include Xilinx, Lattice, and QuickLogic as well as limited support for others. The Xilinx 7-series FPGAs are the focus of this study, with particular focus on the Artix-7 35TCSG324-1 FPGA. For the Artix-7 chip, SymbiFlow presents a flow using the following tools: Yosys [15] for synthesis, Versatile Place & Route (VPR) [16] for place and route, and Project X-Ray [12] for bitstream generation. A simplified visual representation of these tools within the toolchain can be seen in Figure 2. A version of each tool comes provided with installation of SymbiFlow.

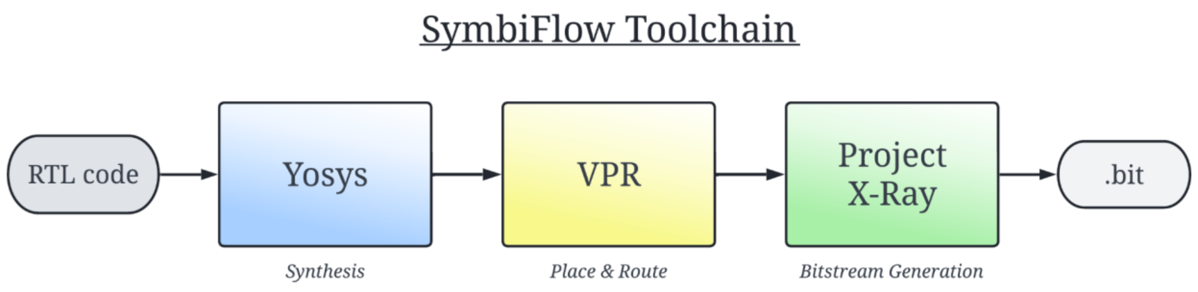


**Figure 2** - Visualization of tools involved in SymbiFlow toolchain

## 1.3 DSP Blocks

In FPGAs, DSP blocks are circuits used to implement multiplication, multiply-accumulate, and similar arithmetic operations. The Arty 35T FPGA used in this work offers 90 of these DSP blocks spread across two columns, while the number available DSPs on larger Artix-7 chips can rise above 2,500 [7], [17]. The DSP block (also called a slice) native to Xilinx 7 series FPGAs is referred to as a DSP48E1. These cells can be used for a variety of tasks based on their architecture. Some of the functions provided by Xilinx for the DSP48E1 are multiply, multiply-accumulate, multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bitwise logic functions, pattern detect, and wide counter [17].

Figure 3 shows the internal architecture of the DSP48E1 block from the Xilinx 7 Series DSP48E1 Slice User Guide [17]. Data can enter the slice through input ports A, B, C, and D, each of which is wired internally through different logic functions. For instance, to implement the 25x18 bit multiplier, data can enter through the A and B ports. Alternatively, setting the “USE_DPORT” parameter to “TRUE” in the RTL design, the multiplier could be configured to output the function B×D instead. The pre-adder can be included in a similar fashion to produce the function B×(A+D). The output of the DSP block is directed through the P port. In addition to the ports shown, reset and clock enable signals exist for the A through D inputs to allow for pipelining.

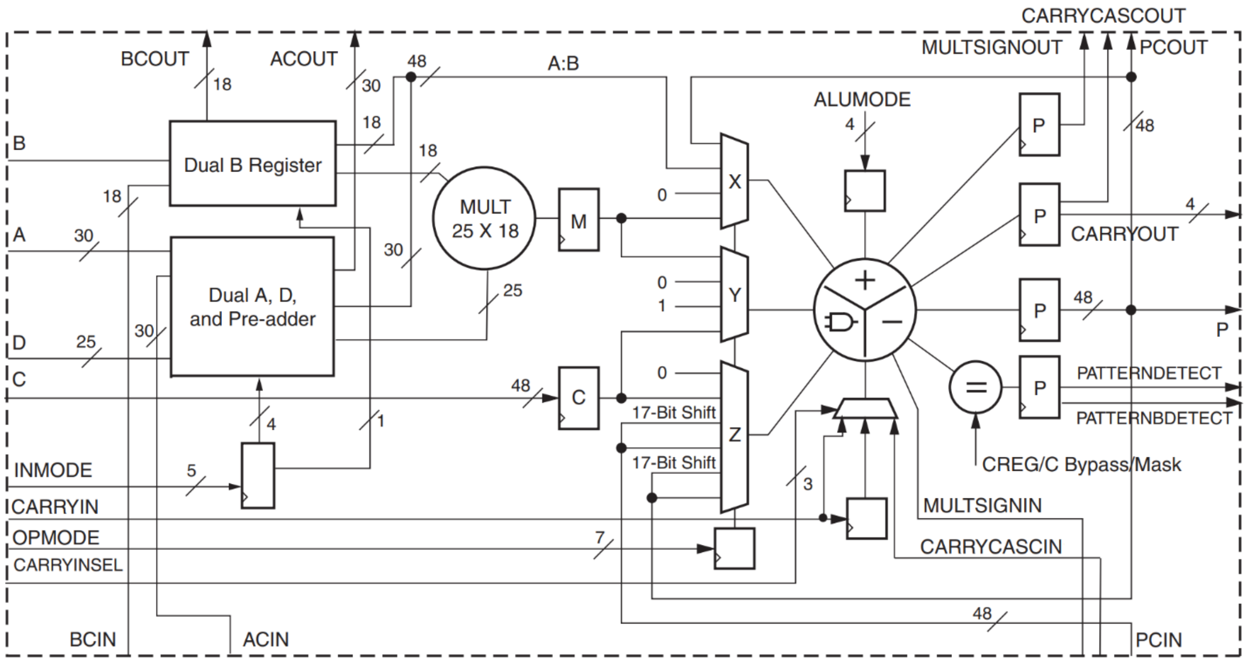
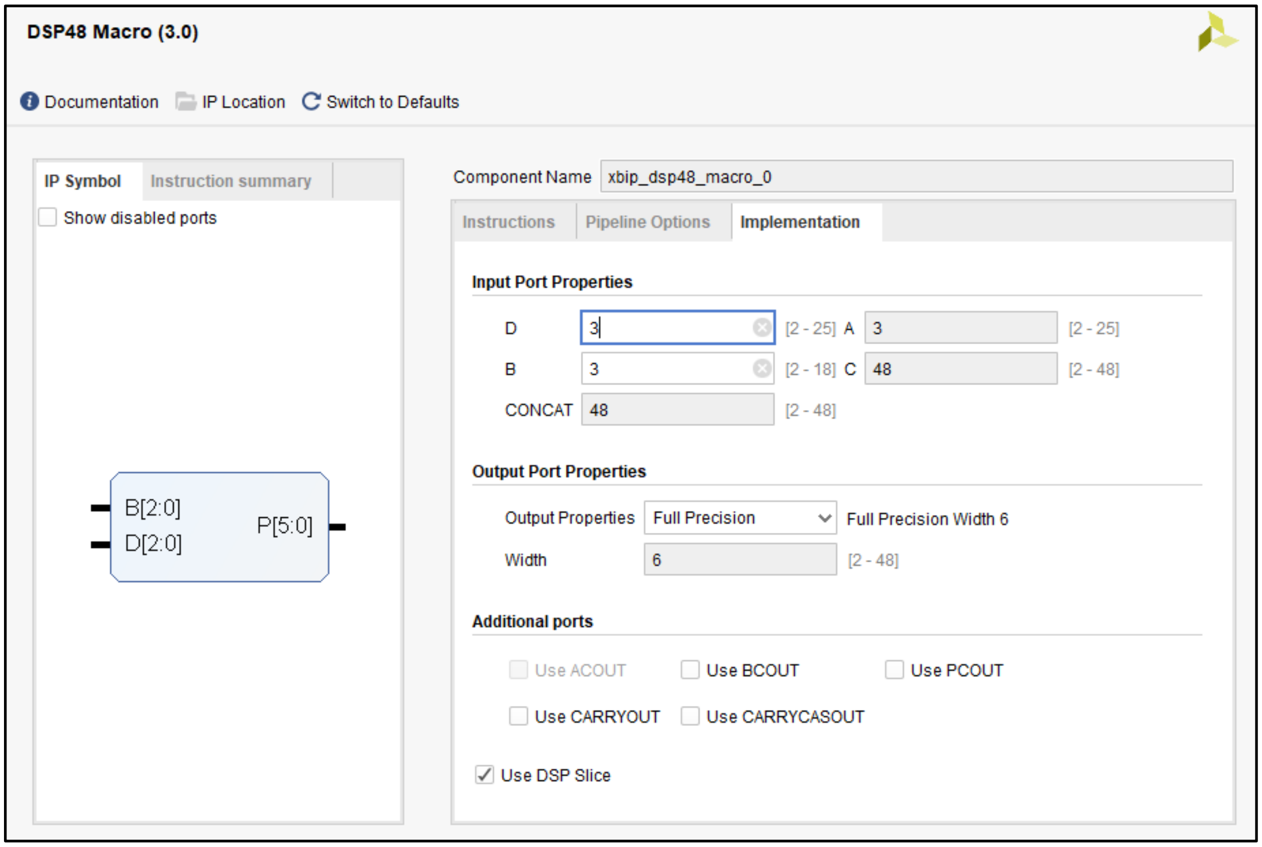


**Figure 3** - Internal architecture of 7 Series DSP48E1 slice (from [17])

To determine the internal routing of signals and function implementation, there are three important inputs: INMODE, OPMODE, and ALUMODE. These signals are connected to multiplexers X, Y, and Z found between the logic units in the DSP48E1. INMODE is responsible for setting the output of the pre-adder, optionally choosing between inputs A and D as the output to be fed into the 25x18 multiplier. OPMODE controls the signals provided to the secondary logic unit. Finally, ALUMODE sets the function to be conducted within the secondary logic unit, taking in the outputs of multiplexers X, Y, and Z. An incorrect initialization of any of these values will result in unintended behavior of the DSP48E1 slice and are critical towards proper configuration of the block.

DSP blocks can be processed by Vivado using an intellectual property (IP) module called the DSP48 Macro [18]. Using the GUI, a user can instantiate the DSP48 Macro with their desired logic function and I/O signal bit widths. This approach simplifies needing to understand values for signals INMODE, OPMODE, and ALUMODE, as they are automatically set to implement the given logic function. A snapshot of the DSP48 Macro is shown in Figure 4. The user's defined DSP slice is in a simplified form on the left side of the window. In this example, a B×D multiplier was selected in the "Instructions" tab and the bit widths for these inputs are being set to three bits wide. This implements a 3x3 bit signed multiplier. Alternatively, if the MSB of each input is set to 0, a 2x2 bit unsigned multiplier can be implemented.



DSP48 Macro (3.0)

Documentation | IP Location | Switch to Defaults

IP Symbol | Instruction summary

☐ Show disabled ports

Component Name: xbip_dsp48_macro_0

Instructions | Pipeline Options | Implementation

**Input Port Properties**

| | | | | | |
|---|---|---|---|---|---|
| D | 3 | [2 - 25] | A | 3 | [2 - 25] |
| B | 3 | [2 - 18] | C | 48 | [2 - 48] |
| CONCAT | 48 | [2 - 48] | | | |

**Output Port Properties**

Output Properties: Full Precision — Full Precision Width 6

Width: 6 [2 - 48]

**Additional ports**

☐ Use ACOUT ☐ Use BCOUT ☐ Use PCOUT

☐ Use CARRYOUT ☐ Use CARRYCASOUT

✓ Use DSP Slice

**Figure 4** - DSP48 Macro provided by Vivado for DSP block creation (from [7])

An important note about IP cores in Vivado is that unlike HDL written by the user, synthesized IP module information is encrypted post-synthesis. This approach denies the user access to configurability outside of the DSP48 Macro GUI and in the synthesized netlist. The need for encryption likely arises due to Xilinx's ownership of these IP cores; it is meant to prevent other individuals or companies from stealing their work and becoming a competitor. In all, Vivado provides support for the DSP48E1 slice with a detailed wizard allowing implementation of a user's exact needs.

At time of this writing, DSP block processing leading to bitstream generation has remained unimplemented for the Artix-7 SymbiFlow toolchain. The inclusion of DSP functionality is considered valuable due to the popularity of Artix-7 devices. Proprietary tools are often inaccessible for open-source contributors and individual researchers, lending to the need for further contributions towards open-source initiatives such as SymbiFlow. Addressing DSP implementation for the Artix-7 FPGA family offers insight into similar efforts for other FPGA families in the future.

## 1.4 Thesis Outline

This document is divided into five chapters, with this introduction serving as the first. An in-depth view into the tools that make up SymbiFlow, how it is used and its status for DSP block support for the Artix-7 family of FPGAs is presented in Chapter 2. Chapter 3 details the SymbiFlow FPGA architecture definitions and FPGA assembly (FASM) file formats used to generate bitstreams. A method of reverse engineering proper DSP FASM information from Vivado is presented as well. Chapter 4 breaks down all modifications made within SymbiFlow to support a basic use of the DSP48E1 block. Finally, Chapter 5 describes how this basic implementation has been expanded to include functions such as the 25x18 bit multiplier, partial pre-adder, and pipelining support.

# CHAPTER 2 - SYMBIFLOW

The SymbiFlow toolchain can be divided into its three main steps: synthesis, place & route, and bitstream generation. Figure 5 provides a detailed view of how the tools accomplishing these tasks are able to communicate with one another by displaying their intermediate files. Further, it introduces the notion of the architecture file and routing resource graph. These two files and their importance for the addition of DSP blocks will be the focus of Chapter 3. In the following sections, each of these tools will be viewed in detail.

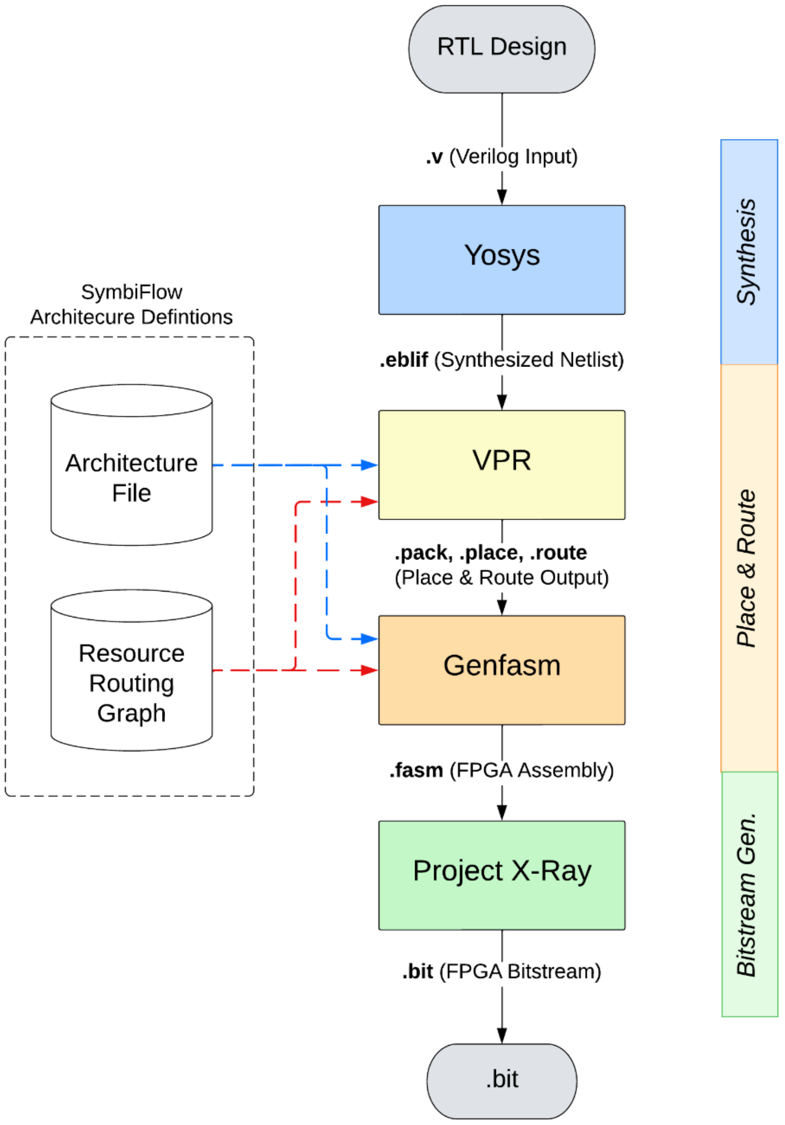


**Figure 5** - Detailed SymbiFlow toolchain with architecture definitions and intermediate files

## 2.1 Yosys

Yosys [15], [19], is an open-source synthesis tool developed by Claire Wolf. It is built as a framework around the University of California Berkeley’s ABC [20] sequential synthesis and verification tool, providing several additional features which support mapping to a variety of popular FPGA families. For the purposes of this work, we use Yosys to take in Verilog-2005 register-transfer level (RTL) code and produce a synthesized netlist in lookup tables (LUTs), memory-based truth tables. Yosys is the first of three tools making up the SymbiFlow toolchain [8]. In SymbiFlow, Yosys synthesis is conducted in two passes to generate an extended BLIF file (EBLIF), which is subsequently passed into the place and route tool VPR.

Synthesis in Yosys is conducted over several passes, some of which include optimization of flip-flops or limiting the design to use LUTs in place of block random access memory (BRAM) modules. It is the subsequent technology mapping and optimizations that lead to the generation of a synthesized netlist from RTL input. When executing Yosys, there are three main commands that are commonly used. They are tailored towards the FPGA architecture that is used within this work. These commands are the following:

- read_verilog
- synth_xilinx
- write_verilog

The “read_verilog” command accepts as input one or more Verilog-2005 RTL files to be synthesized. The “synth_xilinx” command is the Xilinx family specific synthesis command, utilizing Xilinx FPGA cell libraries to create a netlist from the RTL input. Lastly, “write_verilog” instructs Yosys to print out its synthesized netlist for the design.

