# ECE532 Project

# Linear Programming Solver

Adham Ragab
Martin Staadecker
Ahmed Hamoda
Abnash Bassi

April 8th, 2024

CONTENTS

## List of Figures

## List of Tables

## 1. Overview

### *A. Background*

Linear programming is a computational technique for optimizing a linear objective function subject to linear equality and inequality constraints. Linear programming solvers are pivotal in many fields, from electricity grid modelling to economics, because they can efficiently navigate and find optimal solutions in complex decision-making scenarios. However, general-purpose CPUs are slow in solving large linear programs with run times sometimes on the order of days, despite using commercial solvers such as Gurobi [1]. Our ECE532 project objective is to create a working linear programming solver implementation on an FPGA that leverages parallelization to obtain performance speedups relative to existing general-purpose CPU implementations.

There exists two algorithms for solving linear programs: the simplex method (sometimes called active-set method) and the interior-point method. To our knowledge, there has only been one implementation of the simplex method on an FPGA [2], and a few implementations of the interior-point method, particularly for model-predictive control (MPC) where the rapid solving of small to medium-size models is critical [3]. Both algorithmic approaches present challenges and trade-offs as discussed in [4]. For this project, we will focus on implementing a version of the Simplex Method, which is relatively less complex to implement, but is more prone to degeneracy and is limited in the forms of problems it can solve.

Importantly, to our knowledge, no FPGA implementations of linear program solvers exist that can handle very large problems such as those typically found in energy systems modelling. Indeed, existing implementations can handle only up to 100-1000 variables and constraints [2], [3]. We aim to develop an FPGA solver that can handle larger datasets.

### *B. Goals*

The functional requirements for the project are [5] :

1) Implementation of a linear programming solver algorithm: Simplex Algorithm.
2) Use of ethernet to transfer data between the board and the host machine.
3) Use of DDR memory interface for data: Our anticipated memory requirements are 500 MB given typical energy systems datasets (e.g. [6]). This does not fit on-chip but will rely on the DDR memory interface.

4) Visualization of the linear programming solver in progress and final results via a GUI on the host machine. This could include rendering a graph to show the progress of solvers converging to a solution. However, this is a ‘nice-to-have’ and is not part of the project complexity point computation.

Our acceptance criteria are:

1) Have we implemented at least one solver? Answer must be ‘yes’
2) Can we transfer linear programs to the board and their solutions off the board using ethernet? Answer must be ‘yes’
3) Can we use both off-chip and on-chip memory? Answer must be ‘yes’

## C. Block Diagram

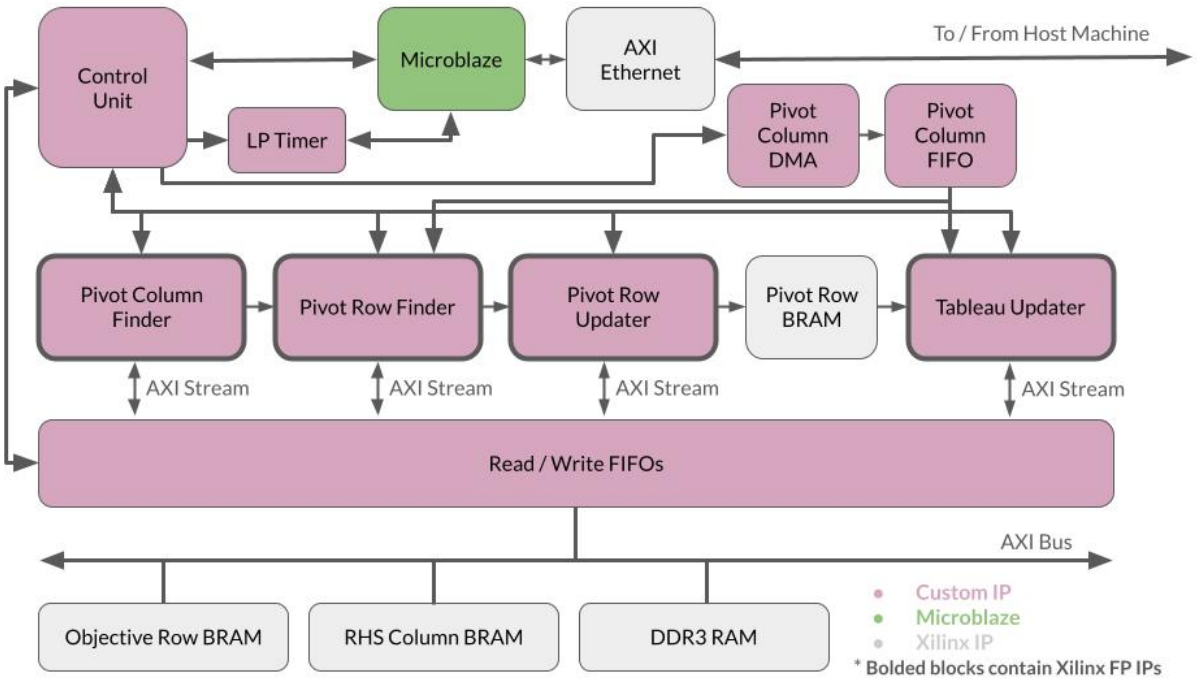

Fig. 1: Linear Programming Solver System Block Diagram

## *D. IP Overview*

| **Component** | **Description** |
|---|---|
| Control Unit | Manages the execution and data flow of the entire LP solver subsystem. |
| MicroBlaze | Xilinx's soft processor core for interfacing the Ethernet and LP subsystems. |
| LP Timer | Custom RTL timer module to time the LP algorithm's execution. |
| AXI Ethernet | Interfaces with Ethernet communications, enabling data exchange between the solver system and the host machine. |
| Pivot Column Finder | Custom RTL module for identifying the pivot column. |
| Pivot Row Finder | Custom RTL module for identifying the pivot row. |
| Pivot Column DMA | Facilitates direct transfer of the pivot column data to and from DDR. |
| Pivot Column FIFO | Allows for streaming pivot column data to start processing data without all data needing to arrive. |
| Pivot Row Updater | Custom RTL module for updating values in the pivot row. |
| Pivot Row BRAM | Xilinx's Block Memory Generator for caching the pivot row. |
| Tableau Updater | Custom RTL module for updating the tableau during each iteration of the algorithm. |
| Objective Row BRAM | Caches the objective row's data (contains coefficients of the objective function in an LP problem). |
| RHS Column BRAM | Caches the values of the RHS column (the constraints' values in the LP problem). |
| DDR3 RAM | Standard double data rate type three random access memory used for storing larger sets of data that do not fit in BRAM. |

TABLE I: Descriptions of LP Solver System Components

### E. Project Complexity

Our project's complexity was computed as follows:

| **Component** | **Complexity Score** |
|---|---|
| Custom LP Algorithm Solver Cores | 3.00 |
| Application-Level Ethernet Protocol for Sending/Receiving Data | 0.25 |
| LP Control Unit | 0.50 |
| Buffering (FIFO) IP Cores | 0.75 |
| *Total* | 4.50 |

TABLE II: Project Complexity Scores

## 2. Outcome

### A. Results

Concerning the requirements and criteria outlined in section 1-B, we have achieved our project goals. The following tableaus have been solved successfully, giving a feasible solution (with a small error compared to Gurobi, which we have used to validate):

| **Tableau Size** | **Wall Clock Time** |
|---|---|
| $4 \times 4$ | $10\mu s$ |
| $4 \times 7$ | $11\mu s$ |
| $5 \times 5$ | $11\mu s$ |
| $5 \times 7$ | $19\mu s$ |
| $10 \times 10$ | $29\mu s$ |
| $65 \times 128$ | $29.198ms$ |
| $1000 \times 1000$[1] | $113.256s$ |
| $1000 \times 1000$[1] | $108.615s$ |
| $1000 \times 1000$[1] | $1.291s$ |

TABLE III: Tableau Solution Times (system is clocked at 100 MHz)

[1]Both tableaus are of the same size, but describe different problems.

