CSEE W4840 Embedded Systems

AccelReg: An Accelerator for Linear Regression

Final Report

Doreen Sisanalli – ds4371
Pranav Asuri – pa2708
Varsha Keshava Prasad – vk2550
Venkat Suprabath Bitra – vsb2127

May 15, 2025

Introduction

Linear regression is a simple and widely used method for modeling the relationship between two variables: one input (x) and one output (y). In this project, we focus on the 1D case, where we fit a straight line to data points so that it best predicts y from x. The best line is found by minimizing the squared differences between the actual y values and the predicted values, a method known as least squares. We use the closed-form solution, which means we directly calculate the optimal line using matrix operations, instead of using iterative methods like gradient descent. This approach is efficient and gives an exact answer for the model parameters.

In this implementation, the dataset containing the paired input and output values is first stored on the SD card, which also holds the operating system for the FPGA. A key aspect of this project is the design of the linear regression training process on the FPGA using a 4-bit quantization-aware approach. This involves representing the numerical values with reduced precision, which is crucial for efficient hardware implementation. The software module reads the dataset from the SD card and sends it to the FPGA. The FPGA then processes this data to compute the intermediate values needed for calculating the optimal slope and intercept for the linear regression model through matrix operations, all while adhering to the 4-bit quantization constraints. The effectiveness of this quantization strategy is enhanced by utilizing a representative dataset during the quantization-aware training phase, which allows the model to achieve performance closely comparable to its full-precision counterpart.

As the results demonstrate, this approach yields an R2 score of 0.9490 and a Mean Squared Error (MSE) of 0.3402 for the 4-bit quantized model. This is visually represented

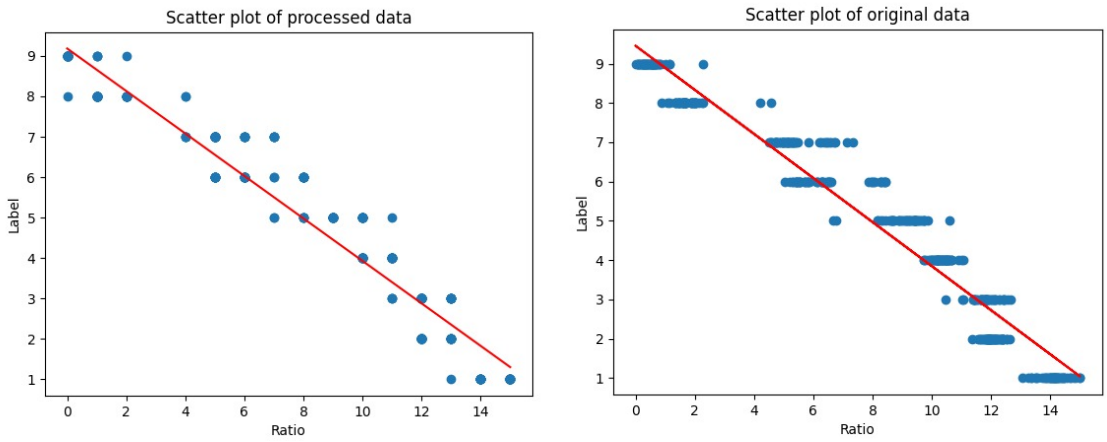


Figure 1: (a) Scatter plot of processed data with 4-bit quantized linear regression fit. (b) Scatter plot of original data with full-precision linear regression fit.

in Figure 1(a), which shows the scatter plot of the processed data with the quantized linear fit. These metrics are very close to the R2 score of 0.9515 and MSE of 0.3232 achieved by the original, non-quantized model, depicted in Figure 1(b) (Scatter plot of original data). Once trained, the quantized model can then be used by the FPGA to make predictions on new input data, enabling efficient and automated linear regression directly in hardware with minimal loss in accuracy.

These fit weights were generated using an implementation in C to highlight the efficacy of the model before the implementation in FPGA. This approach allows for a highly efficient implementation as all operations can be performed in integer format. The algorithm for this implementation will be detailed in a subsequent section.