## 2.2 VPR / Genfasm

During the second stage of the SymbiFlow toolchain, VPR receives the EBLIF netlist output of Yosys to conduct three steps: packing, placement, and routing [21], [22]. The packing step of VPR will use technology mapping to turn the EBLIF netlist into VPR’s own form of netlist [16]. It makes use of the chip's architecture file to understand the ports and interconnects going into and from each tile of the FPGA. The next two steps, placement and routing, are equivalent to the steps in Vivado. The defined logic blocks are then assigned to locations within the chosen FPGA architecture through the placement step. Locations for instantiation can be chosen based on factors such as area or power efficiency. Once placed, VPR makes use of its routing graph to determine the path which signals between logic blocks will take across the FPGA. These paths are directly written as FASM feature statements at the end of VPR through a script called Genfasm. Information involving locations of switchboxes being used and which programmable interconnect points (PIPs) are being activated to route signals are printed to this file. During its runtime, VPR also outputs intermediary files such as .pack. .place, and .route files, which describe logic location and routing assignments, and the execution output log from each VPR step.

VPR generates a FASM file by the end of its execution. Each line of a FASM file is called a feature, and it specifies either a routing connection or other attribute of a specific tile at a specific location on the chip.

Two of the supplemental files needed to run VPR, the routing graph and architecture file for the FPGA, can be generated separately from the SymbiFlow toolchain. The SymbiFlow authors provide a repository through GitHub titled SymbiFlow Architecture Definitions, through which it is possible to contribute information for new families of FPGAs and generate new sets of these files [14], [23]. These files are necessary resources for incorporating DSP blocks into the Artix-7 architecture definitions and subsequent toolchain. The SymbiFlow architecture definitions will be explored further in Chapter 3.

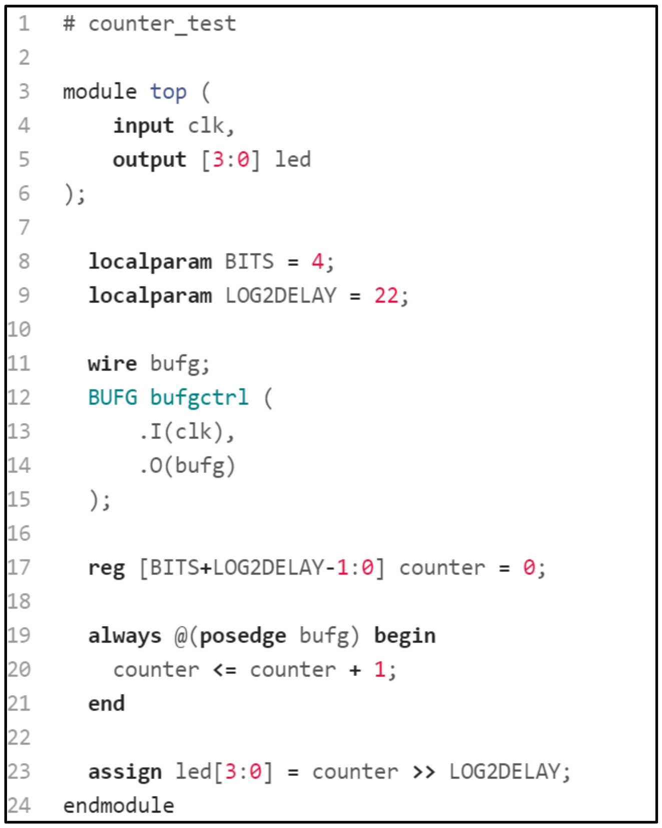
## 2.3 Project X-Ray

Project X-Ray serves as the final step in the SymbiFlow toolchain [12], [24]. It is responsible for taking in the FASM file provided by VPR and Genfasm, producing a bitstream that can be loaded onto the given FPGA. Most of Project X-Ray is a database repository, containing information on possible connections for all supported FPGA families. The Artix-7 portion of this database contains pin maps for many variations of the chip, including the Xilinx Artix-7 xc7a35tcsg324-1 that is used for our testing purposes. To generate a bitstream from the

FASM file, Project X-Ray will consult its list of segbits files [25]. These are a set of databases provided by Project X-Ray containing all possible FASM features for a given tile. The segbits files are necessary to translate FASM features to their appropriate bits in the bitstream. Each connection combination is associated with a bit in the bitstream that will either be toggled on or off depending on whether its feature is present in the FASM file. The fixed length bitstream can then be written by translating the features given to their respective bits from the segbits file before being uploaded to the FPGA.

## 2.4 Using SymbiFlow

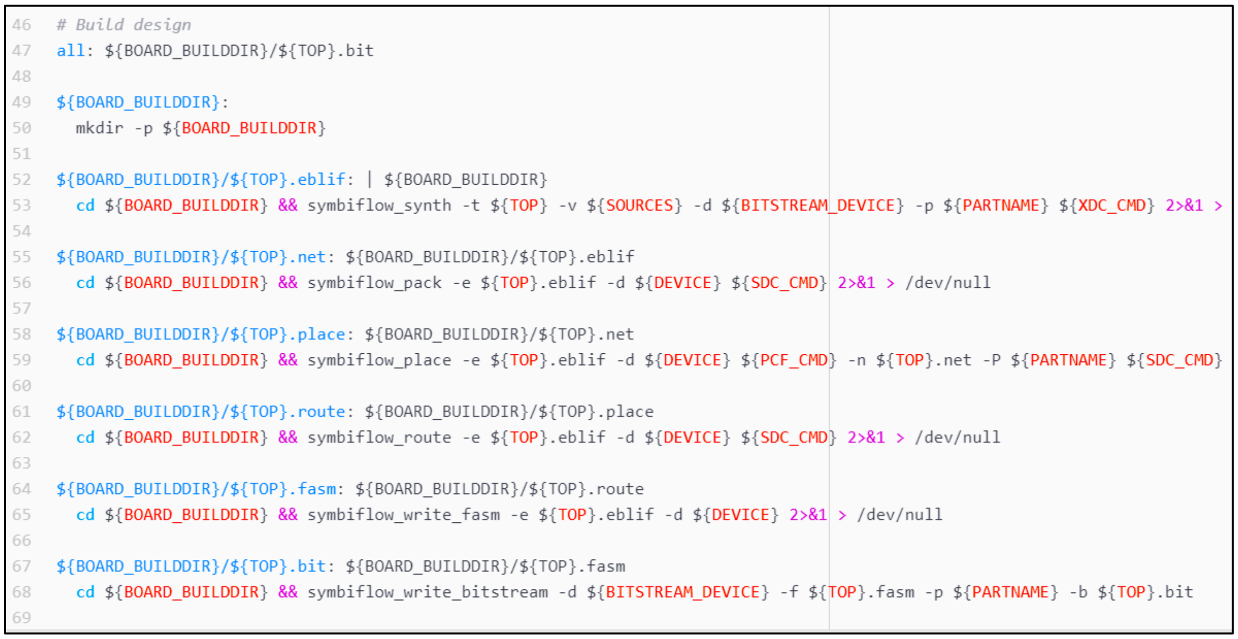
Along with the Yosys, VPR, and Project X-Ray tools, an example directory titled "symbiflow-examples" is provided with SymbiFlow [26]. This includes several designs showcasing how new users can gain familiarity with using the toolchain. One such example is "counter_test", the code for which can be found in Figure 6.



```
# counter_test

module top (
    input clk,
    output [3:0] led
);

  localparam BITS = 4;
  localparam LOG2DELAY = 22;

  wire bufg;
  BUFG bufgctrl (
      .I(clk),
      .O(bufg)
  );

  reg [BITS+LOG2DELAY-1:0] counter = 0;

  always @(posedge bufg) begin
    counter <= counter + 1;
  end

  assign led[3:0] = counter >> LOG2DELAY;
endmodule
```

**Figure 6** - counter_test.v, part of symbiflow-examples (from [26])

The counter_test.v design consists of a clock and buffer with outputs connected to four LEDs. The design is written to count in binary and display the result on the LEDs every clock cycle. It is a simple design, but helpful for understanding how SymbiFlow can take a Verilog RTL file such as counter_test.v and generate a working bitstream for the Artix-7 FPGA.



```
# Build design
all: ${BOARD_BUILDDIR}/${TOP}.bit

${BOARD_BUILDDIR}:
  mkdir -p ${BOARD_BUILDDIR}

${BOARD_BUILDDIR}/${TOP}.eblif: | ${BOARD_BUILDDIR}
  cd ${BOARD_BUILDDIR} && symbiflow_synth -t ${TOP} -v ${SOURCES} -d ${BITSTREAM_DEVICE} -p ${PARTNAME} ${XDC_CMD} 2>&1 >

${BOARD_BUILDDIR}/${TOP}.net: ${BOARD_BUILDDIR}/${TOP}.eblif
  cd ${BOARD_BUILDDIR} && symbiflow_pack -e ${TOP}.eblif -d ${DEVICE} ${SDC_CMD} 2>&1 > /dev/null

${BOARD_BUILDDIR}/${TOP}.place: ${BOARD_BUILDDIR}/${TOP}.net
  cd ${BOARD_BUILDDIR} && symbiflow_place -e ${TOP}.eblif -d ${DEVICE} ${PCF_CMD} -n ${TOP}.net -P ${PARTNAME} ${SDC_CMD}

${BOARD_BUILDDIR}/${TOP}.route: ${BOARD_BUILDDIR}/${TOP}.place
  cd ${BOARD_BUILDDIR} && symbiflow_route -e ${TOP}.eblif -d ${DEVICE} ${SDC_CMD} 2>&1 > /dev/null

${BOARD_BUILDDIR}/${TOP}.fasm: ${BOARD_BUILDDIR}/${TOP}.route
  cd ${BOARD_BUILDDIR} && symbiflow_write_fasm -e ${TOP}.eblif -d ${DEVICE} 2>&1 > /dev/null

${BOARD_BUILDDIR}/${TOP}.bit: ${BOARD_BUILDDIR}/${TOP}.fasm
  cd ${BOARD_BUILDDIR} && symbiflow_write_bitstream -d ${BITSTREAM_DEVICE} -f ${TOP}.fasm -p ${PARTNAME} -b ${TOP}.bit
```

**Figure 7** - common.mk Makefile used to call steps with SymbiFlow (from [26])

SymbiFlow operates from a Makefile which references the tools in the toolchain with their respective commands. A snapshot of the call steps within the *common.mk* Makefile can be seen in Figure 7. The final target "${BOARD_BUILDIR}/${TOP}.bit" to generate a bitstream from the FASM file is called first. This step, as well as all others, has a dependency on the previous step in the flow being completed first. For the ".bit" step, it requires that the ".fasm" step was run to confirm that there is a FASM file to generate a bitstream from. These dependencies cascade down the toolchain until Yosys can be executed to create the synthesized netlist EBLIF file. Table 1 below shows each command found within the file common.mk. In order, they will take in an RTL input to Yosys and put the synthesized netlist through place and route in VPR, while ending with bitstream generation with Project X-Ray. While Project X-Ray has many functions, one of its main tasks is in housing a database of hundreds of FPGA architectures. This tool is used to translate each line of a FASM file to exact bits within a bitstream, determining which features should be set ON or OFF based on entries in its database.

**Table 1** - Steps called and tools used within SymbiFlow

| **Command** | **Description** |
|---|---|
| symbiflow_synth | Calls Yosys to synthesize RTL input into output EBLIF file. Utilizes synth.tcl script to execute Yosys twice |
| symbiflow_pack | Calls VPR to convert EBLIF into a netlist, determining which cell blocks should be used to implement design |
| symbiflow_place | Calls VPR to determine optimal placement location of cells in the netlist |
| symbiflow_route | Calls VPR to route logic between cells in the .place file |
| symbiflow_write_fasm | Calls Genfasm, a tool packaged along with VPR. Takes in the Yosys EBLIF netlist and .pack, .place, and .route files from VPR to create a FASM file |
| symbiflow_write_bitstream | Translates FASM file into a bitstream using Project X-Ray database for the desired FPGA (this project uses the Arty 35T, or xc7a35tcsg324-1 [27]) |

For designs in symbiflow-examples, the SymbiFlow structure will create a “build” directory that contains all intermediate files generated throughout the toolchain.

## 2.5 State of DSP Block Usage in SymbiFlow

Prior to this work, designs using DSP blocks targeted to the Artix-7 family of FPGAs have not been supported in SymbiFlow. While Yosys does offer support for synthesis of DSP blocks, DSP48E1 information is missing from the architecture file and routing graph used within

VPR. The SymbiFlow authors also state that database information for the DSP block has not been fully documented within Project X-Ray as of yet [8]. Due to missing DSP implementation in the architecture file and routing graph, the SymbiFlow TCL script used to execute Yosys includes a “-nodsp” flag throughout to prevent the synthesis of DSP slices during the step. This forces Yosys to use LUT logic in place of a DSP block for multiplication implementation. The Yosys execution TCL script and its associated cell libraries avoid instantiating DSP blocks wherever possible.

In a normal run of SymbiFlow, a second set of cell libraries are used between Yosys synthesis and VPR place and route. The two files, called *cells_sim.v* and *cells_map.v*, are referred to as the VPR libraries. They are necessary to translate the gate level netlist that is output from Yosys onto a set of VPR-readable cells that can be packed, placed, and routed onto the FPGA. The VPR libraries provided with SymbiFlow contain placeholder DSP48E1 cell definitions taken from Xilinx’s publicly available Xilinx Equivalence Checking Library (xeclib) [7]. Without architecture and routing information for the DSP48E1 for VPR, the VPR library definition of the DSP block has nothing to translate onto.

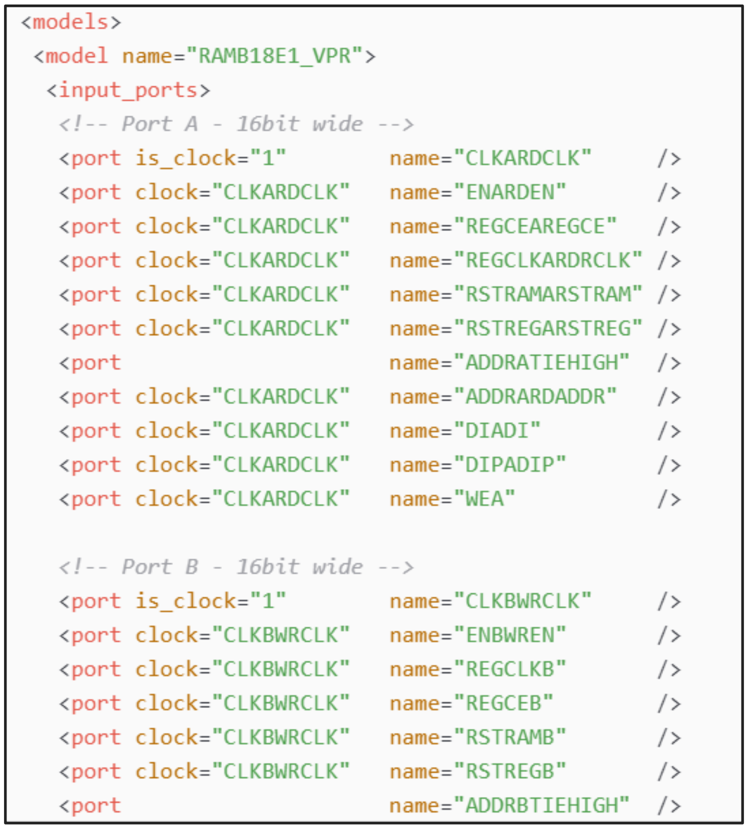
# CHAPTER 3 - ARCHITECTURE DEFINITIONS AND GENFASM

## 3.1 SymbiFlow Architecture Definitions

The SymbiFlow Architecture Definitions [14], [23] (also referred to as symbiflow-arch-defs) are a necessary starting point for incorporating DSP blocks into the Artix-7 flow of SymbiFlow. The repository provides a framework for defining cell primitives that then can produce routing graphs and architecture files for a desired FPGA. The routing graph and architecture file provided with SymbiFlow do not include DSP information at the time of this writing. In the following sections, we will explain how these primitives are used to generate these two files. A design under test will also be used to highlight changes that needed to be made for these architecture definitions to support the DSP48E1 block.