We compared the wall clock time of our own C implementation for the 4x4 and 65x128 tableau sizes, taking 0.029 seconds and 0.045 seconds. This offers a rough idea that, as expected, there is speedup when moving to hardware. However, as we increase our tableau sizes and sparsity, we start running into some of the issues that plague the Simplex Algorithm:

1) **Sensitivity to Input and Problem Formulation**: The Simplex algorithm's performance can be significantly affected by the choice of initial vertex or basic feasible solution and how the problem is formulated. Although we apply a presolve step to help with this, this adds complexity and requires more computational effort.
2) **Handling Degeneracy**: Degeneracy is another challenge that can occur when multiple basic variables are associated with a zero value in the solution. This can cause the Simplex algorithm to cycle, repeatedly visiting the same vertices without progressing towards the solution. Although there are anti-cycling rules like Bland's rule to prevent this, they slow down the algorithm and do not always guarantee finding a feasible solution.
3) **Numerical Stabillity**: The Simplex algorithm involves a series of arithmetic operations that can introduce rounding errors when dealing with floating-point arithmetic. These errors can accumulate and potentially lead to incorrect solutions or convergence to a non-optimal vertex, especially in large-scale problems.

## *B. Future Work*

The team developed a module called "choose_pivot_row" to use cross-multiplication for division steps. However, integrating this module within the given timeline proved too complex. So, we used a divider instead, which increased the module's latency. In the future, more time can be spent ensuring that cross-multiplication works, as it is a latency-saving optimization that can be applied to the solver.

Furthermore, while we created a module structure that seems to break down the algorithm into its more modular parts, it is possible to develop different splits of the algorithm, resulting in different RTL modules. This would require more time to experiment with different solver configurations and architectures, which was not feasible within the project timeline.

This could even involve implementing entirely different algorithms (e.g. Revised Simplex), which have greater potential for performance gains and optimizations, albeit with implementation difficulties. Another option is to try out a third-party Floating-Point IP module to compare it against Xilinx/AMD's version. Additionally, given that our current algorithm runs sequentially, we could experiment with parallelized setups for different stages of the computation to see what performance benefits, at the cost of implementation complexity, can be gained from this.

## 3. Project Schedule

Our project's timeline and milestones, outlined in the initial proposal [5], set a structured path for our team's progress. Here, we compare the projected milestones with the actual progress achieved during the project's execution:

| **Milestone** | **Objective** | **Deliverables** |
|---|---|---|
| 1 | Become familiar with linear programming | Each team member builds a toy linear programming solver (e.g., in Python) that works at solving a real model. Half the members build an interior point solver, the other half a simplex solver. |
| 2 | Pick a specific approach and architecture | Team members pick a specific architecture/approach (e.g., we will do the simplex algorithm) and implement a C version of the algorithm. Simultaneously some team members start developing the IP for the ethernet systems and memory access systems. |
| 3 | Core development | Make progress on the IP cores for ethernet, algorithm, and memory access, with a target of 80% of the functionality of such cores being in place. |
| 4 | Core development | Get all cores working individually. |
| 5 | Integration of systems | Get all the cores working together. |
| 6 | GUI development | Develop a GUI to visualize progress of the solver. |
| 7 | Performance benchmarks + buffer | Develop presentation, fixes, and performance benchmarks |

TABLE IV: Projected Project Milestones

## A. *Projected vs. Actual Milestones*

The original milestones were designed to provide a clear framework for developing a linear programming solver. These milestones (Table IV) ranged from initial familiarization with linear programming to developing a GUI for visualization purposes.

As the project unfolded, our team faced both expected and unforeseen challenges that shifted our timeline and altered our focus at various stages. Our actual achievements, detailed in Table V and Table VI, reflect these adjustments.

| **Milestone** | **Objective** | **Deliverables** |
|---|---|---|
| 1 | Familiarize with linear programming concepts | Team members developed initial solvers in high-level software to understand linear programming concepts. Adham and Ahmed explored primal-dual interior point methods, while Abnash focused on the simplex method. Martin conducted a literature review on hardware implementations of solvers. |
| 2 | Implement the Simplex Method in C and evaluate the Revised Simplex Method. | Implemented the ”standard” Simplex Method in C, facilitating easier port to HDL (Ahmed, Abnash). Assessed and decided against using the Revised Simplex Method due to its complexity and memory efficiency concerns (Martin). |
| 3 | Start RTL development focusing on memory subsystem and control | Developed Ethernet Memory Subsystem (Ahmed) and control logic (Adham). Martin converted electricity grid data into LP byte stream format, while Abnash designed the Simplex Pivot Column Selection Module in Verilog. |
| 4 | Continue development of the LP subsystem in RTL | Adham focused on subsystem and control design. Ahmed improved Ethernet subsystem efficiency and initiated LP Core Initialization. Martin integrated energy model directly with the custom solver. Abnash continued developing the Pivot Row Selection LP core module in Verilog. |

TABLE V: Project Milestones 1-4

| Milestone | Objective | Deliverables |
|---|---|---|
| 5 | Continue RTL development and begin testing and integration | Adham focused on DMA and control design and testing. Ahmed focused on memory writeback logic. Martin and Abnash continued development on the LP core modules for pivot row selection and tableau updates, respectively. |
| 6 | Integrate LP core modules and conduct extensive testing | Conducted DMA testing and reconfiguration (Ahmed, Adham). Martin and Abnash continued developing LP core modules, focusing on resolving issues and streamlining functionality. |
| 7 | Complete integration and most testing in preparation for the demo | Conducted post-implementation simulations and utilization studies (Adham), and integrated and verified functionality of LP modules on FPGA (Ahmed). Completed the development of pivotal LP cores, including Update Tableau and Choose Pivot Row modules (Martin, Abnash). |

TABLE VI: Project Milestones 5-7

## *B. Evaluation of Milestone Progress*

*1) Familiarization with Linear Programming:* Originally, the team aimed to have each member build a toy solver to understand linear programming fundamentals. In practice, we found that diving into specific methodologies, such as primal-dual interior point methods and the simplex method, offered more targeted insights. This focus allowed for a deeper understanding, which proved essential in the subsequent development of our solver.

*2) Choosing the Approach and Architecture:* We planned to select a specific algorithm and implement it in C. The decision to forgo the Revised Simplex Method was based on practical assessments of complexity and memory efficiency.