Algorithm

For 1D linear regression with n observations (x_i, y_i), the model is:

$$y = w_0 + w_1 x$$

For a single observation, this can be written as:

$$y_i = \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

The weights $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$ are typically found by solving the normal equation:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

where:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

The key matrix computations involved are:

$$\mathbf{X}^T\mathbf{X} = \begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix}$$

and

$$\mathbf{X}^T\mathbf{y} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix}$$

The inverse of $\mathbf{X}^T\mathbf{X}$ is calculated as:

$$(\mathbf{X}^T\mathbf{X})^{-1} = \frac{1}{n\sum x_i^2 - (\sum x_i)^2} \begin{bmatrix} \sum x_i^2 & -\sum x_i \\ -\sum x_i & n \end{bmatrix}$$

For datasets where x_i and y_i are 4-bit unsigned integers (i.e., $x_i, y_i \in \{0, 1, \ldots, 15\}$), the computation can be optimized by first calculating five critical integer accumulations:

$$\begin{aligned} S_1 &= n && \text{(count of observations)} \\ S_2 &= \sum_{i=1}^{n} x_i && \text{(sum of } x_i\text{)} \\ S_3 &= \sum_{i=1}^{n} y_i && \text{(sum of } y_i\text{)} \\ S_4 &= \sum_{i=1}^{n} x_i^2 && \text{(sum of } x_i^2\text{)} \\ S_5 &= \sum_{i=1}^{n} x_i y_i && \text{(sum of } x_i y_i\text{)} \end{aligned}$$

Considering the constraints that n is bounded by 512 and x_i, y_i are 4-bit unsigned integers, the bit-width requirements for these sums are as follows:

- $S_1 = n$: Requires 9 bits, as $2^8 < 512 \leq 2^9$.
- $S_2 = \sum x_i$ and $S_3 = \sum y_i$: The maximum value for x_i or y_i is 15. In the worst case, these sums can reach $512 \times 15 = 7680$. This requires $\lceil \log_2(7680 + 1) \rceil = 13$ bits. This can be conceptually understood as the sum of bits for the maximum input data (4 bits) and the maximum count (9 bits).

- $S_4 = \sum x_i^2$ and $S_5 = \sum x_i y_i$: The product x_i^2 or $x_i y_i$ can be up to $15 \times 15 = 225$. This intermediate product requires $\lceil \log_2(225 + 1) \rceil = 8$ bits (conceptually, the sum of bits of the two 4-bit operands, $4 + 4 = 8$). The accumulated sum can then reach $512 \times 225 = 115200$. This requires $\lceil \log_2(115200 + 1) \rceil = 17$ bits (conceptually, the sum of bits for the intermediate product (8 bits) and the maximum count (9 bits)).

These bit-width calculations are essential for designing the hardware accumulators with sufficient capacity to prevent overflow. Hence, we will design each of the sum registers in 17 bits.

These sums can be computed efficiently, for instance, using dedicated parallel blocks or pipelined reduction circuits, processing data as it streams (e.g., from an SD card). Since x_i and y_i are small integers, these sums will also be integers. The only floating-point operations required are the final divisions. The scaling factor, which is the reciprocal of the determinant $\det(\mathbf{X}^T\mathbf{X}) = S_1 S_4 - S_2^2$, is applied after all integer accumulations are complete.

We can implement a pipelined accumulation scheme where, for each incoming data pair (x_i, y_i), four parallel compute units update the sums in real-time:

$$\begin{aligned} S_2 &\leftarrow S_2 + x_i \\ S_3 &\leftarrow S_3 + y_i \\ S_4 &\leftarrow S_4 + x_i^2 \\ S_5 &\leftarrow S_5 + x_i y_i \end{aligned}$$

Simultaneously, S_1 is incremented by a counter with each data pair processed.

After all data points have been processed and the accumulations are complete, the regression coefficients w_0 and w_1 are calculated as:

$$\begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \frac{1}{S_1 S_4 - S_2^2} \begin{bmatrix} S_4 S_3 - S_2 S_5 \\ S_1 S_5 - S_2 S_3 \end{bmatrix}$$

Let $D = S_1 S_4 - S_2^2$ (the determinant), $N_0 = S_4 S_3 - S_2 S_5$, and $N_1 = S_1 S_5 - S_2 S_3$. All D, N_0, and N_1 are integers. Then:

$$w_0 = \frac{N_0}{D}$$
$$w_1 = \frac{N_1}{D}$$

The individual sum accumulators $(S_1, S_2, S_3, S_4, S_5)$ stored in registers can be processed by a combinational logic block to compute the three integer values (N_0, N_1, D). These three values are then used to compute the final floating-point weights w_0 and w_1.