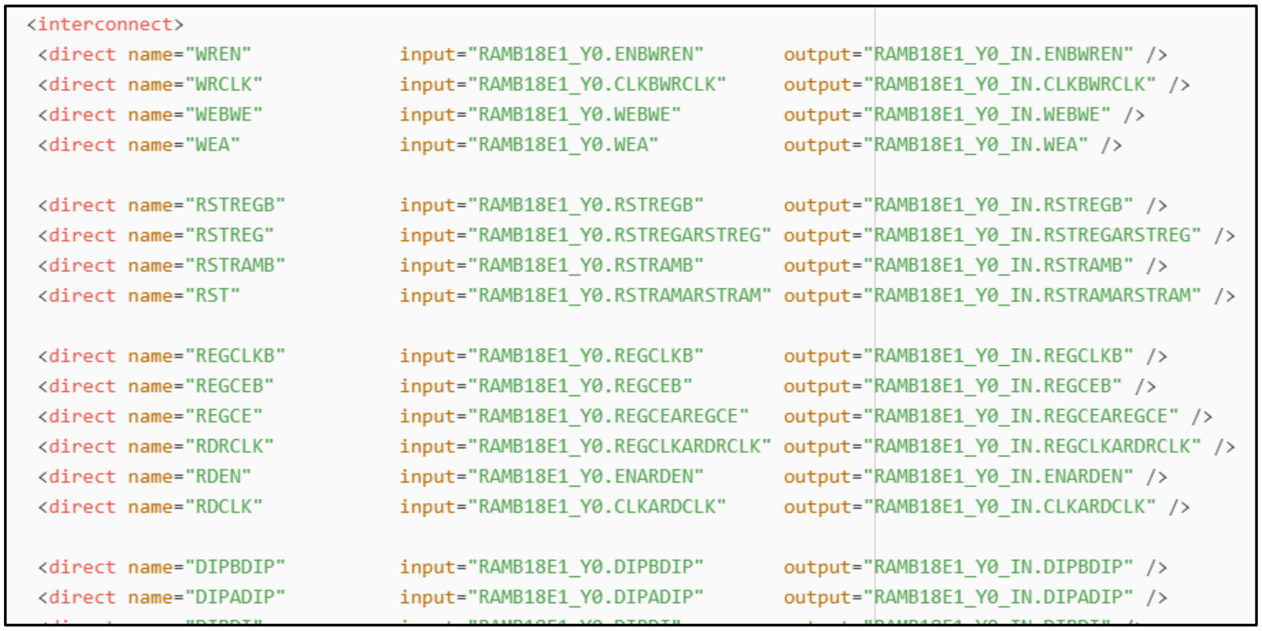
### 3.1.1 Architecture Primitives

FPGA primitives can be thought of as the building blocks for their target architecture. The FPGA is divided into many self-contained tiles with wiring between them, with each tile containing a number of primitives. Current primitives for the Artix-7 family include definitions for the majority of available hardware such as buffers, clocks, LUTs, most BRAMs, and other tiles. Despite each of these cells being very different in function, their primitive definitions are written using the same model. The model involves two files: *.model.xml and *.pb_type.xml. In place of the star, the name of the primitive is placed. For example, there are several different forms of BRAM available on the Artix-7 35T FPGA. One of them is the RAMB18E1. For its primitive definition, its associated files are called ramb18e1.model.xml and ramb18e1.pb_type.xml. The model.xml file describes the input and output ports for the primitive as well as their bit widths. An example of the RAMB18E1 model.xml file is shown in Figure 8. The pb_type.xml file is typically much longer, and this is where interconnects between the tile and the primitive are defined. For tiles that contain multiple types of primitives (such as the BRAM cell), this can lead to having multiple pb_type.xml files that reference each other hierarchically. A look at the interconnects of the RAMB18E1 module can be seen in Figure 9.



```
<models>
 <model name="RAMB18E1_VPR">
  <input_ports>
   <!-- Port A - 16bit wide -->
   <port is_clock="1"         name="CLKARDCLK"     />
   <port clock="CLKARDCLK"    name="ENARDEN"       />
   <port clock="CLKARDCLK"    name="REGCEAREGCE"   />
   <port clock="CLKARDCLK"    name="REGCLKARDRCLK" />
   <port clock="CLKARDCLK"    name="RSTRAMARSTRAM" />
   <port clock="CLKARDCLK"    name="RSTREGARSTREG" />
   <port                      name="ADDRATIEHIGH"  />
   <port clock="CLKARDCLK"    name="ADDRARDADDR"   />
   <port clock="CLKARDCLK"    name="DIADI"         />
   <port clock="CLKARDCLK"    name="DIPADIP"       />
   <port clock="CLKARDCLK"    name="WEA"           />

   <!-- Port B - 16bit wide -->
   <port is_clock="1"         name="CLKBWRCLK"     />
   <port clock="CLKBWRCLK"    name="ENBWREN"       />
   <port clock="CLKBWRCLK"    name="REGCLKB"       />
   <port clock="CLKBWRCLK"    name="REGCEB"        />
   <port clock="CLKBWRCLK"    name="RSTRAMB"       />
   <port clock="CLKBWRCLK"    name="RSTREGB"       />
   <port                      name="ADDRBTIEHIGH"  />
```

**Figure 8** - Snapshot of ramb18e1.model.xml (from [23])



```
<interconnect>
 <direct name="WREN"          input="RAMB18E1_Y0.ENBWREN"       output="RAMB18E1_Y0_IN.ENBWREN" />
 <direct name="WRCLK"         input="RAMB18E1_Y0.CLKBWRCLK"     output="RAMB18E1_Y0_IN.CLKBWRCLK" />
 <direct name="WEBWE"         input="RAMB18E1_Y0.WEBWE"         output="RAMB18E1_Y0_IN.WEBWE" />
 <direct name="WEA"           input="RAMB18E1_Y0.WEA"           output="RAMB18E1_Y0_IN.WEA" />

 <direct name="RSTREGB"       input="RAMB18E1_Y0.RSTREGB"       output="RAMB18E1_Y0_IN.RSTREGB" />
 <direct name="RSTREG"        input="RAMB18E1_Y0.RSTREGARSTREG" output="RAMB18E1_Y0_IN.RSTREGARSTREG" />
 <direct name="RSTRAMB"       input="RAMB18E1_Y0.RSTRAMB"       output="RAMB18E1_Y0_IN.RSTRAMB" />
 <direct name="RST"           input="RAMB18E1_Y0.RSTRAMARSTRAM" output="RAMB18E1_Y0_IN.RSTRAMARSTRAM" />

 <direct name="REGCLKB"       input="RAMB18E1_Y0.REGCLKB"       output="RAMB18E1_Y0_IN.REGCLKB" />
 <direct name="REGCEB"        input="RAMB18E1_Y0.REGCEB"        output="RAMB18E1_Y0_IN.REGCEB" />
 <direct name="REGCE"         input="RAMB18E1_Y0.REGCEAREGCE"   output="RAMB18E1_Y0_IN.REGCEAREGCE" />
 <direct name="RDRCLK"        input="RAMB18E1_Y0.REGCLKARDRCLK" output="RAMB18E1_Y0_IN.REGCLKARDRCLK" />
 <direct name="RDEN"          input="RAMB18E1_Y0.ENARDEN"       output="RAMB18E1_Y0_IN.ENARDEN" />
 <direct name="RDCLK"         input="RAMB18E1_Y0.CLKARDCLK"     output="RAMB18E1_Y0_IN.CLKARDCLK" />

 <direct name="DIPBDIP"       input="RAMB18E1_Y0.DIPBDIP"       output="RAMB18E1_Y0_IN.DIPBDIP" />
 <direct name="DIPADIP"       input="RAMB18E1_Y0.DIPADIP"       output="RAMB18E1_Y0_IN.DIPADIP" />
```

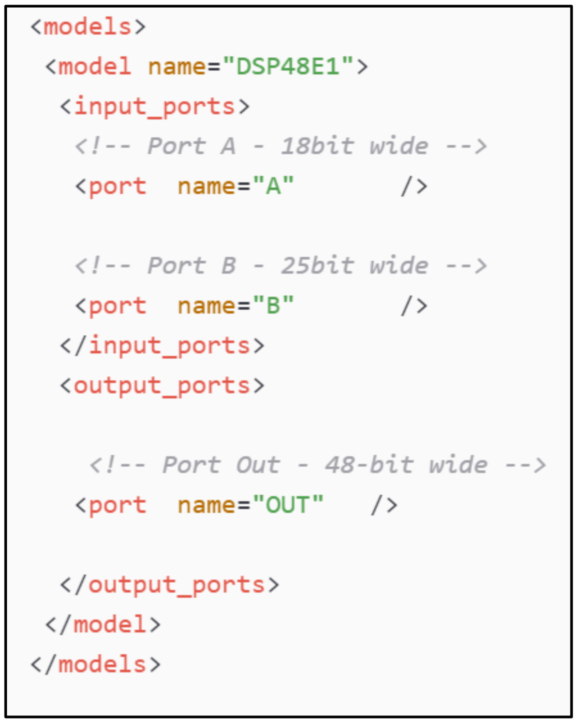
**Figure 9** - Snapshot of bram.pb_type.xml showing RAMB18E1 interconnects (from [23])

### 3.1.2 Generating Routing Graph and Architecture File

The symbiflow-arch-defs repository contains an extensive array of CMake files for automatic routing graph and architecture file generation. After creating all desired primitive and tile definitions for the target architecture, the user can run “`make env`” for the symbiflow-arch-defs directory. This command will create a new build directory that contains copies of the primitive and tile definitions, along with the framework for creating routing graphs and architecture files. By the end of this process, files will have been generated using the primitive and tile definitions for the Artix-7. The most notable of these files for SymbiFlow are the newly created routing graph (rr_graph_xc7a50t_test.rr_graph.real.bin) and architecture file (arch.timing.xml). We found it necessary to convert the routing graph to XML file format using VPR before it is able to be used in the SymbiFlow toolchain. Upon completion, the routing graph and architecture file can be used in place of the original versions provided by SymbiFlow.

### 3.1.3 Developing A Basic DSP48E1 Primitive

The current state of DSP block integration for the architecture definitions and routing graph provided with SymbiFlow is incomplete. This existing primitive for the DSP48E1 contains no routing or interconnect information. Due to this, synthesized DSP blocks being passed from Yosys to VPR cannot be decisively matched with their primitive to conduct place and route, resulting in an error in SymbiFlow. It is apparent that a simple introductory primitive is needed to determine other points within the toolchain that require DSP-related modifications. We begin by creating a new DSP_L tile definition containing DSP48E1 primitives to be included in the symbiflow-arch-defs automated CMake framework. The DSP48E1 primitive outlines the ports and interconnects required to use the DSP block as a basic 25x18 bit multiplier. Our first objective is to define the cell so that VPR can conduct place and route with the tile. We find success in this, with Figure 10 showing the primitive interconnects for the first DSP48E1 to achieve this goal.



```xml
<models>
 <model name="DSP48E1">
  <input_ports>
   <!-- Port A - 18bit wide -->
   <port  name="A"       />

   <!-- Port B - 25bit wide -->
   <port  name="B"       />
  </input_ports>
  <output_ports>

    <!-- Port Out - 48-bit wide -->
   <port  name="OUT"   />

  </output_ports>
 </model>
</models>
```

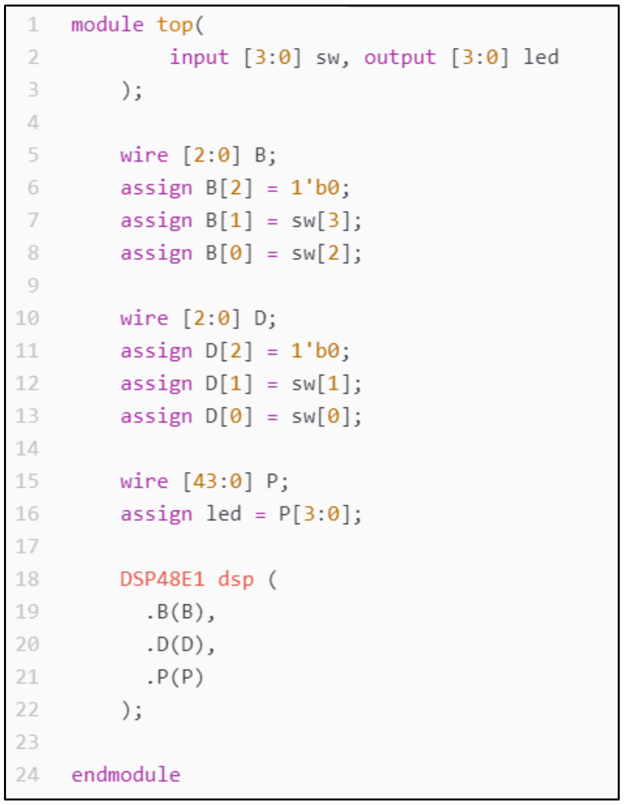
**Figure 10** - Preliminary DSP48E1 primitive used for place and route

This partial primitive definition is successful since VPR conducts routing without knowledge of the DSP_L tile's internals. In fact, the only requirement for our limited DSP48E1 definition is that it matches the VPR library cell definition provided in Yosys. This is necessary for VPR to match the synthesized DSP block to our DSP48E1 primitive. For this design, however, the DSP48E1 instance is blackboxed through Yosys, preventing synthesis on the instance. This provides the benefits of a partial DSP48E1 being provided to VPR that matches its primitive definition. If we were to allow the DSP48E1 to be synthesized in Yosys, full bit widths and a complete definition of the instance would be inferred based on the tool's cell libraries. Synthesizing a full DSP block through Yosys would then prevent us from testing the simple DSP_L tile architecture definition we have created for VPR. While this work later explores how we can include Yosys synthesis for the DSP block, blackboxing the instance for now is beneficial towards designing the DSP48E1 primitive. For the remainder of this chapter, this will continue to be the case.

## 3.2 2x2 bit Multiplier Design Under Test (DUT)

The benchmark chosen to test successful DSP implementation through SymbiFlow is a 2x2 bit unsigned multiplier controlled by input switches on the Arty 35T board. This section will further detail this multiplier design as well as its current FASM representation. A symmetric flow between SymbiFlow and Vivado will be presented to highlight the desired FASM file that will need to be created to generate a functioning bitstream for the design.

### 3.2.1 Symmetric Flow through SymbiFlow and Vivado



```verilog
module top(
        input [3:0] sw, output [3:0] led
    );

    wire [2:0] B;
    assign B[2] = 1'b0;
    assign B[1] = sw[3];
    assign B[0] = sw[2];

    wire [2:0] D;
    assign D[2] = 1'b0;
    assign D[1] = sw[1];
    assign D[0] = sw[0];

    wire [43:0] P;
    assign led = P[3:0];

    DSP48E1 dsp (
      .B(B),
      .D(D),
      .P(P)
    );

endmodule
```

**Figure 11** - Custom 2x2 bit unsigned multiplier using DSP48E1

To introduce a design under test, we create a simple 2x2 bit unsigned multiplier design written in Verilog 2005 that makes use of the DSP48E1 slice. The RTL code used as the input to SymbiFlow is shown in Figure 11. The slice is defined at the chip coordinates X=34, Y=70. The Arty 35T board contains four switches which have each been assigned to one of the four input bits. As the DSP48E1 slice defaults to signed multiplication, the MSB for each input was tied to

1’b0, or GND. This implements a 3x3 bit signed multiplier that functions the same as a 2x2 unsigned multiplier.

While the DUT can be successfully placed and routed through VPR and create a FASM file through Genfasm, the FASM file contains incorrect FASM features that create errors in the bitstream generation step. These errors are a result of having FASM features that are not found in the segbits database mentioned in Chapter 2.4. This means that the preliminary DSP48E1 primitive creates a faulty FASM file that generates features that are not correctly being mapped to bits in the bitstream. To solve this, we need to understand what valid DSP_L FASM features look like. This presents us with a challenge of determining how the completed DSP_L architecture should be generated before continuing to expand upon its tile primitive.