*3) Core Development:* The development of IP cores for ethernet, algorithm, and memory access saw substantial progress, although reaching 80% functionality within 4-5 weeks of the project proved to be an ambitious target. The dynamic nature of RTL development required continuous iteration, which slightly shifted our deliverables from what was initially planned.

*4) Integration and GUI Development:* The integration of the various subsystems posed complex challenges that demanded additional time and resources. The GUI development was consequently nixed to accommodate this complexity.

*5) Performance Benchmarks and Final Stages:* The final stretch of the project was met with intensive testing and integration, aligning closely with our projected milestones. The team successfully managed to integrate the LP modules on the FPGA, despite minor delays due to debugging and optimization of the solver to try to solve larger tableaus.

### C. Conclusion

In retrospect, while our projected milestones provided a foundational blueprint, the actual progress was shaped by the technical realities encountered along the way. The deviations from the initial plan were not merely obstacles but opportunities that allowed us to refine our approach and deliver a more robust linear programming solver, despite some of our initial hopes for optimization being shelved. The flexibility and responsiveness displayed by the team were pivotal in navigating these challenges, resulting in successful project completion.

## 4. System Design Overview

### A. Utilization and Timing

An essential aspect of our project's physical implementation involved analyzing the resource utilization and timing performance on the Nexys Video Board, with our system being clocked at 100 MHz. The following tables summarize the utilization of various FPGA resources, such as Slice Look-Up Tables (LUTs), Slice Registers, Multiplexers (Muxes), Block RAM (BRAM) Tiles, and Digital Signal Processing (DSP) slices, alongside timing metrics which include Worst Negative Slack (WNS), Worst Hold Slack (WHS), and Worst Pulse Width Slack (WPWS), and the maximum frequency ($F_{Max}$) at which we can clock the circuit without violating setup time for flip-flops on our FPGA.

*1) Resource Utilization:* Resource utilization is a critical metric in FPGA design, influencing power consumption, speed, and the ability to fit the design within the available hardware. The table below captures the total and component-wise utilization:

[2]FIFO Generators + BRAM Blocks not on here as they do not appear in the synthesis reports individually; however, we can still get the total usage.

| Component | Slice LUTs | Slice Registers | BRAM Tiles | DSP |
|---|---|---|---|---|
| fifo_write_intr_0 | 127 | 111 | 0 | 0 |
| mblaze_lp_bridge | 106 | 269 | 0 | 0 |
| fifo_write_intr_2 | 176 | 127 | 0 | 0 |
| fifo_write_intr_1 | 127 | 111 | 0 | 0 |
| find_pivot_col_0 | 31 | 66 | 0 | 0 |
| fifo_read_intr_0 | 115 | 102 | 0 | 0 |
| choose_pivot_row_0 | 1001 | 1863 | 0 | 0 |
| fifo_dma_0 | 72 | 88 | 0 | 1 |
| update_pivot_row_0 | 957 | 1711 | 0 | 0 |
| fifo_dma_1 | 72 | 88 | 0 | 1 |
| fifo_redirect_1 | 84 | 57 | 0 | 0 |
| update_tableau_0 | 941 | 1465 | 0 | 4 |

TABLE VII: Resource Utilization Summary: Part 1

The resource utilization reported reflects a design that efficiently utilizes FPGA resources, with the following percentage utilization relative to the total available resources:

- % LUT Utilization: 17.572% ($\frac{23652}{134600}$)
- % FF Utilization: 10.155% ($\frac{27337}{269200}$)
- % BRAM Utilization: 38.082% ($\frac{139}{365}$)
- % DSP Utilization: 1.892% ($\frac{14}{740}$)

While we were initially concerned about the potential resource utilization, these results show that our fear may have been slightly exaggerated. There is room for potential expansion across our data precision, usage of BRAMs for caching, and further expansion to our solver logic.

*2) Timing Performance:* The timing performance metrics ensure the design meets the required speed for real-time operations. Given our 10 ns clock period, the post-implementation timing results are satisfactory with the following metrics:

- Timing Worst Negative Slack (WNS): 0.375 ns
- Timing Worst Hold Slack (WHS): 0.052 ns
- Worst Pulse Width Slack (WPWS): 0 ns

| **Component** | **Slice LUTs** | **Slice Registers** | **BRAMs** | **DSPs** |
|---|---|---|---|---|
| axi_dma_0 | 2467 | 3801 | 2 | 0 |
| axi_timer_0 | 299 | 244 | 0 | 0 |
| xbar_0 | 329 | 138 | 0 | 0 |
| microblaze_0_axi_intc_0 | 236 | 221 | 0 | 0 |
| microblaze_0 | 2398 | 2576 | 20 | 2 |
| dlmb_bram_if_cntlr_0 | 6 | 2 | 0 | 0 |
| clk_wiz_1 | 1 | 0 | 0 | 0 |
| ilmb_bram_if_cntlr_0 | 4 | 2 | 0 | 0 |
| xbar_3 | 2849 | 1022 | 0 | 0 |
| lp_timer_0 | 44 | 66 | 0 | 0 |
| lp_control_unit_0 | 110 | 127 | 0 | 6 |
| fifo_read_intr_0_3 | 115 | 102 | 0 | 0 |
| fifo_read_intr_0_1 | 161 | 118 | 0 | 0 |
| fifo_read_intr_0_2 | 115 | 102 | 0 | 0 |
| bd_929b_eth_buf_0 | 1438 | 2776 | 4 | 0 |
| bd_929b_c_counter_binary_0 | 7 | 25 | 0 | 0 |
| bd_929b_c_shift_ram_0 | 1 | 1 | 0 | 0 |
| bd_929b_mac_0 | 1813 | 3014 | 0 | 0 |
| *Total Usage*[2] | 23652 | 27337 | 139 | 14 |

TABLE VIII: Resource Utilization Summary: Part 2

We calculate our $F_{Max}$ as follows [7]:

$$F_{Max} = \frac{1}{\text{Clock Period} - WNS} \quad (1)$$

- **Clock Period** is the time interval between successive clock edges. Our design's clock period is set at 10 ns to match the 100 MHz operational target.
- **WNS** is the Worst Negative Slack, indicating the amount of time by which the slowest path exceeds the required clock period. A positive WNS means the path completes before the next clock edge, while a negative WNS indicates a timing violation. In our case, the

WNS is 0.375 ns, which shows that the slowest path in the design finishes within the clock period, thereby meeting the timing constraints.

Using the above equation and our timing information, we can calculate the FMax for our project. With a WNS of 0.375 ns and a clock period of 10 ns, we have:

$$F_{Max} = \frac{1}{10ns - 0375ns} \approx 103.90\text{MHz} \tag{2}$$

Thus, our design can potentially run faster than the baseline clock frequency of 100 MHz, with some leeway for variations in operation conditions and future scaling.

*3) Conclusion:* Our project's FPGA resource utilization and timing performance are within acceptable ranges. The careful consideration of design choices, alongside optimizations performed during the development cycle, has ensured that our implementation fits well within the FPGA and operates reliably within the defined clock period constraints.