Avalon System Components

Fig. 2 shows the system configuration inside Intel's Quartus Platform Designer (formerly Qsys) for a project named `soc_system`. This system integrates three key components: a

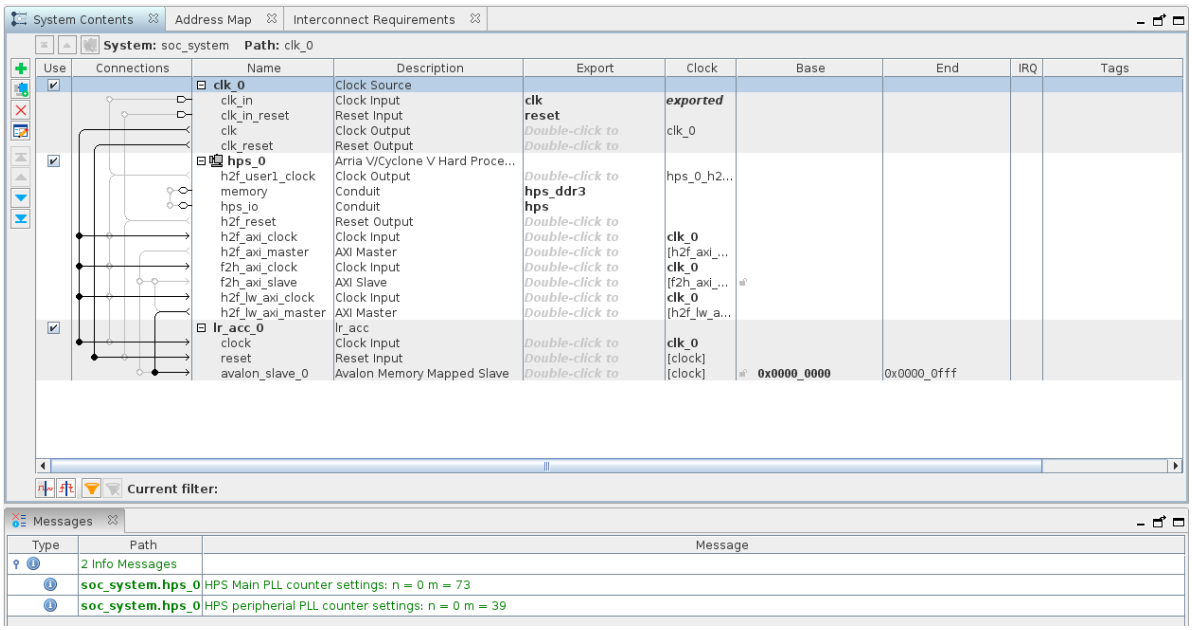


Figure 2: Avalon System Components

clock source (`clk_0`), a Hard Processor System (`hps_0`), and a custom hardware accelerator for linear regression (`lr_acc_0`). These components are connected using Avalon and AXI interconnects to enable communication between the HPS and FPGA fabric, allowing software running on the ARM processor to configure and interact with the hardware accelerator.

The clock source `clk_0` provides the main clock and reset signals to all other components in the design. These signals are exported to the top-level module so they can be driven externally, for example, from a PLL or onboard oscillator. The `hps_0` block represents the Cyclone V HPS (Hard Processor System), which includes dual ARM Cortex-A9 processors and a rich set of memory and IO interfaces. Through the lightweight AXI master interface (`h2f_lw_axi_master`), the processor can access memory-mapped peripherals in the FPGA fabric.

The `lr_acc_0` module is a custom Avalon slave component that implements a linear regression accelerator. It receives clock and reset from the clock source and is memory-mapped into the system address space starting at base address `0x0000_0000` with a 4KB range (ending at `0x0000_0FFF`). The HPS accesses this accelerator using the lightweight AXI master interface for tasks like sending training data, starting computations, and reading regression outputs.

- `clk_0` **(Clock Source):**
  - Provides clock and reset signals to the system.
  - Signals `clk` and `reset` are exported to the top-level design.
  - Drives the clock inputs of both the `lr_acc_0` and the HPS.