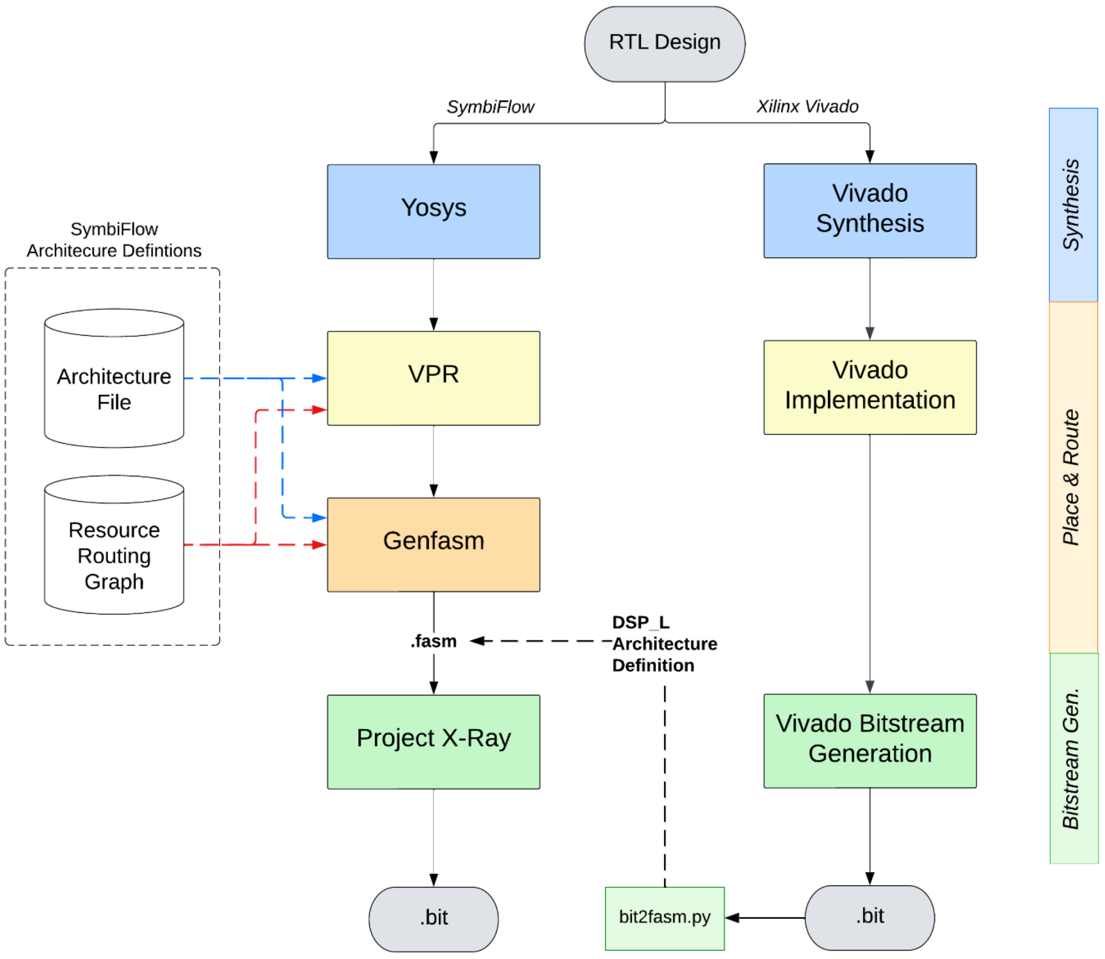
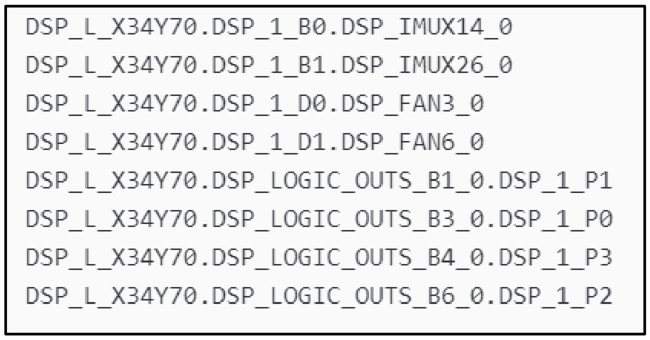


**Figure 12** - Symmetric flow between SymbiFlow and Vivado for 2x2 multiplier. DSP_L tile information is taken from the Vivado FASM file and used in the SymbiFlow FASM file for bitstream generation

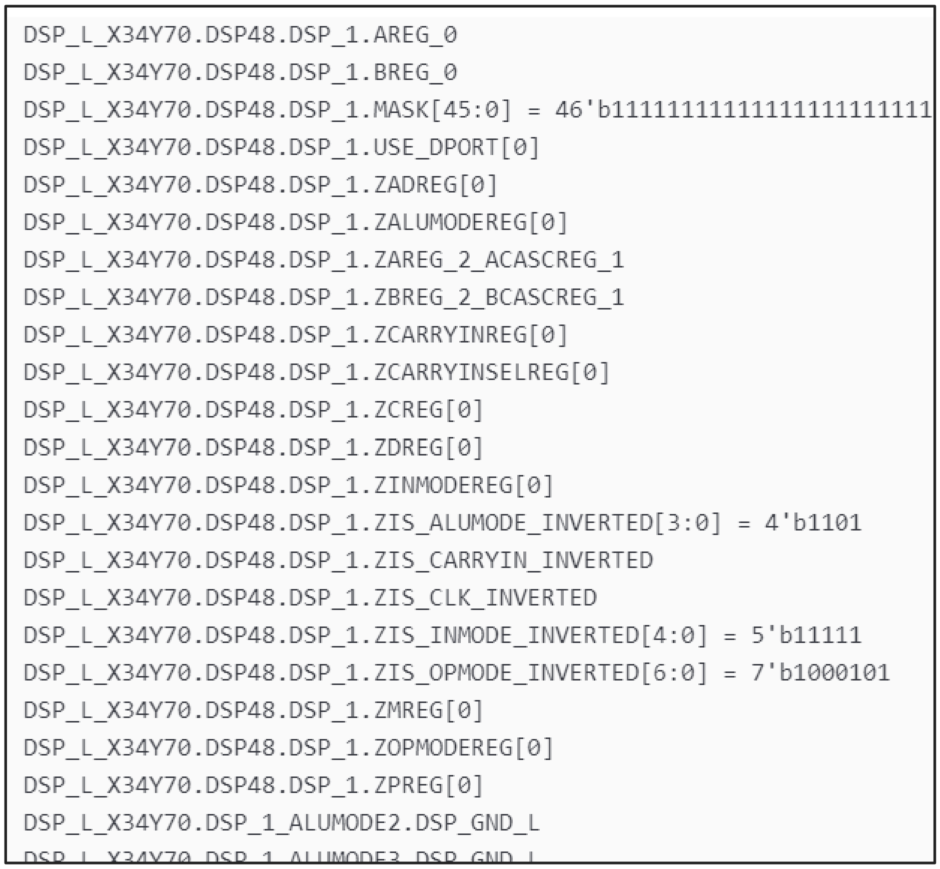
To solve the issue of needing a golden model for the DSP_L FASM information, a symmetrical FPGA synthesis flow is created between SymbiFlow and Vivado. This can be seen in Figure 12. Project X-Ray provides a Python script called bit2fasm.py that can convert an FPGA bitstream into the FASM file that created it. By creating two flows between Vivado and SymbiFlow with the same input RTL Verilog, we are able to compare the FASM file generated by SymbiFlow to the reverse engineered FASM file created from the Vivado bitstream and bit2fasm.py. This gives access to the DSP_L FASM information that we expect to see once the full DSP_L architecture is implemented. In the meantime, however, we are able to confirm the Vivado DSP_L FASM information by substituting only these FASM features into the SymbiFlow FASM file and continue with bitstream generation in Project X-Ray. A functioning multiplier bitstream can be generated using this approach.

### 3.2.2 Differences between SymbiFlow and Vivado FASM files

Snapshots of the DSP_L information from the SymbiFlow and Vivado FASM files are displayed in Figure 13. The SymbiFlow information was generated as a result of our created DSP_L tile definition, where we see the B and D inputs with two bits each along with the four bits of multiplier output to port P. While the full DSP_L information is shown for the SymbiFlow file, only part of the Vivado file is displayed. The Vivado FASM file contains extra information regarding attributes and parameter values for the DSP block, not currently represented in the SymbiFlow tile definition. We see that the binary value 1101 is assigned to the parameter labeled “ZIS_ALUMODE_INVERTED[3:0]”. This, along with two FASM features later in the file setting ALUMODE2 and ALUMODE3 to GND (or 0), are responsible for the signal value of ALUMODE that ultimately enters the DSP48E1 primitive. To understand the final ALUMODE value, it is important to understand this parameter.



```
DSP_L_X34Y70.DSP_1_B0.DSP_IMUX14_0
DSP_L_X34Y70.DSP_1_B1.DSP_IMUX26_0
DSP_L_X34Y70.DSP_1_D0.DSP_FAN3_0
DSP_L_X34Y70.DSP_1_D1.DSP_FAN6_0
DSP_L_X34Y70.DSP_LOGIC_OUTS_B1_0.DSP_1_P1
DSP_L_X34Y70.DSP_LOGIC_OUTS_B3_0.DSP_1_P0
DSP_L_X34Y70.DSP_LOGIC_OUTS_B4_0.DSP_1_P3
DSP_L_X34Y70.DSP_LOGIC_OUTS_B6_0.DSP_1_P2
```



```
DSP_L_X34Y70.DSP48.DSP_1.AREG_0
DSP_L_X34Y70.DSP48.DSP_1.BREG_0
DSP_L_X34Y70.DSP48.DSP_1.MASK[45:0] = 46'b1111111111111111111111
DSP_L_X34Y70.DSP48.DSP_1.USE_DPORT[0]
DSP_L_X34Y70.DSP48.DSP_1.ZADREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZALUMODEREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZAREG_2_ACASCREG_1
DSP_L_X34Y70.DSP48.DSP_1.ZBREG_2_BCASCREG_1
DSP_L_X34Y70.DSP48.DSP_1.ZCARRYINREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZCARRYINSELREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZCREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZDREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZINMODEREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZIS_ALUMODE_INVERTED[3:0] = 4'b1101
DSP_L_X34Y70.DSP48.DSP_1.ZIS_CARRYIN_INVERTED
DSP_L_X34Y70.DSP48.DSP_1.ZIS_CLK_INVERTED
DSP_L_X34Y70.DSP48.DSP_1.ZIS_INMODE_INVERTED[4:0] = 5'b11111
DSP_L_X34Y70.DSP48.DSP_1.ZIS_OPMODE_INVERTED[6:0] = 7'b1000101
DSP_L_X34Y70.DSP48.DSP_1.ZMREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZOPMODEREG[0]
DSP_L_X34Y70.DSP48.DSP_1.ZPREG[0]
DSP_L_X34Y70.DSP_1_ALUMODE2.DSP_GND_L
DSP_L_X34Y70.DSP_1_ALUMODE3.DSP_GND_L
```

**Figure 13** – *(Top)* SymbiFlow FASM information for the DSP_L tile.
*(Bottom)* Vivado FASM information for the DSP_L tile reverse engineered using bit2fasm.py

"ZIS_ALUMODE_INVERTED" is a 4-bit wide signal that connects to multiplexers at the 4-bit ALUMODE input to the DSP48E1. Each bit controls its respective bit's multiplexer in ALUMODE, where a value of 1 will pass the incoming ALUMODE signal through the multiplexer, and a 0 will invert the incoming signal. Therefore, the value "ZIS_ALUMODE_INVERTED[3:0] = 1101" will invert the signal value coming into ALUMODE[1], while remaining bits will be maintained. This, along with two features that set ALUMODE[3:2] to GND, completes the appropriate setup for the ALUMODE signal.

Ultimately, the ALUMODE signal is provided a value of 0001 to perform multiplication for the B and D inputs [17].

While initially seeming redundant, the ”ZIS_ALUMODE_INVERTED[3:0]” parameter is interestingly provided to solve an issue regarding unrouteable VCC or GND signals. For instance, it is often the case that many ports will be connected to GND through the same wire called GFAN in a local switchbox. At times, due to these routed connections becoming numerous within a single switchbox, a *bounce fan* (a routing structure in the switchbox) may be forced to send a VCC and a GND signal due to ports with conflicting values connected to it. The “INVERTED” parameters allow for signal values to be inverted through multiplexers at the input to the DSP48E1 primitive instead of within the switchbox. This serves as a way to lessen congestion of switchbox fan signals.

Not seen in the FASM files from Figure 13, but present in the SymbiFlow and Vivado FASM files, are INT_L features that reference X and Y locations across the chip. These INT_L features are used for defining connections within switchboxes across the FPGA used for signal routing. Unfortunately, the purpose of the INT_L lines in the FASM file was something that was not outlined explicitly in any documentation. It was not until the discovery of the Routing Resources GUI in Vivado that we were able to confirm routing information completed by SymbiFlow. This will be explored further in the following chapter.

With the completion of the basic 2x2 multiplier symmetric flow, we can generate DSP bitstreams through with the help of Vivado FASM information. We gain a deeper understanding of the DSP_L tile attributes through what we have observed as differences between the SymbiFlow and Vivado FASM files. In Chapter 4, we will continue separating the individual aspects of the toolchain that require changes to fully support DSP blocks with strictly the open-source tools.

# CHAPTER 4 – COMPLETE SYMBIFLOW DSP BITSTREAM

## 4.1 Diagnosing DSP Support Points

To this point, we have created a version of SymbiFlow that blackboxes a DSP48E1 instance through Yosys to avoid synthesis. The instance is read into VPR using our basic DSP48E1 primitive and generates a bitstream using the Vivado DSP_L FASM information substituted in place of our created SymbiFlow DSP_L FASM information. This flow is shown in Figure 14. This chapter will return to each of the points that require some form of modification for DSP block support and explain how the change was made. Beginning with RTL, we will trace along the SymbiFlow toolchain, ending with a complete open-source toolchain that can support bitstream generation of a DSP block design for Artix-7 devices.

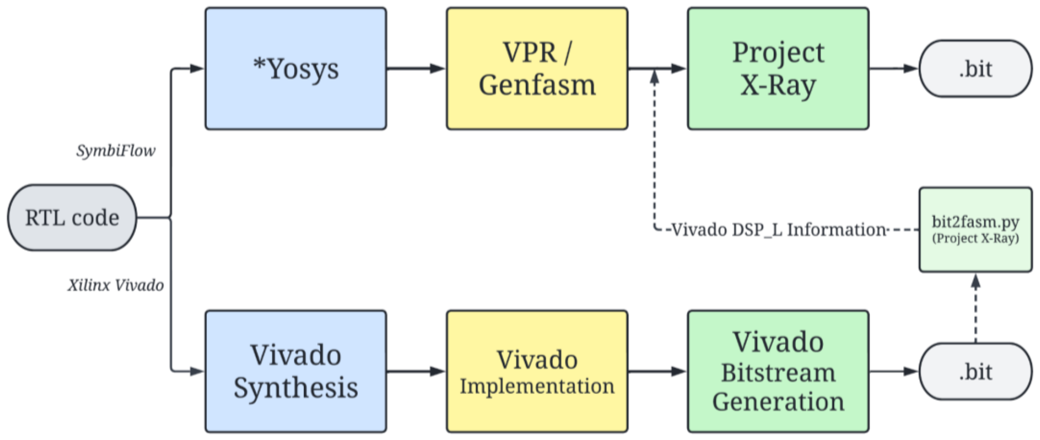


**Figure 14** – 2x2 bit multiplier flow through SymbiFlow (top flow). The bottom flow assists in generating the DSP_L template and is only used once.
*The DSP48E1 is blackboxed through synthesis in order to pass Yosys

## 4.2 Investigating the Vivado Routing Resources GUI

A structural definition of the DSP48E1 instance is provided to SymbiFlow to ensure that the symmetric implementation of the DSP block is identical in both SymbiFlow and Vivado. Using the 2x2 bit multiplier described in Chapter 3.2, we utilize tools within Vivado to determine the value of DSP48E1 parameters and input ports that force the multiplication behavior we need within the tile.

Vivado provides in-depth information regarding its routing network through the Routing Resources GUI view. We can view the routed paths from input pins to the DSP block and from the DSP block to output pins highlighted in green. These represent the logic signals in the design. Figure 15 shows the GUI for Vivado's version of the 2x2 multiplier after place and route.

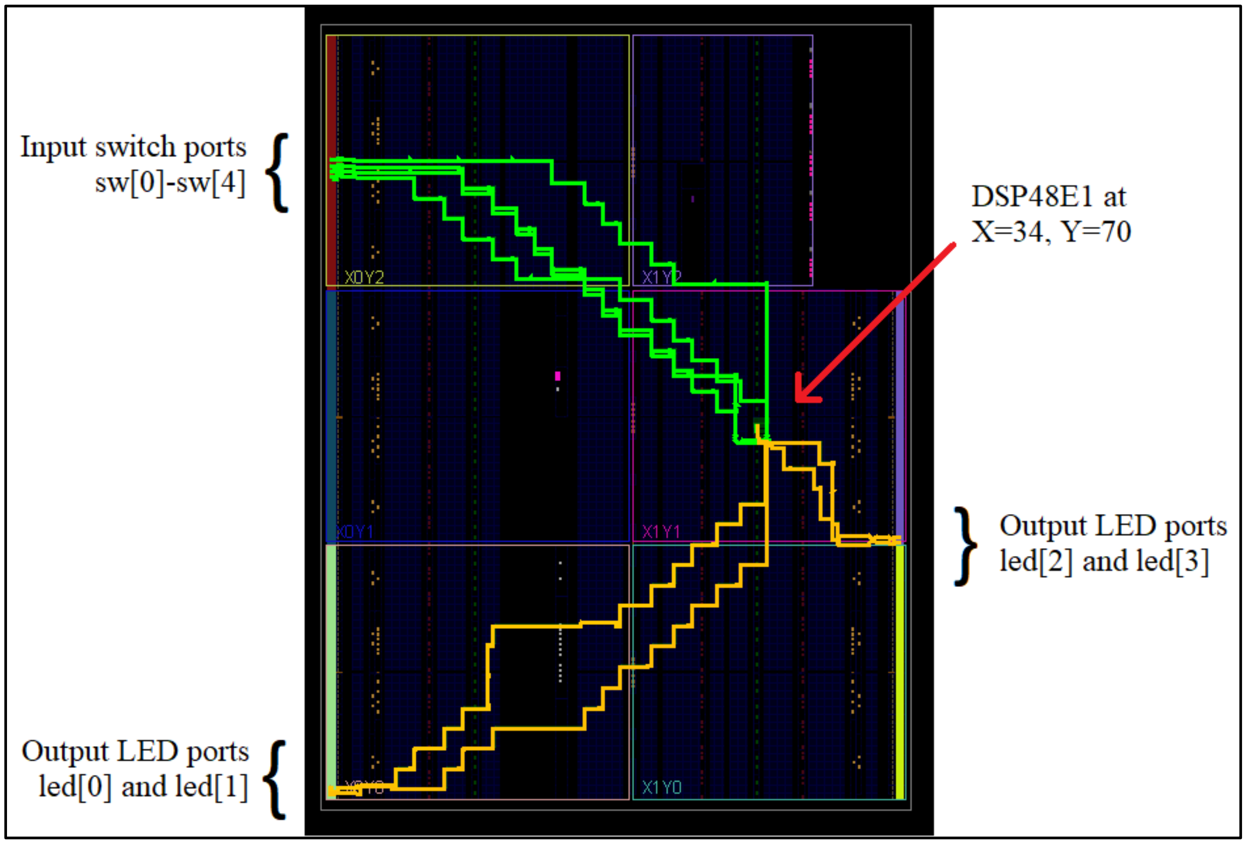
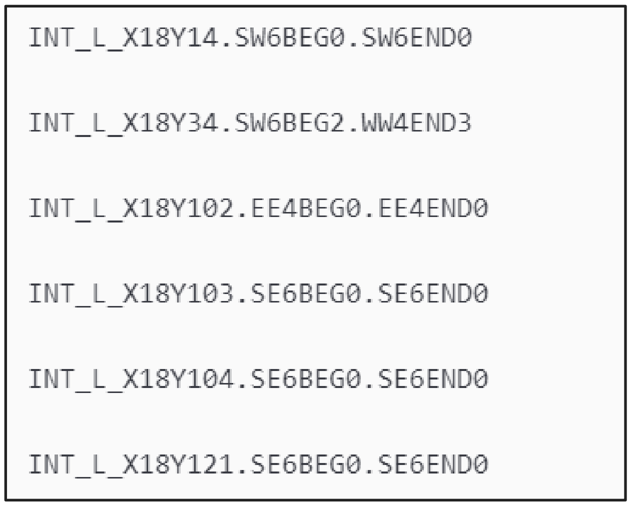


**Figure 15** - Vivado Routing Resources GUI for 2x2 multiplier design. Green signals are for input logic, while orange trace output logic.