## *B. Problem Setup*

In order to test our solver with real data, we developed a significant software stack which included our own pre-solver. All the code is available under the "pyomo_fpga_bridge" directory in our project.

The software stack is represent in Figure 2 and is as follows.

First, a real energy model from [6] was used. The model was downsized to represent the entire Western North American electricity grid but only for a single hour. Since the model is built on the Pyomo Python optimization library, we developed a solver plugin for Pyomo which, when called from within Pyomo, starts our solving process.

The plugin first writes the optimization model to a text file (in the .mps format). It then calls our custom presolver which reads in the text files and performs a sequence of operations to reduce the size of the optimization model as described in a following section. Then the presolver passes the reduced model to our "standard form converter" which applies further transformations to ensure the problem reaches the standard form required for the Simplex algorithm. Finally, a sparse numpy matrix is generated from this standard form and sent over ethernet to the FPGA.

Note that at every step, the model is written a text file and solved with the Gurobi commercial solver to ensure the model hasn't been improperly altered.

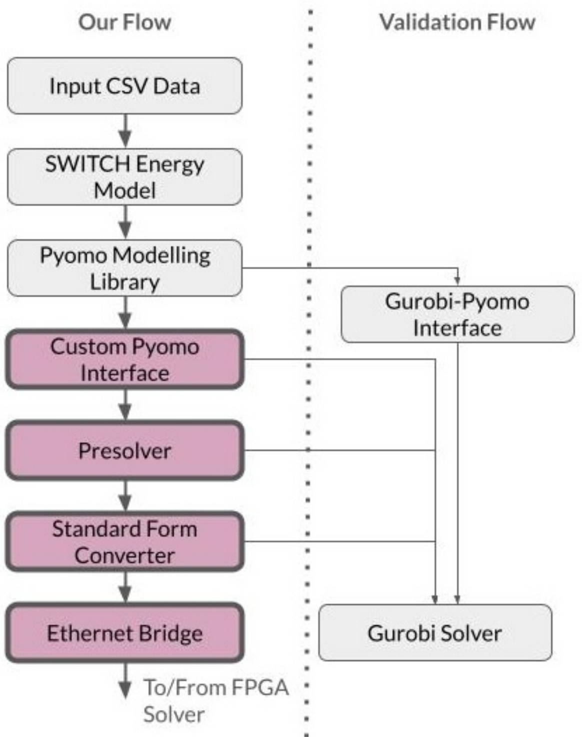

Fig. 2: Presolve Flow to simplify problem for LP Solver

*1) Custom Presolver:* Our presolver consists of 10 custom operations which are performed repeatedly until no more reductions to the model can be found. For context, a linear programming model has 4 components.

1) A set of decision variables $x \in \Re^n$. (each element in the vector $x$ is a decision variable)
2) Bounds on these variables (e.g. $l_i \leq x_i \leq u_i$).
3) An objective $c^T x$ that must be either minimized or maximized ($c \in \Re^n$).
4) A set of constraints (e.g. $a_{11}x_1 + a_{12}x_2 \leq b_1$) which can be generalized to the form $Ax \leq b \quad A \in \Re^{m \times n}$. Note that although we use a $\leq$ sign, individual constraints can also be $\geq$ constraints or equalities ($=$).

As such, each of the following 10 transformations convers one or multiple of these 4 components of the linear programming model.

*a) Presolve Transform 1: Constraints to bound:* Any inequality constraint with a single variable is converted to a bound on that variable rather than a constraint. This is shown mathematically for the $\leq$ inequality case below.

$$a_{ij}x_j \leq b_i \rightarrow x_j \leq \frac{b_i}{a_{ij}}$$

*b) Presolve transform 2: Remove variables fixed by equality constraint:* Any equality constraint with a single variable effectively sets the value of that variable. As such, that variable can be replaced throughout the model (constraints and objectives) with its value.

$$a_{ij}x_j = b_i \rightarrow x_j = \frac{b_i}{a_{ij}}$$

*c) Presolve transform 3: Remove variables fixed by bounds:* If the lower and upper bound of a variable are equal, the variable must be equal to its bounds and can hence be replaced as in transform 2.

$$l_i <= x_i <= u_i \rightarrow x_i = l_i = u_i \quad \forall l_i = u_i$$

*d) Presolve transform 4: Change inequality to equality using bounds:* If all the variables in a $\leq$ constraint have lower bounds that ensure the left hand side will always be at least as large as the right hand side, the constraint can be considered an $=$ constraint. For example,

$$2x_1 + 3x_2 \leq 7 \tag{3}$$
$$2 \leq x_1 \tag{4}$$
$$1 \leq x_2 \tag{5}$$
$$\downarrow \tag{6}$$
$$2x_1 + 3x_2 = 7 \tag{7}$$

The equivalent rule is applied on $\geq$ constraints.

*e) Presolve transform 5: Change inequality to equality if the objective pushes in that direction:* If a $\leq$ constraint has a variable that only appears in the objective, and that, given its coefficient in the objective, will attempt to be maximized, the constraint is converted to a $=$ constraint since the variable will always increase until the constraint binds it. The one caveat is, if this variable has an upper bound, the bound must be found to be redundant (using a similar process to what is done in presolve transform 4).

$$\text{Maximize obj:} \quad ... + c_i x_i + ... \quad (c \geq 0) \tag{8}$$

$$a_1 x_1 + a_2 x_2 + ... + a_i x_i + ... \leq b_i \quad (a_i > 0, u_i = \infty) \tag{9}$$

$$\downarrow \tag{10}$$

$$a_1 x_1 + a_2 x_2 + ... + a_i x_i + ... = b_i \tag{11}$$

*f) Presolve transform 6: Substitute out equalities with $\leq$ 10 terms:* Equality constraint can be rewritten in order to define one variable in terms of the others. As such, this first variable can replaced throughout the model with the others to make the model smaller. This optimization is only performed for equality constraints with less than 10 variables to avoid making the model excessively dense. For example, in the following case $x_3$ can be substituted out.

$$3x_1 + 2x_2 + 4x_3 = 5 \tag{12}$$

$$\downarrow \tag{13}$$

$$x_3 = 5 - 2x_2 - 3x_1 \tag{14}$$

*g) Presolve transform 7: Remove unused bounds:* Variables that aren't used in the objective or constraints are effectively useless. In these cases, the variable and its bounds can be removed.

*h) Presolve transform 8: Remove empty constraints:* Occasionally, during other presolve steps (e.g. variable substitution), a constraint will have all its variables cancel out. This constraint can be removed (after ensuring its inequality is still respected, e.g. $0 \leq 5$ works but not $0 \leq -2$).

*i) Presolve transform 9: Fix unconstrained variables to its bounds:* A variable that does not appear in any constraints (only the objective) will be either maximized or minimized until it reaches its bound. As such, we preemptively fix it to its bound.

*j) Presolve transform 10: Remove weaker constraints:* Some constraints apply to the same variables but with different coefficients. In these cases, if the coefficients are always positive and it is a $\leq$ constraint, we can find which of these constraints is "tightest". All other constraints are redundant and can be removed.