- `hps_0` **(Hard Processor System):**
  - Represents the ARM Cortex-A9 subsystem embedded in Cyclone V SoCs.
  - Interfaces include:
    - `h2f_axi_master`: For general AXI-based communication from HPS to FPGA.
    - `h2f_lw_axi_master`: Lightweight interface for mem-mapped peripheral access.
    - `f2h_axi_slave`: For FPGA-to-HPS communication.
    - Clock inputs/outputs for synchronization with the fabric.
  - Also interfaces with external memory (e.g., DDR) and IO via exported conduits.
- `lr_acc_0` **(Linear Regression Accelerator):**
  - Custom Avalon memory-mapped slave IP core.
  - Receives clock and reset from `clk_0`.
  - Connected to the HPS via `h2f_lw_axi_master`.
  - Mapped to address space `0x0000_0000` to `0x0000_0FFF` (4KB).
  - Implements logic to perform linear regression computations.

Software-Hardware Avalon Interface

Fig. 5 shows the **Signals & Interfaces** tab for the `avalon_slave_0` interface in Intel Quartus Platform Designer (Qsys) defines the boundary between software and hardware. This interface is responsible for connecting the ARM processor subsystem (HPS) to a custom hardware peripheral, which in this case is likely the *Linear Regression Accelerator*.

The `avalon_slave_0` is configured as an **Avalon Memory-Mapped Slave Interface**, which allows the processor to communicate with FPGA logic using standard memory read and write instructions. Through this interface, software running on the processor can send commands, write input data, and read computation results directly from the hardware peripheral via mapped memory addresses. The interface includes all the standard signals required for such communication, including `read`, `write`, `chipselect`, `address`, `writedata`, and `readdata`. Each of these signals plays a critical role in enabling synchronous, low-latency, memory-mapped access between the processor and the accelerator logic within the FPGA. In the center block diagram of `avalon_slave_0`, the following signals are defined. These represent the interface between the processor (HPS) and a custom hardware peripheral, such as a Linear Regression Accelerator.

Signal Descriptions

- `writedata[31:0]`: 32-bit data written by the processor to the hardware module.