Using the Routing Resources view, we were able to determine that many of the lines in the FASM file were for tracing signals through switchboxes across the chip. Previously, the INT_L features had been unknown to us. The Routing Resources GUI allowed us to view switchbox interconnects that held input and outputs that matched those of the switchboxes. Figure 16 shows a comparison between a switchbox interconnect line in the FASM file and its associated connection in the GUI.



```
INT_L_X18Y14.SW6BEG0.SW6END0

INT_L_X18Y34.SW6BEG2.WW4END3

INT_L_X18Y102.EE4BEG0.EE4END0

INT_L_X18Y103.SE6BEG0.SE6END0

INT_L_X18Y104.SE6BEG0.SE6END0

INT_L_X18Y121.SE6BEG0.SE6END0
```

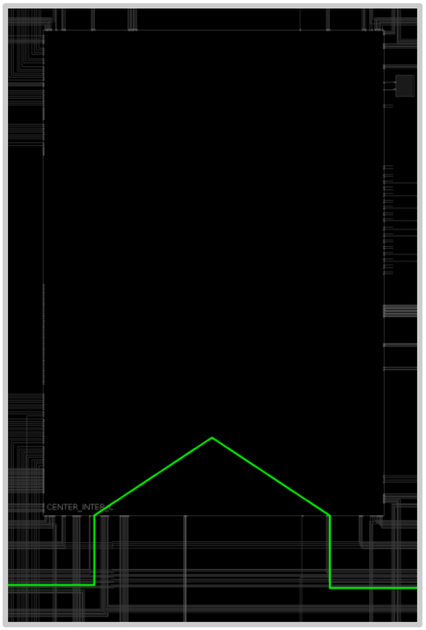


**Figure 16** – *(Left)* INT_L switchbox interconnect lines in Vivado FASM file. *(Right)* Corresponding switchbox pins displayed in the Routing Resources GUI

An important takeaway from consulting the Vivado Routing Resources GUI for this design was not only the switchbox information, but also the ability to view signals that Vivado automatically sets for the DSP block. In order to create a functioning DSP block for Vivado, we used the DSP48 Macro IP Core. IP (Intellectual Property) cores in Vivado are helpful for defining complex blocks without the need to understand the inner workings. For our purposes, the method of hiding instantiation of the DSP block became an obstacle that the Routing Resources GUI allowed us to overcome. This is because the DSP48 Macro ties several of its reset and mode signals to values that force the block to behave as a 2x2 bit multiplier. Table 2 below lists all DSP48E1 ports added to the new routing graph and architecture file that were forced to GND as done by Vivado. Pins not included in this table are automatically set to VCC by SymbiFlow.

**Table 2** - List of DSP48E1 ports tied to GND needed to produce a working 2x2 multiplier

| **Pin Name** | **Pin Description (from [17])** | **Set Externally?** |
|---|---|---|
| A[2:29] | A input to pre-adder | Y |
| B[2:17] | B input to 25x18 multiplier | Y |
| D[0:24] | D input to pre-adder | N |
| OPMODE[6] | X, Y, Z routing multiplexer control | N |
| ALUMODE[2:3] | Selects function of main logic unit | N |
| CARRYINSEL[0:1] | Selects the carry source | Y |
| INMODE[0],[4] | Selects function of pre-adder and multiplier | N |
| CEA1 | Clock enable for the first A register | Y |
| CEA2 | Clock enable for second A register | Y |
| CEB1 | Clock enable for first B register | Y |
| CEB2 | Clock enable for second B register | Y |
| CEC | Clock enable for C register | Y |
| CED | Clock enable for D register | N |
| CEM | Clock enable for post-multiply M register | Y |
| CEP | Clock enable for P register | Y |
| CEAD | Clock enable for pre-adder output AD | N |
| CEALUMODE | Clock enable for ALUMODE registers | N |
| CECTRL | Clock enable for OPMODE and CARRYINSEL | Y |
| CECARRYIN | Clock enable for CARRYIN register | Y |
| CEINMODE | Clock enable for INMODE registers | N |
| RSTA | Reset for both A registers | Y |
| RSTB | Reset for both B registers | Y |
| RSTC | Reset for C register | Y |
| RSTD | Reset for D register | N |
| RSTM | Reset for M register | Y |
| RSTP | Reset for P register | Y |
| RSTCTRL | Reset for OPMODE and CARRYINSEL | Y |
| RSTALLCARRYIN | Reset for internal carry and CARRYIN register | Y |
| RSTALUMODE | Reset for ALUMODE registers | Y |
| RSTINMODE | Reset for INMODE registers | Y |

The view of the Routing Resources GUI of these signals being tied to ground can be seen below in Figure 17. Signals highlighted in blue are tied to GND, while red signals are connected to VCC.

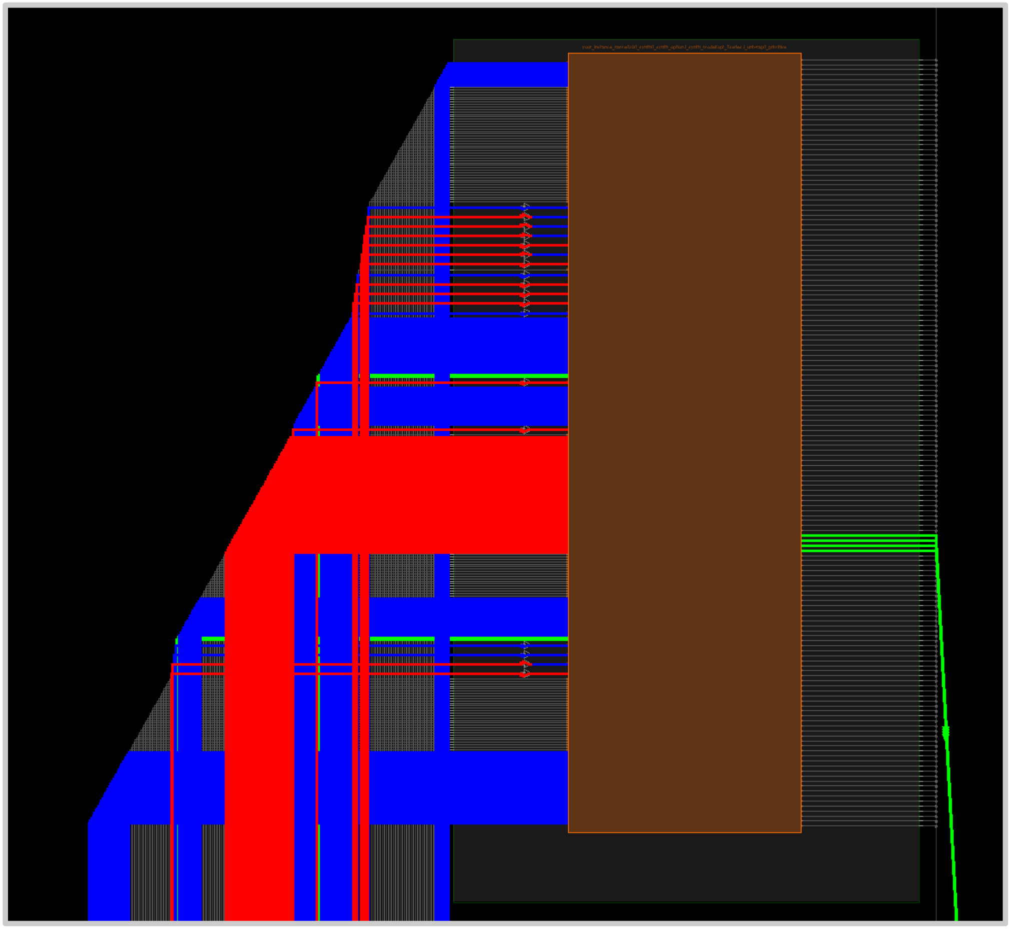


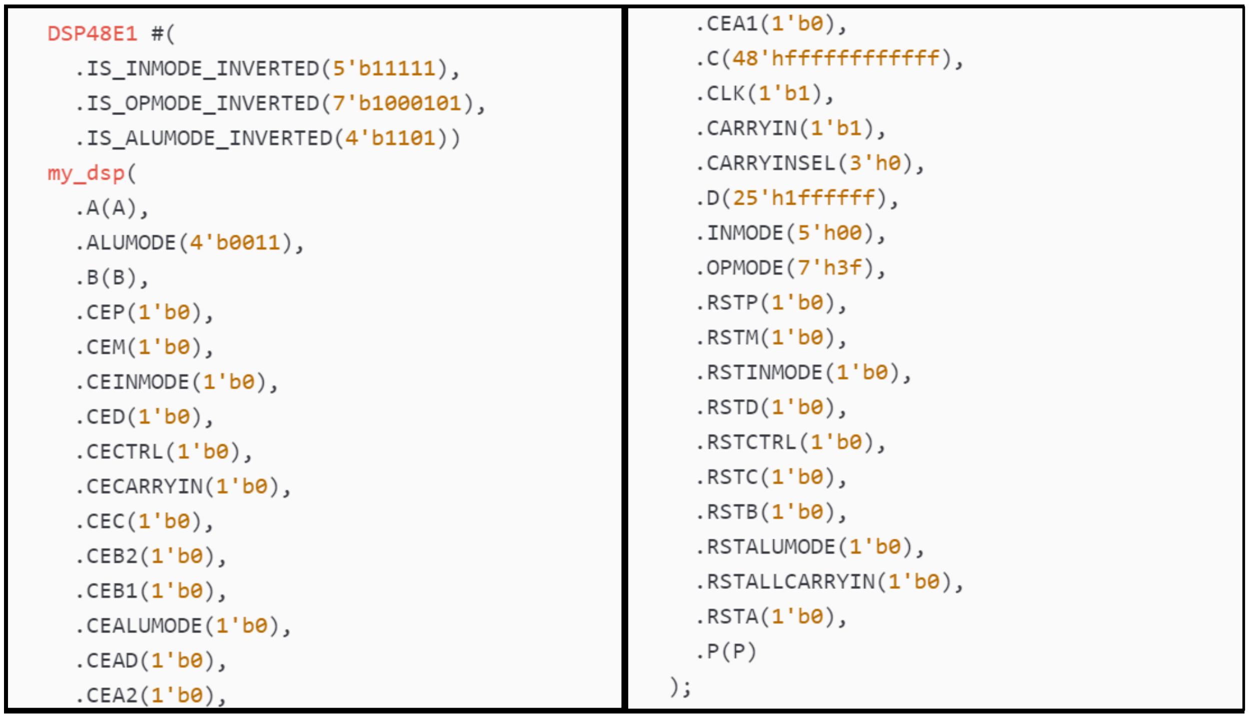
**Figure 17** - Routing Resource GUI view of the DSP block with incoming/outgoing signals. Red signals are VCC, blue are GND. Green represents switches and LEDs (logic)

Viewing signal values headed into the DSP block also granted us access to necessary values for INMODE, OPMODE, and ALUMODE. These signals, discussed in Section 1.3, are critical as their values configure the functionality of the DSP block. The values required to force the DSP48E1 to behave as a multiplier for the A and B input ports are listed in the final column of Table 3. Input signal values multiplexed by the “IS_INVERTED” attribute are provided as well. As a reminder, 0 values in the “IS_INVERTED” attribute will invert the corresponding input signal bit before being fed into the DSP48E1.

**Table 3** - Combination of input signals and inverted attributes on INMODE, OPMODE, and ALUMODE to induce an A*B port multiplier for DSP48E1

| | **Input Signal** | **"IS_INVERTED" Attribute** | **Value Entering DSP48E1** |
|---|---|---|---|
| **INMODE** | 5'b00000 | 5'b11111 | 5'b00000 |
| **OPMODE** | 7'b0111111 | 7'b1000101 | 7'b0000101 |
| **ALUMODE** | 4'b0011 | 4'b1101 | 4'b0001 |

The collection of ports that the Vivado DSP48 Macro ties to GND automatically when defining an A*B DSP multiplier needs to be set manually for SymbiFlow. To do this, we provide the values set by Vivado in the structural definition of the DSP48E1 in RTL code. This definition can be seen in Figure 18. The A and B inputs can be seen connected to the A and B inputs of the DSP48E1 instance, with our discovered values for INMODE, OPMODE, and ALUMODE set from the Vivado GUI and "IS_INVERTED" values set based on the Vivado FASM file.



```
DSP48E1 #(
  .IS_INMODE_INVERTED(5'b11111),
  .IS_OPMODE_INVERTED(7'b1000101),
  .IS_ALUMODE_INVERTED(4'b1101))
my_dsp(
  .A(A),
  .ALUMODE(4'b0011),
  .B(B),
  .CEP(1'b0),
  .CEM(1'b0),
  .CEINMODE(1'b0),
  .CED(1'b0),
  .CECTRL(1'b0),
  .CECARRYIN(1'b0),
  .CEC(1'b0),
  .CEB2(1'b0),
  .CEB1(1'b0),
  .CEALUMODE(1'b0),
  .CEAD(1'b0),
  .CEA2(1'b0),
  .CEA1(1'b0),
  .C(48'hffffffffffff),
  .CLK(1'b1),
  .CARRYIN(1'b1),
  .CARRYINSEL(3'h0),
  .D(25'h1ffffff),
  .INMODE(5'h00),
  .OPMODE(7'h3f),
  .RSTP(1'b0),
  .RSTM(1'b0),
  .RSTINMODE(1'b0),
  .RSTD(1'b0),
  .RSTCTRL(1'b0),
  .RSTC(1'b0),
  .RSTB(1'b0),
  .RSTALUMODE(1'b0),
  .RSTALLCARRYIN(1'b0),
  .RSTA(1'b0),
  .P(P)
);
```

**Figure 18** - 2x2 bit multiplier RTL code with MODEs and INVERTED parameters
*(Images presented in book format for readability)*

## 4.3 Yosys DSP Information into VPR

Yosys will infer information left out of RTL design file definitions. For example, when provided the incomplete DSP48E1 instance in the previous 2x2 multiplier example, Yosys expands the bit widths of each input to match the full size of their port. The A and B inputs of the DSP48E1 were used, leading Yosys to define the 2x2 inputs as 25x18 instead. This led to issues with our testing of a limited DSP block in VPR. We created a DSP48E1 primitive that expects a 2x2 input, which in reality receives a full 25x18 bit input due to Yosys inference. Due to this issue, VPR cannot associate the Yosys DSP block with our DSP48E1 primitive and outputs an error. To combat this, we have blackboxed the DSP48E1 instance through Yosys so that it is provided to VPR in a form that is exactly what we need.

We solve the issue of removing the Yosys blackbox by making modifications to the DSP48E1_VPR definition in the VPR cell libraries. As mentioned in Chapter 2.5, the two files critical for this are *cells_sim.v* and *cells_map.v*. These libraries are structured such that when instantiating a DSP48E1 in *cells_map.v*¸ a call to *cells_sim.v* is made to instantiate an instance of the DSP cell found in that file. This is to say that *cells_map.v* is a wrapper to *cells_sim.v*. Provided with SymbiFlow are DSP48E1 definitions taken from the set of Xilinx cell libraries called xeclib [7]. While this has all the available ports and DRC checks needed for the cell, it does not match our limited partial definition of the instance found in the DSP48E1 primitive. This is corrected over several steps.

The first modification needed is to solve the issue of redundant DSP logic definition. The current set of definitions in *cells_map.v* and *cells_sim.v* sees a call to *cells_map.v*, which creates the logic of a DSP block and then internally references the DSP definition found within *cells_sim.v*, creating a set of logic. This is a result of the placeholder definition for the DSP48E1. From this, attempting to synthesize a DSP block with the base VPR libraries results in a DSP48E1 and matching logic in the form of LUTs being instantiated. We are able to edit the *cells_map.v* definition to match more closely to other cells within its library by removing redundant DRC checks and Verilog logic assignment. Doing so prevents the extra set of LUTs from being instantiated. Also, to remain consistent with other definitions throughout the VPR libraries, the cell name within the lower *cells_sim.v* file is changed from “DSP48E1” to “DSP48E1_VPR”.