*2) Standard form transformations:* To convert our problem into standard form, the following transformations were implemented.

1) Flip upper bounds to lower bounds since the standard form doesn't allow upper bounds.

For example, $x_i \leq 5$ becomes $x'_i \geq -5$ where $x'_i = -x_i$. Of course, $x_i$ must be replaced with $x'_i$ in all the equations.

2) The previous transform doesn't work if $x_i$ also has a lower bound (since the lower bound would become an upper bound which defeats the purpose of the flip). In these cases, the upper bound is simply converted to a constraint (e.g. $x_i \leq u_i \rightarrow x_i \leq b_i$ where $b_i = u_i$).
3) Unbounded variables aren't allowed in the standard form and are replace with two bounded variables tied together with a constraint. Mathematically: $-\infty \leq x_i \leq \infty \rightarrow x_i = x'_i - x'_i, \quad x'_i \geq 0, \quad x''_i \geq 0$. Of course, $x_i$ must be replaced throughout the model.
4) All lower bounds must be 0 in standard form. As such lower bounds of the form $4 \leq x_i$ are replaced with $0 \leq x'_i$ where $x'_i = x_i - 4$. Substitutions are performed
5) Minimization problems are changed into maximization problems by multiplying the objective by $-1$.
6) Slack and artificial variables are added into the model to convert all inequalities into equalities. For example, $x_i + x_j \leq 5$ becomes $x_i + x_j + s_1 = 5, s_1 \geq 0$. Artificial variables are introduced with the Big-M technique where $\geq$ inequalities are given a slack variable that is heavily peanalized for non-zero values.

### *C. Data Transfer*

There are three components to data transfer in this project: transferring the tableau to/from the FPGA, connecting the Microblaze with the LP control unit, and reading/writing data from memory.

*1) Ethernet:* After the LP problem is formulated in standard form, it is sent from the host machine to the FPGA through Ethernet. The Microblaze handles receiving the data and storing the tableau elements in their respective memory modules (i.e. DDR, RHS column BRAM, or OBJ row BRAM). We noticed that sending data from the host machine to the FPGA (∼385 KB/s) is much faster than sending data from the FPGA back to the host machine (∼38 KB/s). Hence, after the LP algorithm is done executing, we format the solution on the Microblaze and send that information back to the host machine. This ends up being more efficient, as we only need to send back a single row in the tableau as opposed to the entire tableau. **Code file: main.c**

*2) Microblaze LP Bridge & LP Timer:* After all data is received and stored in memory, the Microblaze communicates with a custom IP that we call the 'bridge'. This IP acts as a memorymapped AXI slave to the Microblaze, and serves as the interface between the Microblaze and the LP control unit. It provides essential data to the various LP blocks and control unit such as the dimensions of the tableau, as well as the start signal to initiate the LP algorithm. This same start signal is communicated to a custom timer IP module that calculates the exact runtime of the algorithm to the nearest microsecond. The timer stops as soon as the control unit receives a 'terminate' signal from any of the LP solver blocks. **Code files: mblaze_lp_bridge_v1_0_S00_AXI.v, mblaze_lp_bridge_v1_0.v, lp_timer.v**

*3) Read/Write FIFO Interfaces:* A key challenge with this project was making sure that memory access is not the bottleneck of the algorithm. If we want to accelerate Simplex, we needed to make sure that all the LP solver blocks have data at the ready and are not wasting extra cycles writing data back to memory. We designed custom FIFO read and write interfaces to tackle this issue.

- **FIFO Read Interface**: This custom IP block uses an AXI-full interface to read N-items from a specific location in memory and sequentially writes these items using an AXI-stream interface to a Xilinx FIFO Generator IP [8]. We utilize an AXI-full interface to take advantage of the burst feature which allows us to read 256 elements at a time from memory. This block is also able to take in a custom stride pattern, in the case where we would like to read a specific column from memory. While we could have went with existing memory-accessing modules, such as Xilinx's DMA Controller [9], we found that we were able to implement the same functionality as a DMA in a much simpler custom format while achieving the same read latency. **Code files: fifo_read_interface.v, fifo_dma.v**
- **FIFO Write Interface**: This custom IP block takes in an AXI-stream interface from Xilinx's FIFO Generator IP [8] and uses an AXI-full interface to write the results to a specified memory location sequentially. Once again, we utilize an AXI-full interface to be able to write back data in bursts. Since all of our data writes are sequential in nature, we didn't need to implement striding as we did with the Read Interfaces. **Code file: fifo_write_interface.v**
- **FIFO Redirect Controller**: Since the final LP solver block updates the entire tableau and needs to write it back to memory, we run into an issue of synchronization, where not only do we need to update the DDR memory with the new tableau, but we also need to update our cache (namely, the objective row and RHS column BRAMs) with these new elements as well. In order to handle three simultaneous memory write-back operations to

different memory modules while maximizing the performance of the algorithm, we designed a custom 'redirect' controller that takes in an AXI-stream from the last LP module, and 'steers' this data through three separate AXI-streams to their appropriate memory modules. We utilize this module in conjunction with FIFOs and our custom FIFO Write Interfaces to accomplish this task with minimal latency.

For example, if this module receives an updated element whose index is in the objective row, then it should be updated both in DDR and the objective row BRAM. Hence, the block will assert the valid signals on both of these busses and wait until they are buffered into their respective FIFOs.

Using this method ensures that not only is all of our data synchronized, but also this synchronization happens at no extra delay, since all of our memory modules are updated by the time the last element in our tableau is written back to DDR. **Code file: fifo_redirect.v**

### *D. Control, Caching, and Buffering*

The LP solver operates efficiently with the help of control, caching, and buffering mechanisms. These systems work together to manage the flow of data and instructions, ensuring that the solver operates smoothly and effectively.

*1) Control:* The LP solver has a control unit implemented within a Verilog module called 'lp_control_unit', which serves several crucial functions:

- Steers data and logic through the LP Subsystem, ensuring that operations are executed in the correct sequence and that data flows smoothly between different components of the system.
- Responsible for driving progress from one solving core to the next. It manages the transitions and interactions among various states, such as pivoting columns and rows, updating pivot rows, and writeback operations.
- Oversees data transfers between block RAM (BRAM) blocks and external DDR memory. This allows for maintaining high data throughput and minimizing latency.