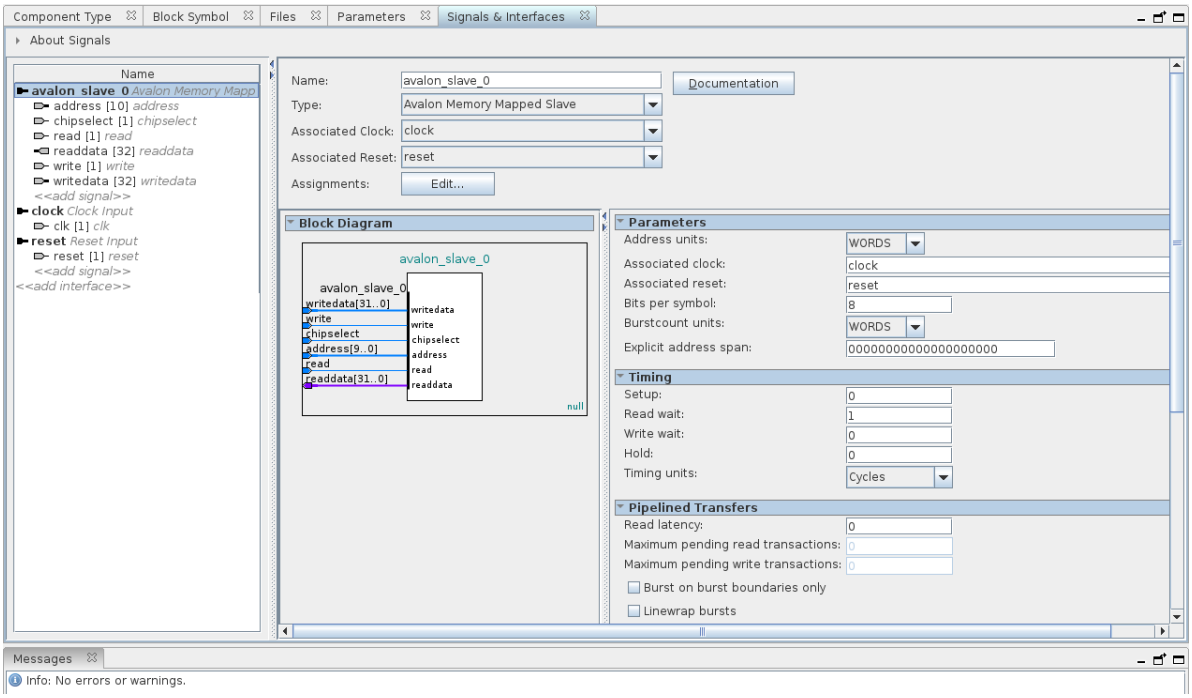


Figure 3: Software-Hardware Avalon Interface

- `write`: Single-bit control signal that indicates a write transaction is requested.
- `chipselect`: Asserted when the memory-mapped region assigned to this slave is being accessed by the processor.
- `address[9:0]`: A 10-bit address line allowing access to 1024 unique words within the module.
- `read`: Single-bit control signal indicating a read transaction is requested.
- `readdata[31:0]`: 32-bit data read from the hardware module by the processor.

These signals together allow the CPU (typically an ARM Cortex-A9 core in the HPS) to communicate with the custom logic through memory-mapped I/O. Standard processor instructions such as `ldr`, `str`, or Linux-based MMIO access can be used to interact with the peripheral.

i. Software-Hardware Interface Configuration (Right Panel)

The configuration settings in the Parameters panel define how the Avalon Memory-Mapped Slave interface behaves:

- **Associated Clock**: `clock` — All signal activities on this interface are synchronized to this signal.

– **Associated Reset**: `reset` — Used to reset the hardware registers inside the peripheral.
– **Address Units**: Set to `WORDS`, meaning each address increment maps to a 32-bit word rather than a byte.
– **Bits per Symbol**: 8 — Byte-level granularity, although accesses are word-based.
– **Explicit Address Span**: Left empty in this instance; can be filled to override auto-assigned address range.

ii. Timing Characteristics

The following parameters specify the cycle-level behavior of transactions over this interface:

– **Setup**: 0 cycles — No setup delay is required.
– **Read Wait**: 1 cycle — The `readdata` will be valid one cycle after asserting `read`.
– **Write Wait**: 0 cycles — Write transactions complete in the same cycle they are issued.
– **Hold**: 0 cycles — No additional hold time required after a transaction.
– **Timing Units**: All timing values are expressed in clock cycles.

Block Diagrams

Linear Regression Main Pipeline

From Fig. 4, the linear regression accelerator uses a set of output stages to convey partial and final computation results through a memory-mapped interface. Each of these outputs is accessible via specific addresses decoded using bits a_9 to a_0. These output registers—such as `N0`, `N1`, `D`, and `S1--S5`—hold values calculated across the regression pipeline, which ultimately computes the coefficients w_0 and w_1. This address mapping is crucial for software to retrieve intermediate results or verify convergence across the stages.

Linear Regression Accelerator 8b Block

Fig. 5 illustrates a **pipelined binary adder tree** designed to compute the sum of 16 unsigned 18-bit inputs, labeled `op0` through `opF`. The architecture employs a hierarchical reduction approach, where pairs of inputs are incrementally summed using multiple