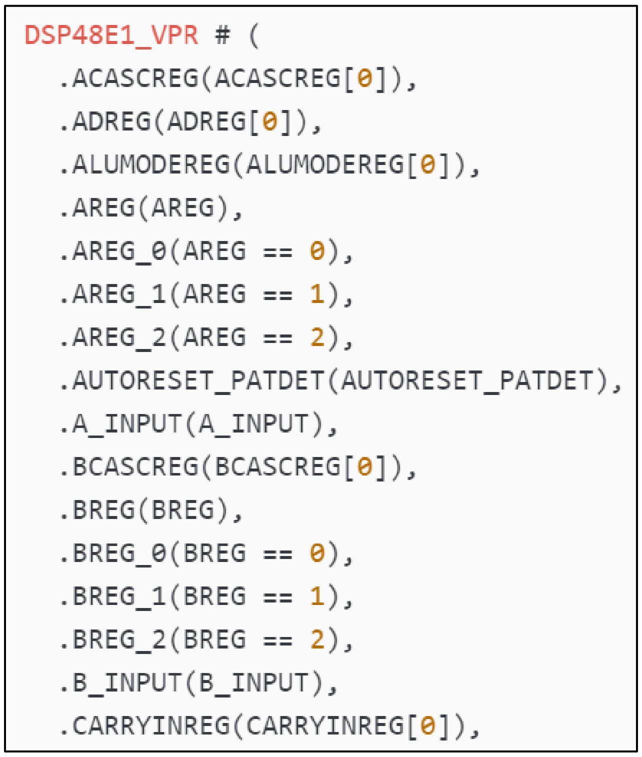
A second change to the VPR libraries consists of trimming its full definition down to only the connections we need to implement the 2x2 multiplier DUT. We use this as a starting point to simplify the flow and demonstrate bitstream generation, with the idea to regrow the DSP48E1_VPR definition as we introduce more functionality to the block. To decide which ports can be left undefined, we refer to the ports left unconnected in the Vivado Routing Resource GUI from the previous section. Table 4 lists all DSP48E1 ports that are removed from the partial definition used in the VPR libraries.

**Table 4 -** All ports removed from the VPR libraries based on Routing Resource GUI

| **Pin Name** | **Pin Description (from** [17]**)** |
|---|---|
| ACIN[0:29] | Cascaded data input from ACOUT of previous DSP48E1 slice (muxed with A). |
| BCIN[0:17] | Cascaded data input from BCOUT of previous DSP48E1 slice (muxed with B). |
| CARRYCASCIN | Cascaded carry input from CARRYCASCOUT of previous DSP48E1 slice. |
| MULTISIGNIN | Sign of the multiplied result from previous DSP48E1 slice for MACC extension. |
| PCIN[0:47] | Cascaded data input from PCOUT of previous DSP48E1 slice to adder. |

The final important piece of information for the VPR cell libraries is the definition of parameters. The EBLIF output of Yosys is used further along the toolchain when writing the FASM features for the DSP_L tile. Within this EBLIF file, we need to ensure that all values for parameters are being written to prepare for FASM file generation. We have seen in the Vivado FASM file reverse-engineered using bit2fasm.py the values that these parameters will need to have, with an example of this being the parameter AREG = 0. Each parameter value is defined of the type “param integer” instead of the common binary value assigned to other parameters throughout the VPR libraries. In fact, this difference in type causes an issue later in the flow, where the 32-bit integer value assigned to each parameter cannot be interpreted by Genfasm, which expects single bit inputs. The implemented fix for this issue is to allow for values to be assigned to integers in the upper *cells_map.v*, while only using their LSB to instantiate the DSP48E1_VPR found within *cells_sim.v*. This way, the binary value being assigned to an integer is returned to binary before being interpreted by Genfasm. While another solution could be to change all data types from integer to binary, this does not solve the issue for parameters like

AREG and BREG that can be assigned to 0, 1, or 2 [17]. These parameters have been broken into three separate attributes (e.g., AREG_0, AREG_1, AREG_2) each assigned a binary value. One and only one of these attributes will have the value of 1'b1 at a given time to represent the three possible values for their combined parameter. Their values are determined using if statements within the DSP48E1_VPR instance definition of *cells_map.v*, seen in Figure 19 below.



```
DSP48E1_VPR # (
  .ACASCREG(ACASCREG[0]),
  .ADREG(ADREG[0]),
  .ALUMODEREG(ALUMODEREG[0]),
  .AREG(AREG),
  .AREG_0(AREG == 0),
  .AREG_1(AREG == 1),
  .AREG_2(AREG == 2),
  .AUTORESET_PATDET(AUTORESET_PATDET),
  .A_INPUT(A_INPUT),
  .BCASCREG(BCASCREG[0]),
  .BREG(BREG),
  .BREG_0(BREG == 0),
  .BREG_1(BREG == 1),
  .BREG_2(BREG == 2),
  .B_INPUT(B_INPUT),
  .CARRYINREG(CARRYINREG[0]),
```

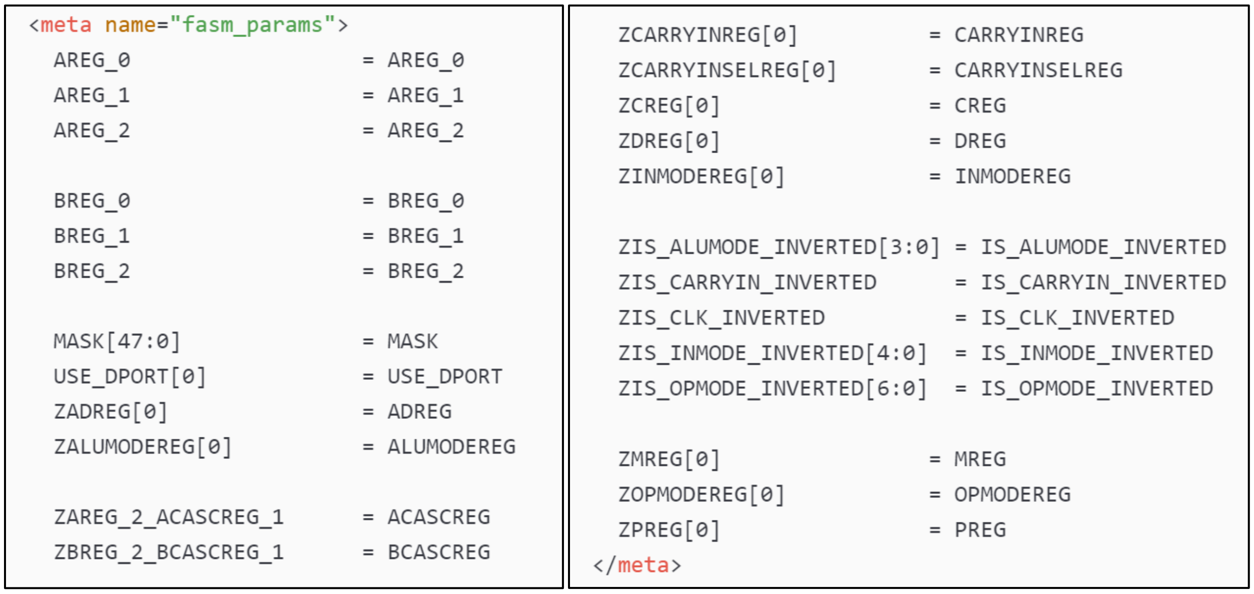
**Figure 19** – Part of *cells_map.v* definition of a DSP48E1_VPR cell. AREG_0, AREG_1, and AREG_2 parameters are added to assist with generating FASM parameters with Genfasm

## 4.4 Developing DSP48E1 Primitive

We know that Yosys has full functionality for synthesizing RTL structures to a DSP48E1 instance. From our previous work, we also have confirmed the DSP_L information required in the FASM file to generate a successful bitstream. This provides us with an input to VPR and its desired output, controlled entirely on the DSP48E1 primitive that we use to build our routing graph and architecture file. We are able to phase out the need for Vivado FASM information for the 2x2 DSP multiplier DUT by defining this primitive correctly. This is achieved by drafting a new DSP_L tile using our desired input and golden output as the framework for testing.

The preliminary DSP48E1 primitive used for place and route was defined with two input ports and a single output port. Using what we learned from the Vivado Routing Resource GUI, we know that we will need to include interconnects for all ports that Vivado implicitly ties to GND. These port definitions, seen previously in Table 2, are added to form a primitive with only the information needed for the multiplier DUT. Remaining ports that do not affect the DSP48E1 multiplier functionality are left undefined.

The most important part of the DSP48E1 primitive is including FASM parameter information, which will translate parameters values defined in the Yosys EBLIF file into FASM. This requires a section of the primitive to indicate how EBLIF parameters should be translated into FASM features. Within the primitive, this section is denoted by the metadata "fasm_params." The Genfasm and VPR authors provide documentation on how to write custom metadata for these parameters as well as writing other FASM features [28], [29]. The custom "fasm_params" section of the DSP48E1 primitive is shown in Figure 20, with the right side of each assignment being the parameter to look for in EBLIF and the left being what should be written in FASM if the parameter is to be set.


```
<meta name="fasm_params">
  AREG_0                    = AREG_0
  AREG_1                    = AREG_1
  AREG_2                    = AREG_2

  BREG_0                    = BREG_0
  BREG_1                    = BREG_1
  BREG_2                    = BREG_2

  MASK[47:0]                = MASK
  USE_DPORT[0]              = USE_DPORT
  ZADREG[0]                 = ADREG
  ZALUMODEREG[0]            = ALUMODEREG

  ZAREG_2_ACASCREG_1        = ACASCREG
  ZBREG_2_BCASCREG_1        = BCASCREG
```

```
  ZCARRYINREG[0]            = CARRYINREG
  ZCARRYINSELREG[0]         = CARRYINSELREG
  ZCREG[0]                  = CREG
  ZDREG[0]                  = DREG
  ZINMODEREG[0]             = INMODEREG

  ZIS_ALUMODE_INVERTED[3:0]  = IS_ALUMODE_INVERTED
  ZIS_CARRYIN_INVERTED       = IS_CARRYIN_INVERTED
  ZIS_CLK_INVERTED           = IS_CLK_INVERTED
  ZIS_INMODE_INVERTED[4:0]   = IS_INMODE_INVERTED
  ZIS_OPMODE_INVERTED[6:0]   = IS_OPMODE_INVERTED

  ZMREG[0]                  = MREG
  ZOPMODEREG[0]             = OPMODEREG
  ZPREG[0]                  = PREG
</meta>
```

**Figure 20** - "fasm_params" section of DSP48E1 primitive
*(Images presented in book format for readability)*

These two steps, implementing the limited port definitions and "fasm_params" for the DSP48E1 primitive, allow SymbiFlow to generate a FASM file with DSP_L features nearly identical to our golden Vivado FASM file. Previously, the reliance on Vivado was to support an incomplete architecture definition for the DSP48E1. This successfully separates the open-source tools from requiring Vivado and the symmetric flow to conduct DSP design bitstream generation.

## 4.5 Changes to Genfasm

As it stands, the FASM file generated by SymbiFlow through VPR will define connections for all ports entering the DSP48E1 through its neighboring switchbox. The Vivado FASM file, when reverse-engineered, omits any reference to signals routed within this switchbox that do not connect to either local VCC or GND. In other words, signal values are represented solely using INT_L or INT_R switchbox features and references to IMUX routing logic is not included in FASM for the DSP_L tile. The Vivado FASM files for multiple designs have been used to regenerate a functioning bitstream when sent through Project X-Ray, leading us to believe that missing signal connections for the local DSP_L switchbox are the default case, or are inferred. The contrast between this and SymbiFlow defining all of its port connections is that Project X-Ray will throw an error when specifying one of these connections in FASM for the D input. Signals attempting to route to any of the D input bits will result in a connection "DSP_FAN" within the DSP switchbox. These connections are unrecognized in Project X-Ray's Artix-7 database, despite being the default value.

Upon further testing, it is discovered that a functional bitstream can be generated when manually removing FASM features mentioning "DSP_FAN" connections for the DSP_L tile. We can easily modify the Genfasm code to only generate FASM features that do not contain this "DSP_FAN" connection. The configurability of open-source tools is demonstrated in full effect, as we can rebuild the binary file for the Genfasm tool and have it swapped in place of the old version provided with SymbiFlow. Doing so, along with implementing all changes detailed in this chapter, results in a modified SymbiFlow toolchain that can successfully generate a bitstream for the 2x2 bit DSP multiplier DUT.

## 4.6 Summary of Modifications

Incorporating DSP block support for Artix-7 devices in SymbiFlow has required modifications to be made throughout the toolchain. This chapter itemized the tools and changes needed to support the new architecture. Changes span as far as the input RTL code, where certain parameter and port values must be specified to emulate the work of Vivado's DSP48 Macro, to Genfasm, where "DSP_FAN" features must be blocked from the output FASM file. Changes are required for Yosys and VPR, where the DSP48E1_VPR cell definition must be limited to match the partial primitive found within VPR's architecture file and routing graph pair. We have shown that enacting these changes allows SymbiFlow to generate a bitstream for the DUT, a 2x2 bit DSP multiplier, previously not possible for Artix-7 devices with the toolchain.

# CHAPTER 5 – EXPANDING DSP FUNCTIONALITY

In the previous chapter, we demonstrate our process for breaking down the larger toolchain into its sub-tools, then continuing further to investigate where DSP support is needed within each sub-tool. This process and the subsequent toolchain modifications have led us to an open-source FPGA toolchain that supports the DSP48E1 instance for usage as a basic 2x2 unsigned multiplier. Chapter 5 uses the development process from the previous chapter to demonstrate use of more complex functions, such as a full 25x18 bit multiplier, partial use of the pre-adder, and pipelining support for the DSP block. The following sections explore each function individually and their unique aspects for integrating support for them within the Artix-7 flow of SymbiFlow.

## 5.1 Full 25x18 bit Multiplier

Implementing the full 25x18 bit multiplier (the largest size supported by Artix-7 DSP blocks) in place of the 2x2 bit does not require any change to the DSP48E1 primitive created before, with all changes happening in RTL. This is because the interconnects have already been defined. When working with a smaller multiplier such as the 2x2 bit, the unused upper bits of each input signal are tied to GND within Yosys when left undefined.

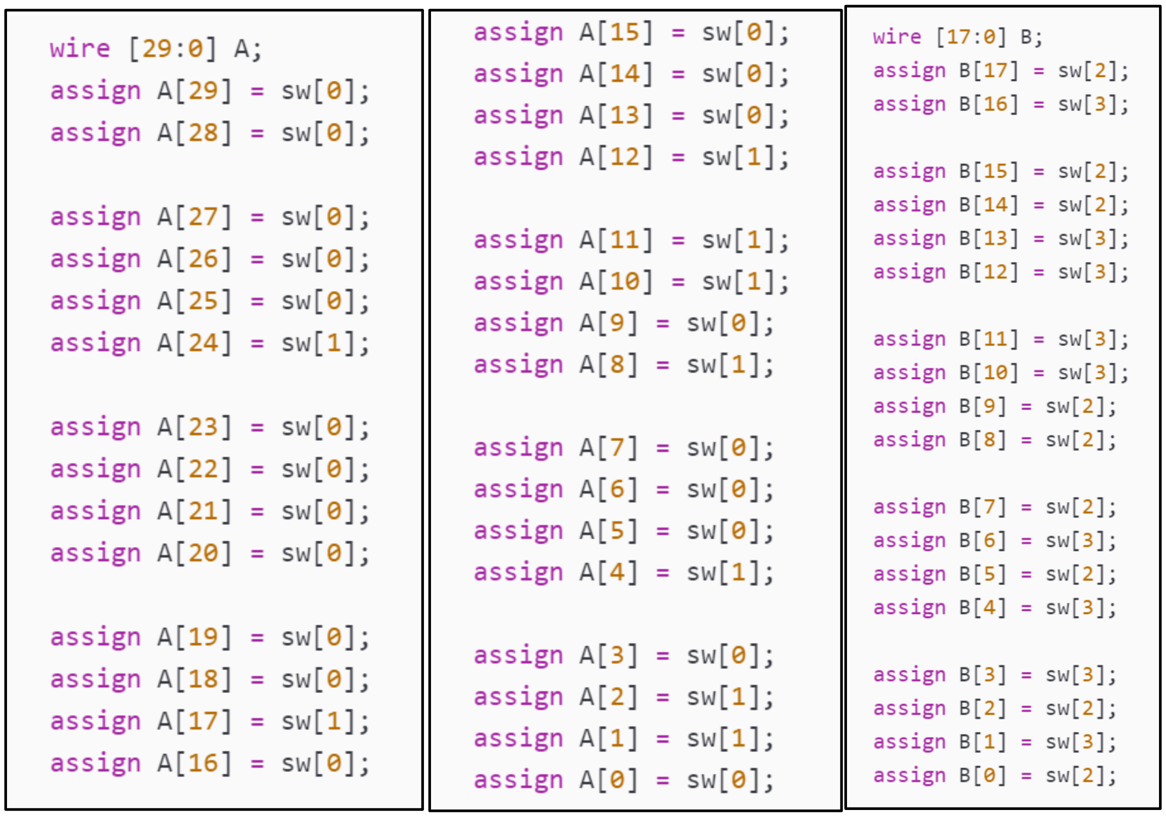
To ensure that no change must happen to the parameter values within RTL, we reimplement the symmetric flow between SymbiFlow and Vivado. Using the Vivado Routing Resource GUI, we were able to determine values for the various input MODE signals that handle routing of data within the DSP48E1. These three signals, INMODE, OPMODE, and ALUMODE, have their values set to those found in Table 4. In a similar way, this same multiplier RTL was put through the Vivado bitstream generation flow and reverse engineered into FASM representation using the script bit2fasm.py. This FASM file gives us the “IS_INVERTED” attributes also found in Table 5. The current support for the DSP48E1 block requires that the instance be defined structurally, meaning these values must be set in the RTL input to SymbiFlow. We can compare these values to those found for the 2x2 multiplier and confirm that there has not been a change.