The control unit's functionality is encapsulated within an FSM inside 'lp_control_unit', which consists of the following states:

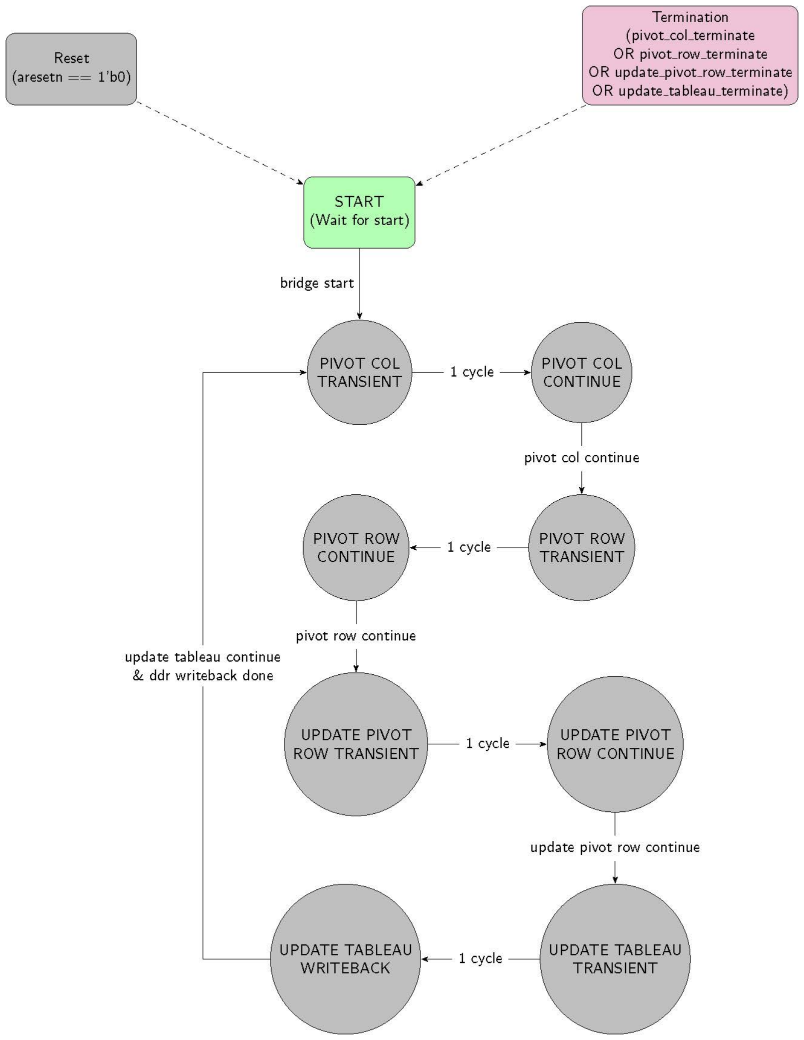

Fig. 3: LP Control Unit FSM

- **START**: This is the initial state where the FSM awaits a start signal (‘bridge_start‘). Upon receiving this signal, the FSM transitions to the ‘PIVOT_COL_TRANSIENT‘ state to begin the pivot column operation.
- **PIVOT_COL_TRANSIENT**: Acts as a setup stage for the pivot column operation. It deasserts previous control signals and prepares the system for the next stage by transitioning to the ‘PIVOT_COL_CONTINUE‘ state after one clock cycle.
- **PIVOT_COL_CONTINUE**: This state waits for the ‘pivot_col_continue‘ signal. Once received, it resolves addresses for the pivot column BRAM block and initiates the pivot row operation, transitioning to the ‘PIVOT_ROW_TRANSIENT‘ state.
- **PIVOT_ROW_TRANSIENT**: Similar to ‘PIVOT_COL_TRANSIENT‘, this state prepares

for the pivot row operation to complete and proceeds to ‘PIVOT_ROW_CONTINUE‘ after one clock cycle.

- **PIVOT_ROW_CONTINUE**: Here, the FSM checks the ‘pivot_row_continue‘ signal to transition to the ‘UPDATE_PIVOT_ROW_TRANSIENT‘ state. It also resolves the address for the pivot row BRAM block and calculates the index for the upcoming tableau update.
- **UPDATE_PIVOT_ROW_TRANSIENT**: This state prepares for updating the pivot row in the tableau to complete and moves to ‘UPDATE_PIVOT_ROW_CONTINUE‘ after one clock cycle.
- **UPDATE_PIVOT_ROW_CONTINUE**: It waits for the completion of the pivot row update (‘update_pivot_row_continue‘) before proceeding to initiate the tableau update operation.
- **UPDATE_TABLEAU_TRANSIENT**: Prepares for the tableau update to complete and proceeds to ‘UPDATE_TABLEAU_WRITEBACK‘ after one clock cycle.
- **UPDATE_TABLEAU_WRITEBACK**: The final state in the cycle where the FSM manages the writeback to DDR and checks for the completion of the update tableau operation. Once the ‘update_tableau_continue‘ and ‘ddr_writeback_done‘ signals are received, it signifies the completion of one cycle of operation, and the FSM transitions back to ‘PIVOT_COL_TRANSIENT’. to begin a new operation cycle or returns to ‘START‘ if a termination condition is met.

Each FSM state is designed to perform specific control actions required for the LP solver’s operation, such as asserting or deasserting control signals, managing transitions between operations, and resolving memory addresses. The FSM ensures that each step is executed in the correct sequence and that the solver maintains a steady progression through its algorithmic stages. The states transition based on control signals that represent the completion of tasks or the readiness to begin new ones, providing a robust framework for coordinating the solver’s complex operations.

The ‘lp_control_unit’ module was synthesized to use DSP blocks (using the ‘use_dsp’ pragma) for the address resolution calculations. The module was also tested in functional and timing simulations outside of the system to verify that the correct control signals are asserted within each state and with little to no glitches/meta-stability, as well as part of the larger LP subsystem on-board.

*2) Caching:* Caching is essential for the performance of the LP solver since it directly impacts the speed at which the solver can access and process data.:

- BRAM blocks, using Xilinx's block memory generator [10], are employed as caches for the LP solver. They store the rows and columns (namely, the right-hand-side column, the pivot column, and the pivot row) that are frequently accessed during the solving process, thus reducing the need to repeatedly fetch data from the slower DDR memory.
- This caching mechanism significantly decreases the latency associated with memory access, allowing the solver cores to operate more efficiently and with fewer interruptions.

Writing to BRAM blocks was tested using on-board testing, with integrated logic analyzers (ILAs) attached to ensure that the data across the AXI4 interface was correct.

*3) Buffering:* Buffering is implemented to handle the asynchronous nature of data processing in the LP solver:

- First-In-First-Out (FIFO) buffers are used for streaming data to and from the solving cores. They provide a cushion that absorbs the variability in data processing times, thus maintaining a steady flow of data.
- These FIFO buffers also allow the solving cores to commence computation without having to wait for the entire data set to arrive. This is crucial for maintaining overall system throughput, especially when dealing with streaming data.

These FIFO buffers were generated using Xilinx's FIFO generator IP [8], and work (and were tested) in conjunction with the custom FIFO IP modules described in section 4-C3.

To summarize, the control unit's logic, caching through BRAM, and buffering using FIFO work together to guarantee that the LP solver executes its tasks with the necessary efficiency and speed.

## *E. LP Solver Modules*

The linear programming solver was broken into four key modules: Choose Pivot Column, Choose Pivot Row, Update Pivot Row, and Update Tableau. These correspond to the stages of the Simplex algorithm. We cycle through the four stages in multiple iterations until the algorithm reaches a final termination condition. The termination conditions include being unable to select any further pivot column or pivot row. This indicates that the algorithm has done as much optimization as it could to the input data tableau. The four modules were custom and integrated Vivado Floating Point IP within for select floating point operations.

All of our LP Solver modules are designed to support 32-bit precision using IEEE-754 for floating point representation. Another constraint in hardware is $2^{16}$ as the maximum dimension of the tableau for both the number of rows and the number of columns. This choice supported the maximum possible tableau that could be fit into the 512 MB of DDR memory on the Nexys Video Board. These values are parameterized and could be modified for future implementations that use, for example, full-precision or more DDR. Each module begins operation once it receives a start signal from the FSM and completes its operation by asserting either a terminate signal to end the solver or a continue signal to move to the next stage.

*1) Choose Pivot Column:* The first module of the LP algorithm is responsible for pivot column selection. This module was written entirely in Verilog and does not directly use Vivado IP. The module takes in the objective row of the tableau data over an AXI-S interface one element at a time. This objective row is cached and available to read into the module before it starts so there is limited delay in processing the data. Each element of the objective row is compared to select the most negative value; this approach matches the Simplex algorithm for maximization problems. To use this module for a minimization problem, we simply flip the signs of the objective row during the problem setup stage and no changes are required in the Choose Pivot Column module.

While we accept elements of the objective row, we use an internal counter to represent the column index of the element. After iterating through the entire objective row, if no negative element is found, we assert a terminate signal. This is provided to the FSM as a flag to end the entire Simplex algorithm. However, if we do successfully find an element that is more negative than the others in the objective row, its index is now our chosen pivot column. During the objective row checking, we also verify that the results are not special values in IEEE-754 such as NaN.

The module supplies the chosen pivot column index for use in later stages of the algorithm. For testing purposes, we also output the actual element. In the case of a successful pivot column selection, we raise a Continue flag to tell the FSM to proceed with the next module: Choose Pivot Row.

*2) Choose Pivot Row:* The Choose Pivot Row module takes in the pivot column data, which was retrieved using the pivot column index supplied by the previous module, and the right-most column of the tableau. This is accessed via two input AXI-S interfaces: one connected to a FIFO with the right-most column data and another with the pivot column. Two counters are used, one per column, to track the row index.

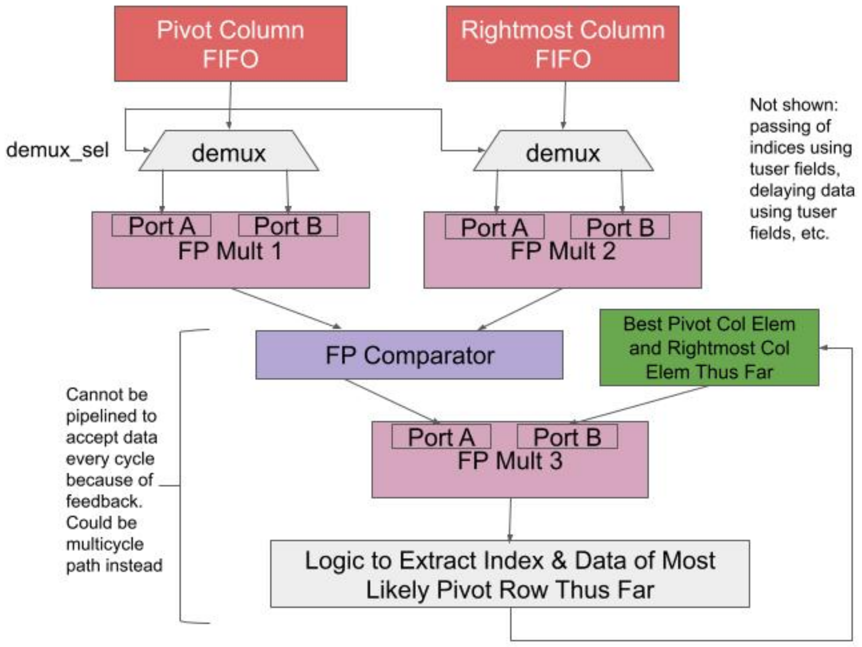
Pivot Column FIFO
Rightmost Column FIFO
demux_sel
demux
demux
Not shown: passing of indices using tuser fields, delaying data using tuser fields, etc.
Port A
Port B
FP Mult 1
Port A
Port B
FP Mult 2
FP Comparator
Best Pivot Col Elem and Rightmost Col Elem Thus Far
Cannot be pipelined to accept data every cycle because of feedback. Could be multicycle path instead
Port A
Port B
FP Mult 3
Logic to Extract Index & Data of Most Likely Pivot Row Thus Far

Fig. 4: Complicated two-stage cross multiplier design for Choose Pivot Row Module

Our initial design of this module was a two-stage approach using cross-multipliers as shown above in Figure 4. However, this proved too complicated and was abandoned after substantial development efforts. Substantial logic was needed to handle passing of the indices and raw data between modules because the cross-multiplication products cannot be reused for the next stage; they must be re-computed for every new comparison. Moreover, we could not pipeline the second stage of this design to accept data every cycle because of feedback: we need the outcome of the second stage before feeding new data into the second stage, or we may be comparing against an older best possible pivot row. Instead, we chose to implement the ratio test in hardware in order to choose the pivot row. Figure 5 is a summary of the Choose Pivot Row module.

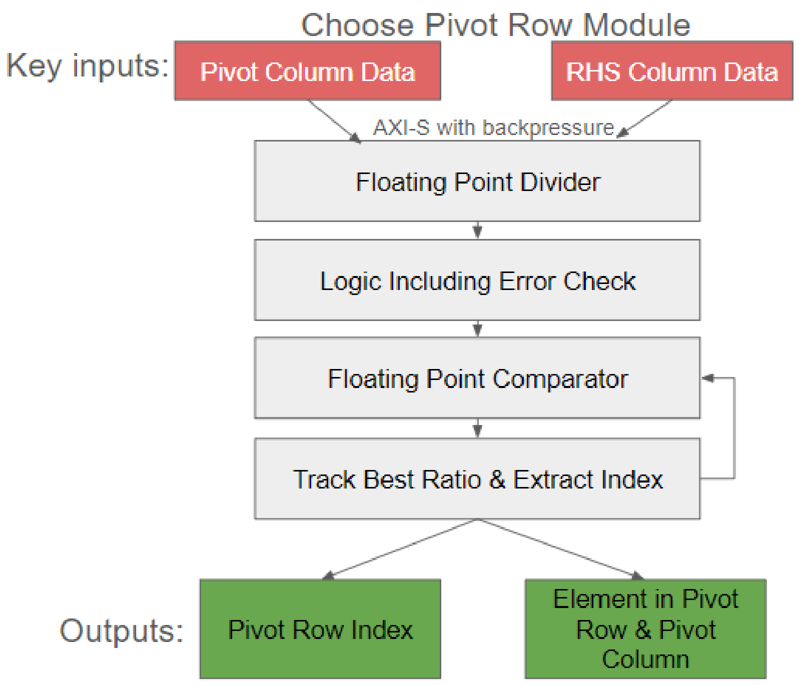

Fig. 5: High-Level Overview of Choose Pivot Row Module

To select the pivot row, we use the Vivado Floating Point IP to implement a floating point divider. The divider computes the ratio between the right-most element and the element in the pivot column, per row. The smallest non-negative ratio is selected as our pivot row. Both the pivot row index and the value located at the intersection of the pivot row and pivot column are provided as outputs of this module for use in later stages.