**Table 5** - MODE signals and their INVERTED parameters needed to use the DSP48E1 as 25x18 bit multiplier

| | **Input Signal** | **"IS_INVERTED" Attribute** | **Value Entering DSP48E1** |
|---|---|---|---|
| **INMODE** | 5'b00000 | 5'b11111 | 5'b00000 |
| **OPMODE** | 7'b0111111 | 7'b1000101 | 7'b0000101 |
| **ALUMODE** | 4'b0011 | 4'b1101 | 4'b0001 |

A change made within the Verilog RTL is to expand bit widths of the A and B inputs to 25 bits and 18 bits, respectively. To properly test this in hardware, we include constraints file definitions for the four green LEDs and four RGB LEDs found on the Arty 35T FPGA. Each green LED can represent 1 bit due to it either being ON or OFF, while each RGB LED can represent 3 bits with each bit controlling either red, green, or blue. In total, this allows us to confirm the 16 least significant bits of the multiplier output in hardware. This configuration does not allow us to test the entire output of the multiplier, however the LEDs can easily be reassigned and separate bits can be tested with the regenerated bitstream.

The Arty 35T FPGA board includes four input switches that are used to control the input bits to the multiplier. Due to hardware restrictions, it is not possible to assign each bit of the 25x18 bit multiplier to a unique input switch. This is solved by randomly assigning input A bits to either the switch 0 or switch 1 signals, and input B bits to either the switch 2 or switch 3 signals. We account for 16 possible input combinations with this method and the ability to reconfigure the switch signals in RTL to be retested in hardware. Figure 21 shows the RTL files used for the 25x18 bit multiplier test, with the following Table 6 listing the LED outputs for all input switch combinations. For the output of RGB LEDs, the first letter of the colors is shown.



```
wire [29:0] A;
assign A[29] = sw[0];
assign A[28] = sw[0];

assign A[27] = sw[0];
assign A[26] = sw[0];
assign A[25] = sw[0];
assign A[24] = sw[1];

assign A[23] = sw[0];
assign A[22] = sw[0];
assign A[21] = sw[0];
assign A[20] = sw[0];

assign A[19] = sw[0];
assign A[18] = sw[0];
assign A[17] = sw[1];
assign A[16] = sw[0];
```

```
assign A[15] = sw[0];
assign A[14] = sw[0];
assign A[13] = sw[0];
assign A[12] = sw[1];

assign A[11] = sw[1];
assign A[10] = sw[1];
assign A[9] = sw[0];
assign A[8] = sw[1];

assign A[7] = sw[0];
assign A[6] = sw[0];
assign A[5] = sw[0];
assign A[4] = sw[1];

assign A[3] = sw[0];
assign A[2] = sw[1];
assign A[1] = sw[1];
assign A[0] = sw[0];
```

```
wire [17:0] B;
assign B[17] = sw[2];
assign B[16] = sw[3];

assign B[15] = sw[2];
assign B[14] = sw[2];
assign B[13] = sw[3];
assign B[12] = sw[3];

assign B[11] = sw[3];
assign B[10] = sw[3];
assign B[9] = sw[2];
assign B[8] = sw[2];

assign B[7] = sw[2];
assign B[6] = sw[3];
assign B[5] = sw[2];
assign B[4] = sw[3];

assign B[3] = sw[3];
assign B[2] = sw[2];
assign B[1] = sw[3];
assign B[0] = sw[2];
```

**Figure 21** - RTL assignment of input switches onto A and B input signal bits

**Table 6** - LED output combinations of 25x18 bit DSP multiplier. RBG values: OFF (000), Blue (001), Green (010), Cyan (011), Red (100), Purple (101), Yellow (110), White (111)

| **sw[3]** | **sw[2]** | **sw[1]** | **sw[0]** | **led[3]** | **led[2]** | **led[1]** | **led[0]** | **RGB led[3]** | **RGB led[2]** | **RGB led[1]** | **RGB led[0]** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | R | Y | Y | G |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | R | OFF | G | G |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | OFF | Y | OFF | P |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | G | OFF | C | Y |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | G | OFF | C | C |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Y | OFF | W | G |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | OFF | B | G | Y |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Y | OFF | P | Y |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | OFF | OFF | OFF | OFF |

## 5.2 Partial Pre-adder

The pre-adder function for the DSP48E1 is located before the 25x18 bit multiplier between the A and D input signals, as seen in Figure 22. The A input signal is 30 bits wide and the D input signal is 25 bits wide, with the output of the pre-adder being a 25-bit wide signal that is fed directly into the 25x18 bit multiplier. In previous experiments with the 25x18 bit multiplier, the value of the INMODE routing signal was set to pass the A input signal through the pre-adder without conducting any addition. We change this by providing a value through the D input and setting INMODE such that the signals will be summed before reaching the multiplier.

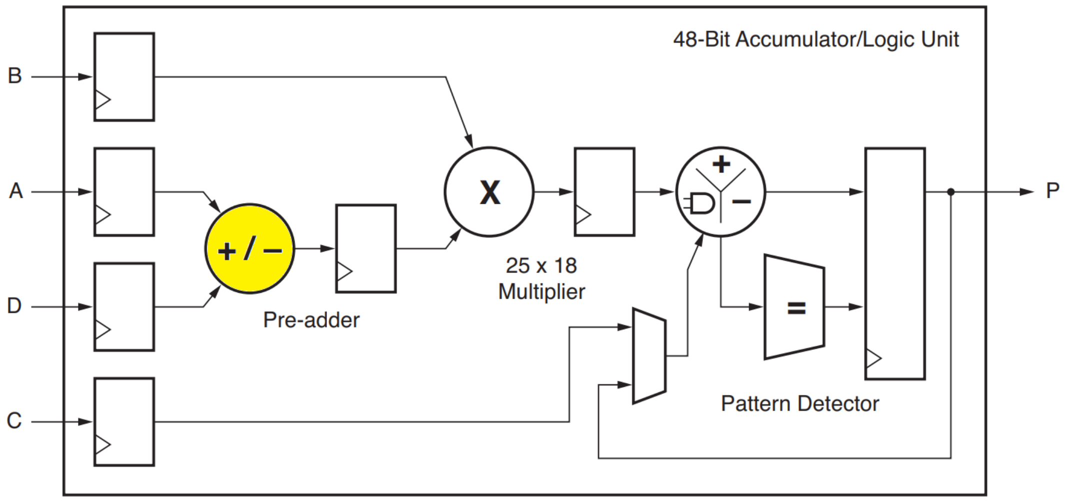


**Figure 22** - Visualization of DSP48E1 internals showing inputs signals A and D entering the pre-adder before reaching the 25x18 bit multiplier (from [17])

Similar to the multiplier, the MODE signal values were determined using the Vivado Routing Resource GUI and a FASM file generated by bit2fasm.py. The values are listed in Table 6 and must be reflected in the input Verilog RTL definition of the DSP48E1. Introducing pre-adder support requires further change to the VPR libraries to facilitate inclusion of the D port. The attribute “USE_DPORT” must be listed as a FASM feature, as shown in the reverse engineered Vivado FASM file. The VPR libraries include this parameter in its default state as a string value assigned to “FALSE”. This is not readable by Genfasm and can be corrected within the *cells_map.v* VPR library file. In the DSP48E1_VPR instantiation with this file, if the

“USE_DPORT” parameter is set to “TRUE”, it is assigned a binary value of 1’b1, otherwise it is assigned a binary value of 1’b0. Doing so allows for translation to FASM using DSP48E1 primitive “fasm_params” statements.

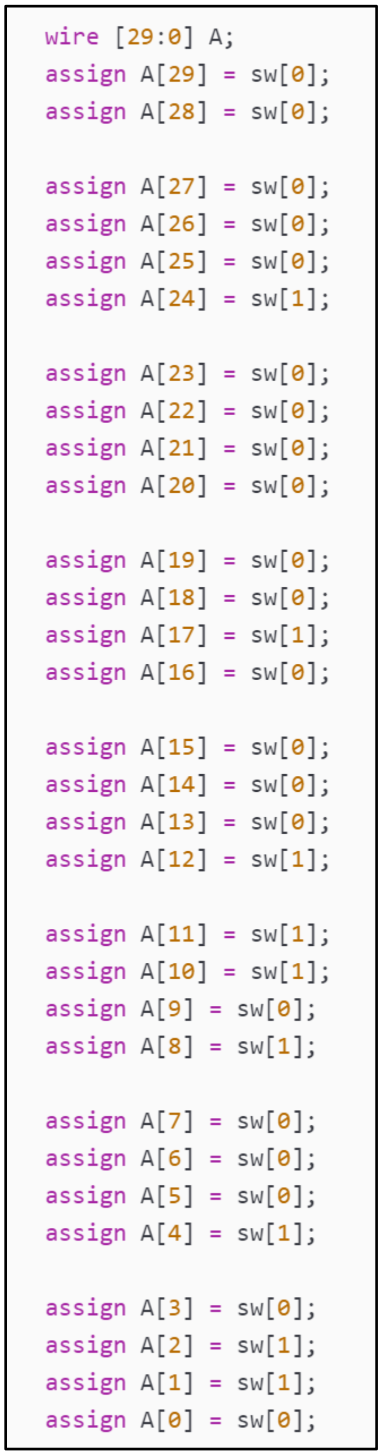
**Table 7** - MODE signals and their INVERTED parameters needed to use the DSP48E1 as a 25x18 bit multiplier with pre-adder

| | **Input Signal** | **“IS_INVERTED” Attribute** | **Value Entering DSP48E1** |
|---|---|---|---|
| **INMODE** | 5’b00100 | 5’b11111 | 5’b00100 |
| **OPMODE** | 7’b0111111 | 7’b1000101 | 7’b0000101 |
| **ALUMODE** | 4’0011 | 4’1101 | 4’0001 |

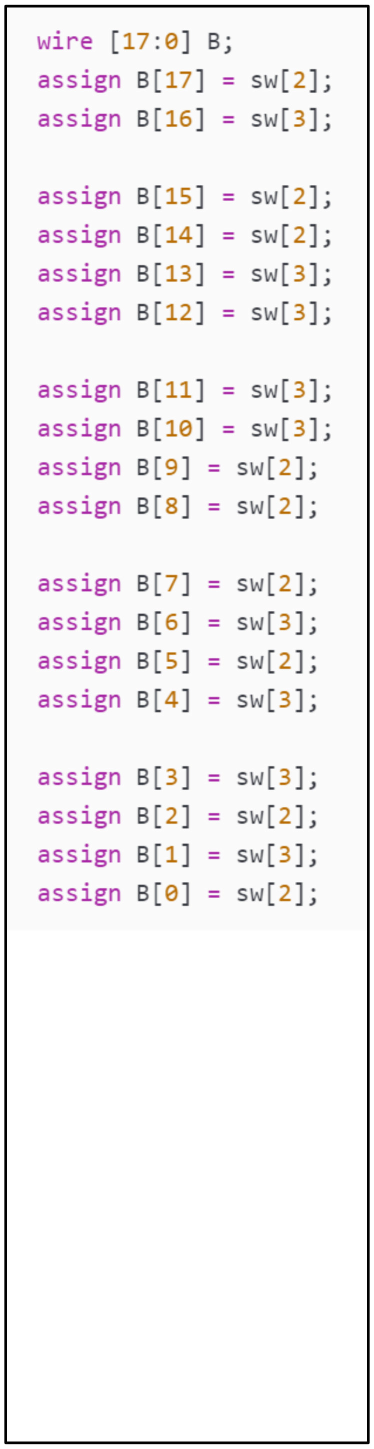
Despite these adjustments  and structurally defining all signal values in RTL, VPR place and route still generated an error for a fully defined D port. When the upper 11 bits of the 25 bit D input are connected to a non-static value, like VCC or GND, VPR is not able to successfully conduct design place and route.

We explain this VPR behavior by analyzing switchbox *fan bounces*. VPR routes signals throughout the FPGA and to the DSP block through switchboxes. A *fan bounce* is a switchbox node that is used to reroute a signal along a path not initially reachable by its switchbox entry node. They are commonly used to provide wide access to local GND connections, increasing the flexibility of signal routing available within the switchbox. We have observed that when using the 11 most significant bits of the D port for pre-adder implementation, *fan bounces* within the switchbox local to the DSP block are assigned to different signals simultaneously. In other words, nodes that are often used as connections to GND are double assigned to incoming switch signal values. The reason for this issue requires further investigation. Here, we only demonstrate using bits 0 through 13 of the D port for a partial pre-adder in successful bitstream generation.

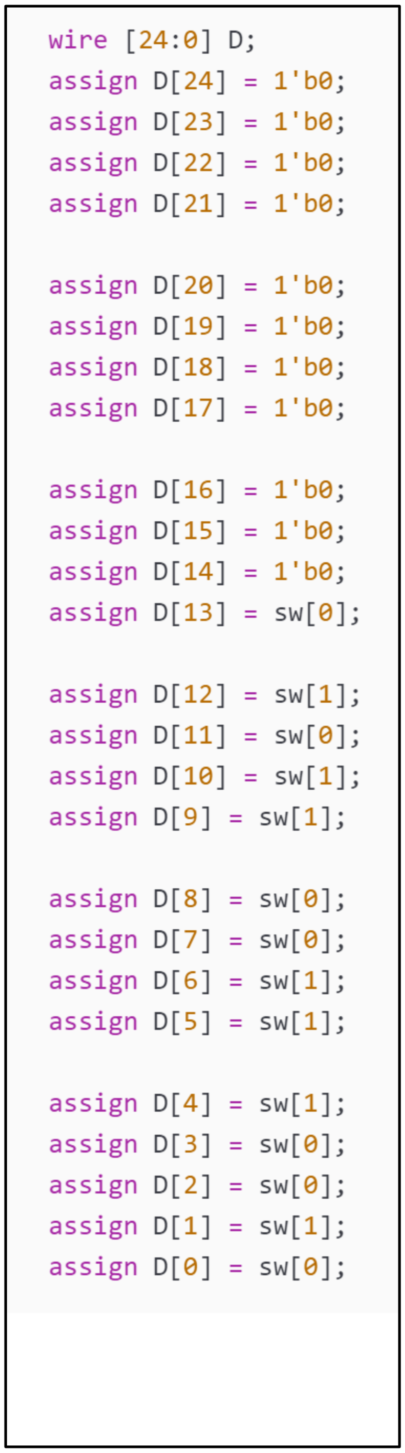
The partial pre-adder implementation for our experiment is created with random switch signals assigned to the first 14 bits of the D input. The Verilog RTL used can be found in Figure 23, with the LED pre-adder and multiplier outputs found in Table 8. The bitstream generated using this configuration was found to operate properly on an Arty 35T board.



```
wire [29:0] A;
assign A[29] = sw[0];
assign A[28] = sw[0];

assign A[27] = sw[0];
assign A[26] = sw[0];
assign A[25] = sw[0];
assign A[24] = sw[1];

assign A[23] = sw[0];
assign A[22] = sw[0];
assign A[21] = sw[0];
assign A[20] = sw[0];

assign A[19] = sw[0];
assign A[18] = sw[0];
assign A[17] = sw[1];
assign A[16] = sw[0];

assign A[15] = sw[0];
assign A[14] = sw[0];
assign A[13] = sw[0];
assign A[12] = sw[1];

assign A[11] = sw[1];
assign A[10] = sw[1];
assign A[9] = sw[0];
assign A[8] = sw[1];

assign A[7] = sw[0];
assign A[6] = sw[0];
assign A[5] = sw[0];
assign A[4] = sw[1];

assign A[3] = sw[0];
assign A[2] = sw[1];
assign A[1] = sw[1];
assign A[0] = sw[0];
```



```
wire [17:0] B;
assign B[17] = sw[2];
assign B[16] = sw[3];

assign B[15] = sw[2];
assign B[14] = sw[2];
assign B[13] = sw[3];
assign B[12] = sw[3];

assign B[11] = sw[3];
assign B[10] = sw[3];
assign B[9] = sw[2];
assign B[8] = sw[2];

assign B[7] = sw[2];
assign B[6] = sw[3];
assign B[5] = sw[2];
assign B[4] = sw[3];

assign B[3] = sw[3];
assign B[2] = sw[2];
assign B[1] = sw[3];
assign B[0] = sw[2];
```



```
wire [24:0] D;
assign D[24] = 1'b0;
assign D[23] = 1'b0;
assign D[22] = 1'b0;
assign D[21] = 1'b0;

assign D[20] = 1'b0;
assign D[19] = 1'b0;
assign D[18] = 1'b0;
assign D[17] = 1'b0;

assign D[16] = 1'b0;
assign D[15] = 1'b0;
assign D[14] = 1'b0;
assign D[13] = sw[0];

assign D[12] = sw[1];
assign D[11] = sw[0];
assign D[10] = sw[1];
assign D[9] = sw[1];

assign D[8] = sw[0];
assign D[7] = sw[0];
assign D[6] = sw[1];
assign D[5] = sw[1];

assign D[4] = sw[1];
assign D[3] = sw[0];
assign D[2] = sw[0];
assign D[1] = sw[1];
assign D[0] = sw[0];
```

**Figure 23** - RTL assignment of input switches onto A, B, and D input signal bits

**Table 8 -** LED output combinations of 25x18 bit DSP multiplier with pre-adder

| sw[3] | sw[2] | sw[1] | sw[0] | led[3] | led[2] | led[1] | led[0] | RGB led[3] | RGB led[2] | RGB led[1] | RGB led[0] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | R | Y | Y | G |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | R | OFF | G | G |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | OFF | Y | OFF | P |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Y | OFF | C | G |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | G | OFF | C | C |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Y | OFF | W | G |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | OFF | OFF | OFF | OFF |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | OFF | Y | G | B |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Y | OFF | P | Y |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | OFF | OFF | OFF | OFF |

## 5.3 Pipelined Multiplier

The DSP48E1 includes registers and routing logic to implement pipelining within the block. Pipelining creates many benefits for a system, including efficiency in timing and power consumption. In hardware, pipelining is implemented through a series of registers between the internal functions of the block, allowing multiple data operations to occur during one clock cycle.