In terms of specific implementation details, we used the Vivado Floating Point IP's tuser field to keep track of the element indices. This helped prevent mismatch of data if one interface received data from its respective FIFO earlier than the other FIFO. The tuser field was also used to extract the pivot row index outputted by this module. We chose to pipeline to 29 cycles based on performance optimization suggestions from Vivado; this allowed for high throughput of 1 cycle per operation while meeting our 100 MHz timing goal. Moreover, we chose to use the divider in blocking mode, allowing us to exert back pressure upon the incoming FIFO data when the pipelined divider was full. This also helped with possible misaligned data arrival between the two column FIFOs.

When calculating the ratio test, we handled divide by zero, overflow, underflow, and other invalid operations (such as infinity divided by infinity) by checking the error bits supplied by the Vivado Floating Point IP divider. Rather than checking for invalid inputs at the input of the divider, we chose to check the result at the output because skipping one ratio would add more complexity to the design (e.g. no longer able to easily detect when we have checked all rows at the output of the module) without any throughput benefit given that the module is heavily pipelined. The error bits from the divider were bitmasked from the lower bits of the tuser field so some logic was needed to handle separating the error bits from the actual counter bits.

Registers, logic in Verilog, and the Vivado Floating Point comparator were used to select the smallest ratio encountered as we iterate through rows. The comparator was pipelined to 3 cycles. If a row does have an invalid ratio or the ratio is zero, it is not a viable pivot row and the row is skipped. However, if underflow occurs at the last row and no viable pivot row has been found, we made an algorithm design choice to accept that as the possible pivot row. This allows the algorithm to proceed with another stage of optimization.

*3) Update Pivot Row:* After we have identified the pivot row and pivot column indices for this iteration of the algorithm, the Update Pivot Row module is part of carrying out the pivot operation. The pivot row data is accessed as input over AXI-S from a FIFO. This module also takes as input the value located in the pivot row and pivot column, which we call val_in, passed from the previous module. Each of the elements of the pivot row are divided by val_in using the Vivado Floating Point IP generated floating point divider. The divider is pipelined to 31 cycles to prevent any throughput bottlenecks at this update module. Blocking mode was used on the divider to allow for the exertion of back pressure e.g. if the FIFO has more available data but the divider is currently full. The outputs of the module are native BRAM output signals streaming out the modified elements of the pivot row. To directly write into BRAM, we added logic within the module to calculate the correct address for the BRAM writes. The outcome of this module is that the pivot row reduces the value at the pivot column index to 1.

*4) Update Tableau:* The Update Tableau Module is responsible for updating the entire tableau based on the pivot row, pivot column, and updated pivot row from the previous modules. We iterate through each row of the tableau and use a fused multiply-add generated by the Vivado Floating Point IP generator to subtract multiples of the updated pivot row from the row. The outcome is a 0 in the pivot column of each row, except the pivot row which has a 1 based on the previous module.

Notably, although we iterate over the entire tableau fed in from our cached memory, we must be careful to avoid further modifying the pivot row. We use muxing logic to avoid reading the old value of the pivot row because the changes from our previous module are not yet reflected in DDR, which is the source of the cached FIFO data used in this module. We also pass a zero coefficient to the fused multiply-add to skip over the pivot row during our updates. Instead, we supply the updated pivot row from Update Pivot Row.

The output of the Update Tableau module is the entire tableau with modifications. This is supplied as an AXI-S interface to eventually be written back in DRAM, in addition to being cached where needed.

## 5. Design Tree

GitHub Repository: https://github.com/abnashkb/ece532/

Our files are organized into six main subfolders on the GitHub repository as follows:

- docs: PDF of this final report and presentation slides
- ignore: stash of files used during development but not required for understanding the current state of the project, such as deprecated modules
- lp_modules: Verilog files for our LP modules alongside the .xci files of the Vivado IP used for the LP modules and testbenches used for development.
- project_tcl_scripts: tcl scripts generated from our Vivado project, which can be used to regenerate the project
- pyomo_fpga_bridge: links to a separate repository where we host code used convert a real energy model into a streamable presolved standard form tableau. The repository also contains code to test this generated tableau against Gurobi.
- sdk_code contains C code run on the Microblaze, including for data transfer over Ethernet
- sw_benchmark has C and Python code used in our software implementations of Simplex; these were used to learn the Simplex algorithm and test our results from hardware
- verilog_module: any Verilog files not directly related to the LP modules

## 6. Tips and Tricks

- Familiarize yourself with the AXI4 full/lite/stream interfaces and only implement necessary AXI signals. This will save you much development time.
- Plan on integrating testbench-verified modules into your overall design as soon as possible. Things will go wrong and you will need more time than planned.
- Try to make your Verilog modules as parameterized or modular as possible, especially if they are used in interfacing between peripherals or memory.
- Draw out any FSM or design on paper first before you implement! I cannot stress how helpful this is in terms of catching logical errors and reducing coding time.
- Team dynamics is as important as team ability. Getting along with your teammates makes working on the project enjoyable, and making new friends is always welcome.
- Even if you think you don't need a testbench, you need a testbench. Following a development process that includes, at minimum, functional simulation of RTL in software before even moving to hardware will save you hours spent debugging.
- For solver-based projects, start with writing your own implementation in software first. This helps you understand the algorithms at hand and gives you something to compare your hardware results against for more effective testing.

## References

[1] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023. [Online]. Available: https://www.gurobi.com

[2] S. Bayliss, C.-s. Bouganis, G. A. Constantinides, and W. Luk, "An fpga implementation of the simplex algorithm," in *2006 IEEE International Conference on Field Programmable Technology*, 2006, pp. 49–56.

[3] J. Liu, H. Peyrl, A. Burg, and G. A. Constantinides, "Fpga implementation of an interior point method for high-speed model predictive control," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[4] M. S. K. Lau, S. P. Yue, K. V. Ling, and J. M. Maciejowski, "A comparison of interior point and active set methods for fpga implementation of model predictive control," in *Proceedings of the European Control Conference 2009*, 2009, pp. 156–161.

[5] A. Ragab, M. Staadecker, A. Hamoda, and A. Bassi, "Linear programming solver," University of Toronto, Tech. Rep., 2023, eCE532 Project Proposal.

[6] REAM_Research_Group, "Dataset used to model long-duration energy storage in the western u.s." https://github.com/REAM-lab/Staadecker_et_al_2022_archive, 2022.

[7] Xilinx, "Understanding timing summary report - Xilinx support," 2021. [Online]. Available: https://support.xilinx.com/s/article/57304?language=en_US

[8] Xilinx, "Fifo generator - ip core," https://www.xilinx.com/products/intellectual-property/fifo_generator.html, 2018.

[9] Xilinx, "Axi dma controller - ip core," https://www.xilinx.com/products/intellectual-property/axi_dma.html, 2018.

[10] Xilinx, *7 Series FPGAs Memory Resources User Guide (UG473)*, Xilinx, A Subsidiary of Advanced Micro Devices, Inc., 2018. [Online]. Available: https://docs.amd.com/v/u/en-US/ug473_7Series_Memory_Resources