Enabling pipeline support for the DSP48E1 requires several changes to parameter and input signal values not used for either the 25x18 bit multiplier or pre-adder. The MREG and PREG features in the reverse engineered FASM file are removed, indicating that each MREG and PREG should each be assigned one pipeline register. Two additional parameters, AREG and BREG, are set to 2 instead of the default of 1, increasing the number of pipeline registers for the inputs [17], [28]. AREG and BREG values of 2 allow for the use of two input registers. For example, the A input is passed in series through two pipeline registers named A1 and A2. These changes instantiate four stages of pipeline registers through the block. In stages one and two, inputs A and B and loaded into their respective A1/A2 and B1/B2 registers. Stage three saves the output of the multiplier in the M register, which is passed directly to the P register, where is saved once again before output at stage four. This pipelined flow is shown in Figure 24.

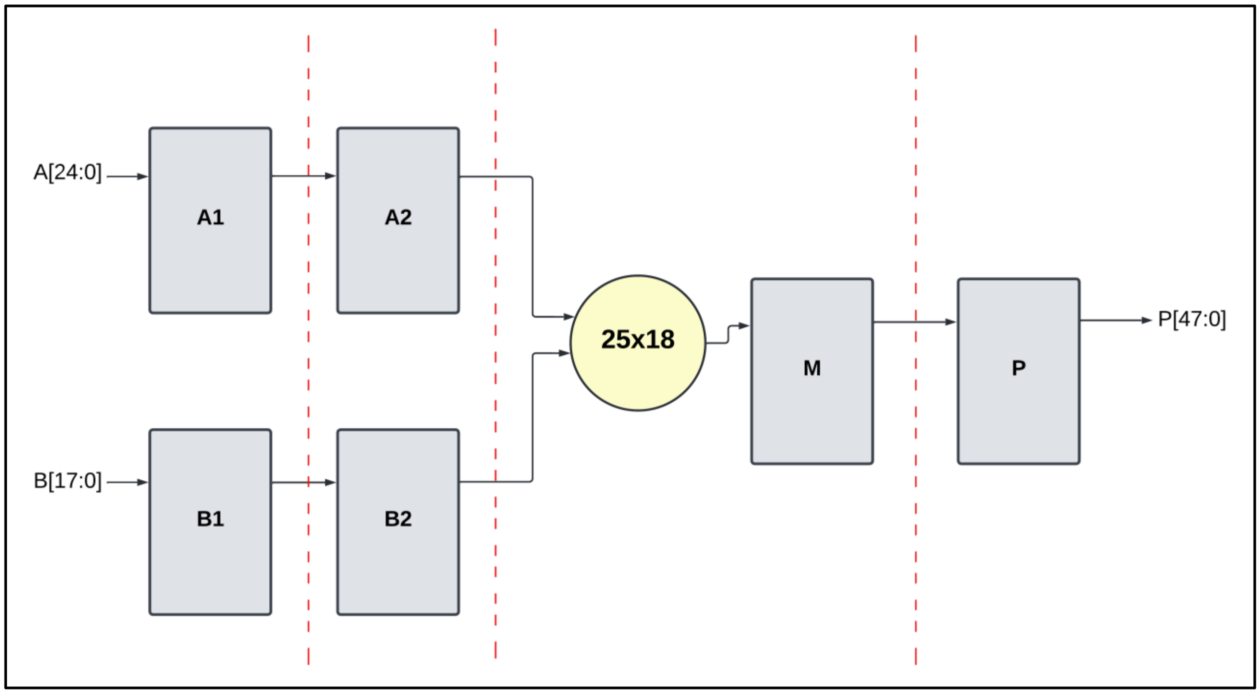
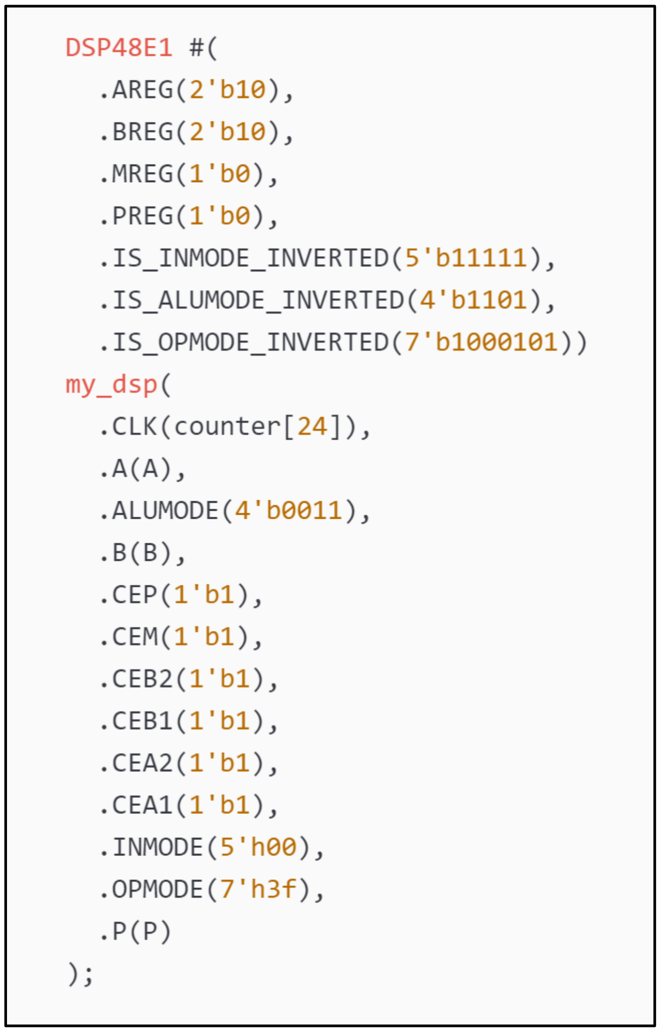


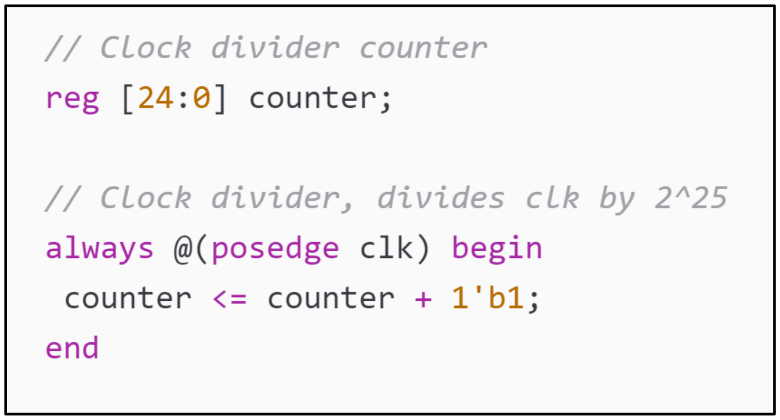
**Figure 24** - Pipelined register flow within the DSP48E1. Two pipelined registers take inputs A and B before storing their multiplied output into register M. Output of register M is stored in register P, then output from the DSP block P port. Red dashed lines indicate pipeline stages

The second asynchronous 25x18 bit multiplier change is to enable the clock inputs to the newly added pipeline registers. These signals, denoted by the prefix “CE”, exist for each of the pipeline registers and are set within the DSP48E1 definition with a value of 1’b1. The modified definition with clock enable signals can be seen in Figure 25. For testing, we added a simple clock divider to the system that divides the main clock frequency by $2^{25}$. This allowed us to visually see the delay in DSP output during testing using the Arty 35T hardware. For on board testing, the previous assignments of switches to the A and B input bits from the 25x18 bit multiplier described earlier in this chapter were used. With a system clock of 100 MHz and a $2^{25}$ times divider, a 3 Hz clock was created to drive the DSP48E1, resulting in several seconds of delay. The clock divider RTL code can be found in Figure 26.



```
DSP48E1 #(
  .AREG(2'b10),
  .BREG(2'b10),
  .MREG(1'b0),
  .PREG(1'b0),
  .IS_INMODE_INVERTED(5'b11111),
  .IS_ALUMODE_INVERTED(4'b1101),
  .IS_OPMODE_INVERTED(7'b1000101))
my_dsp(
  .CLK(counter[24]),
  .A(A),
  .ALUMODE(4'b0011),
  .B(B),
  .CEP(1'b1),
  .CEM(1'b1),
  .CEB2(1'b1),
  .CEB1(1'b1),
  .CEA2(1'b1),
  .CEA1(1'b1),
  .INMODE(5'h00),
  .OPMODE(7'h3f),
  .P(P)
);
```

**Figure 25** - RTL instantiation of a pipelined 25x18 bit multiplier DSP48E1



```
// Clock divider counter
reg [24:0] counter;

// Clock divider, divides clk by 2^25
always @(posedge clk) begin
 counter <= counter + 1'b1;
end
```

**Figure 26** - Clock divider used to drive DSP48E1. 100 MHz system clock "clk" is brought down to a 3 Hz clock signal when “counter[24]” rises from 1'b0 to 1'b1 during counting

The pipelined 25x18 multiplier generated the expected outputs with roughly a single second delay between when the switches change to when an output is shown on the LEDs. It was possible to make multiple changes to the switches before seeing an output change. This was expected due to the 3 Hz clock driving the DSP48E1, which successfully slows down the pipelining to a human visible speed. The 3 Hz signal operating over four stages allows an output to be displayed in roughly 1.3 seconds. Pipelining can be incorporated with the multiplier and partial pre-adder design through the addition of pipelined registers in REG parameters and the setting of clock enable register signals.

# CHAPTER 6 – CONCLUSION

In this thesis document, we presented our modifications to the open-source FPGA bitstream generation toolchain SymbiFlow that allow for basic implementation of a DSP48E1 digital signal processing block in the bitstream of a Xilinx Artix-7 FPGA. SymbiFlow was used to conduct synthesis, place and route, and bitstream generation. Our background work highlighted missing DSP block support in the flow. Code modifications were made in RTL designs, VPR libraries, the DSP48E1 primitive, and the Genfasm tool to support DSP block bitstream generation for an Artix-7 35T FPGA.

The Vivado Routing Resources GUI was used to determine input signal values set within the DSP48 Macro IP core, information that is intentionally obscured in the development environment. These values were then included to form a full structural definition in RTL. Modifications were made to the VPR libraries, where undefined ports were removed to simplify the cell. This limited definition of the DSP block was made to match the newly created DSP48E1 primitive and to match the DSP48E1_VPR library definition used for place and route. The Genfasm tool was lightly modified to allow for DSP bitstreams to be generated for a simple 2x2 bit unsigned multiplier.

Through expansion of the VPR libraries and primitives, a full 25x18 bit multiplier, partial pre-adder, and pipelining are now supported for Artix-7 DSP blocks in SymbiFlow. Designs that use each of these constructs were successfully tested on a Digilent Arty A7 35T board.

## 6.1 Future Work

It is our hope that this work contributes towards future innovation with open-source FPGA tools. In the following subsections, we will describe ways in which this work can be extended towards achieving this overall goal.

### 6.1.1 Incorporating C Port for DSP48E1

Additional SymbiFlow functionality is still needed to support all DSP48E1 block functionality in bitstream generation. With additional edits to the *cells_sim.v* and *cells_map.v* (VPR library files) and the DSP48E1 primitive, SymbiFlow could support C port operations such as pattern detection, 3-input multiplication, or 2-input multiplication with addition [17]. A symmetric flow, similar to the one described in Section 3.2.1, would be a good approach for debugging and testing needed changes.

### 6.1.2 Investigating the D Port Congestion Issue

As discussed in Section 5.3, bits 14 through 25 of the D input port to the DSP48E1 are currently unusable by VPR. The issue may be related to the way the VPR tool routes signals through the neighboring DSP_L tile switchbox. At times, we have noticed that input signal values will overlap with VCC or GND inputs, causing switchbox congestion and leading to the assignment of different values simultaneously to programming pips. Future work should investigate and address this issue.

### 6.1.3 Allowing Behavioral Modeling of DSP48E1

At present writing, DSP blocks created within SymbiFlow must be defined structurally within RTL code to accurately set their parameter values and handle the tying off of control signals to GND and VCC. Yosys and VPR libraries could be expanded to allow for the behavioral modeling of DSP48E1 instances while maintaining necessary parameter values and GND/VCC connections. With this change, Yosys synthesis could map a DSP block with all needed parameters with RTL input such as “A*B”, creating a functioning multiplier without explicitly setting port inputs, outputs, and parameters.

# BIBLIOGRAPHY

[1] J. M. Mbongue, A. Shuping, P. Bhowmik, and C. Bobda, “Architecture Support for FPGA Multi-tenancy in the Cloud,” in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2020, pp. 125–132. doi: 10.1109/ASAP49362.2020.00030.

[2] D. Andrews *et al.*, “Programming models for hybrid FPGA-cpu computational components: a missing link,” *IEEE Micro*, vol. 24, no. 4, pp. 42–53, Jul. 2004, doi: 10.1109/MM.2004.36.

[3] “Xilinx - Adaptable. Intelligent.,” *Xilinx*. https://www.xilinx.com/

[4] “Intel® FPGA Products - FPGA and SoC FPGA Devices and Solutions,” *Intel*. https://www.intel.com/content/www/us/en/products/details/fpga.html

[5] “Lattice Semiconductor | The Low Power FPGA Leader.” https://www.latticesemi.com/en

[6] “Artix-7 FPGA Family,” *Xilinx*. https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

[7] Xilinx Corporation, “Xilinx Vivado,” *Xilinx*. https://www.xilinx.com/products/design-tools/vivado.html

[8] F4PGA, “F4PGA - the GCC of FPGAs,” 2022. https://f4pga.org/

[9] J. J. Rodríguez-Andina, M. D. Valdés-Peña, and M. J. Moure, “Advanced Features and Industrial Applications of FPGAs—A Review,” *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 853–864, Aug. 2015, doi: 10.1109/TII.2015.2431223.

[10] “Project IceStorm.” http://bygone.clairexen.net/icestorm/

[11] “Welcome to Project Trellis — Project Trellis documentation.” https://prjtrellis.readthedocs.io/en/latest/

[12] “Project X-Ray.” F4PGA, Apr. 29, 2022. [Online]. Available: https://github.com/f4pga/prjxray

[13] “QuickLogic - Low Power MCUs, FPGAs and eFPGA IP, & 100% open source tools,” *QuickLogic Corporation*. https://www.quicklogic.com/company/company-overview/

[14] “F4PGA Architecture Definitions documentation.” https://f4pga.readthedocs.io/projects/arch-defs/en/latest/

[15] Claire Wolf, “Yosys Open SYnthesis Suite.” SymbiFlow, Jan. 18, 2022. [Online]. Available: https://yosyshq.net/yosys/

[16] V. Betz and J. Rose, “VPR: a new packing, placement and routing tool for FPGA research,” in *Field-Programmable Logic and Applications*, 1997, pp. 213–222. doi: 10.1007/3-540-63465-7_226.

[17] Xilinx Corporation, “7 Series DSP48E1 Slice User Guide (UG479),” Mar. 2018, [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1

[18] “Vivado Design Suite User: Guide Designing with IP,” p. 114, 2021.

[19] Claire Wolf, “Yosys Open SYnthesis Suite :: Documentation.” https://yosyshq.net/yosys/documentation.html

[20] University of California Berkeley, “ABC: A System for Sequential Synthesis and Verification.” https://people.eecs.berkeley.edu/~alanmi/abc/

[21] “Verilog to Routing (VTR).” Verilog to Routing, Apr. 29, 2022. [Online]. Available: https://github.com/verilog-to-routing/vtr-verilog-to-routing

[22] “VTR — Verilog-to-Routing 8.1.0-dev documentation.” https://docs.verilogtorouting.org/en/latest/vtr/

[23] “F4PGA Architecture Definitions.” SymbiFlow, Apr. 26, 2022. [Online]. Available: https://github.com/SymbiFlow/f4pga-arch-defs

[24] “Project X-Ray 0.0-3575-g3a0602f4 documentation.” https://f4pga.readthedocs.io/projects/prjxray/en/latest/

[25] “segbits files — Project X-Ray 0.0-3587-g9733e8a6 documentation.” https://f4pga.readthedocs.io/projects/prjxray/en/latest/dev_database/common/segbits.html

[26] “Welcome to F4PGA examples! — F4PGA examples documentation.” https://symbiflow-examples.readthedocs.io/en/latest/

[27] “Artix-7 35T Arty FPGA Evaluation Kit,” *Xilinx*. https://www.xilinx.com/products/boards-and-kits/arty.html

[28] “FPGA Assembly (FASM) — FPGA Assembly (FASM) 0.0.2-98-g9a73d70 documentation.” https://fasm.readthedocs.io/en/latest/

[29] “FPGA Assembly (FASM) Output Support — Verilog-to-Routing 8.1.0-dev documentation.” https://docs.verilogtorouting.org/en/latest/utils/fasm/