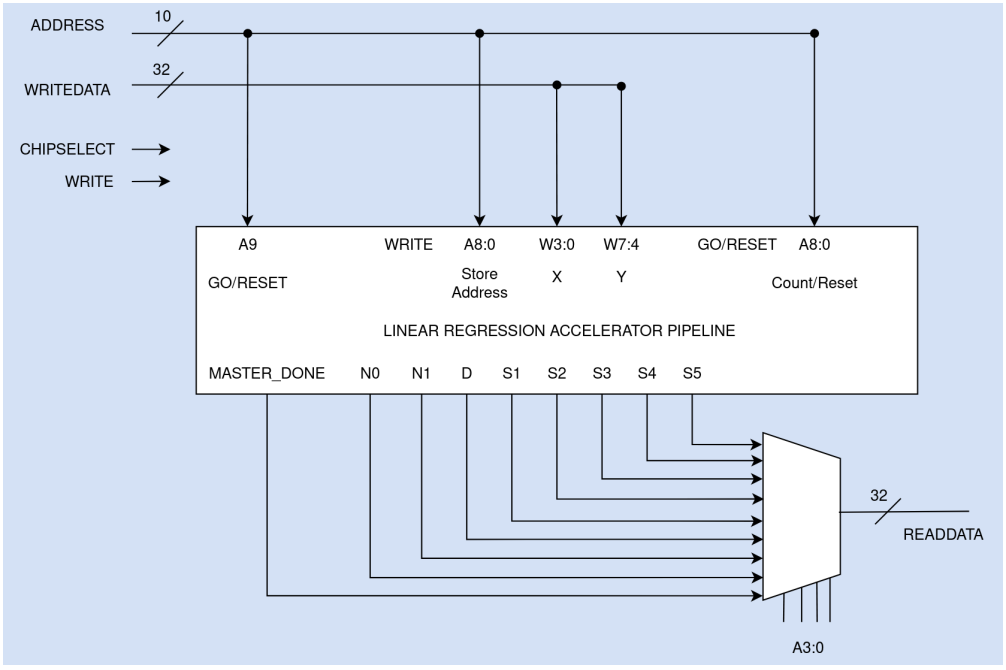


Figure 4: Linear Regression Main Pipeline

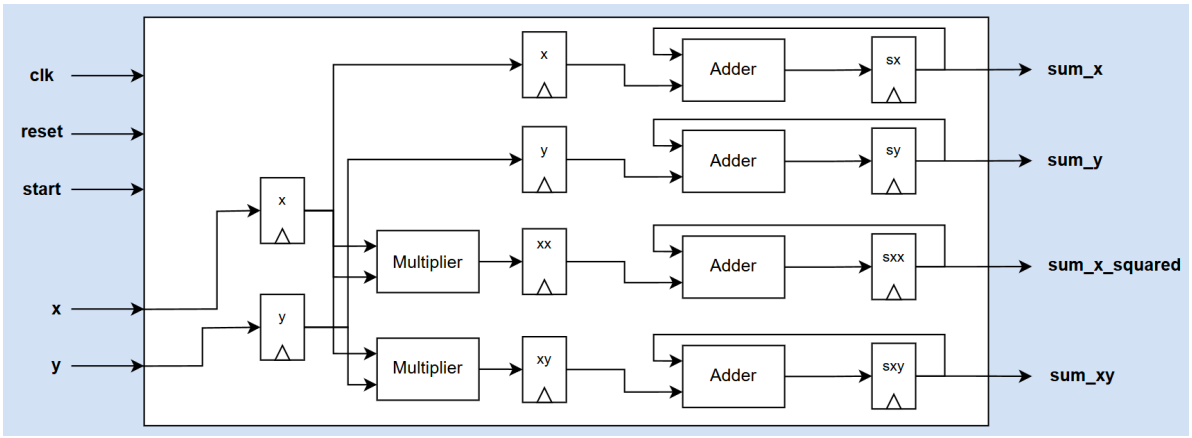


Figure 5: Linear Regression Accelerator 8b Block

layers of adders. Registers are inserted between each stage to form a fully pipelined structure, significantly enhancing throughput and enabling higher clock frequencies. This is especially beneficial for FPGA implementations, where timing closure and parallelism are critical.

Each stage reduces the number of operands by half. Initially, eight pairwise adders reduce the 16 inputs into 8 intermediate values. These values are then fed into four second-level adders, reducing them to 4 values, and so on, until the final result is produced. The output, labeled `result`, is a single 18-bit value representing the total sum.

Hierarchy and Pipelining Structure

Inputs:

- 16 inputs: `op0` to `opF`
- Each input is 18 bits wide

Stage 1 — Pairwise Addition:

- 8 adders compute:

$$\texttt{(op0 + op1)}, \quad \texttt{(op2 + op3)}, \quad \ldots, \quad \texttt{(opE + opF)}$$

- Each output is stored in a pipeline register

Stage 2 — Second-Level Addition:

- 4 adders compute:

$$\begin{aligned}
\text{Sum1} &= (\texttt{op0} + \texttt{op1}) + (\texttt{op2} + \texttt{op3}) \\
\text{Sum2} &= (\texttt{op4} + \texttt{op5}) + (\texttt{op6} + \texttt{op7}) \\
\text{Sum3} &= (\texttt{op8} + \texttt{op9}) + (\texttt{opA} + \texttt{opB}) \\
\text{Sum4} &= (\texttt{opC} + \texttt{opD}) + (\texttt{opE} + \texttt{opF})
\end{aligned}$$

- Outputs are again registered

Stage 3 — Third-Level Addition:

- 2 adders compute:

$$\begin{aligned}
\text{SumA} &= \text{Sum1} + \text{Sum2} \\
\text{SumB} &= \text{Sum3} + \text{Sum4}
\end{aligned}$$

- Outputs are pipelined to the final stage

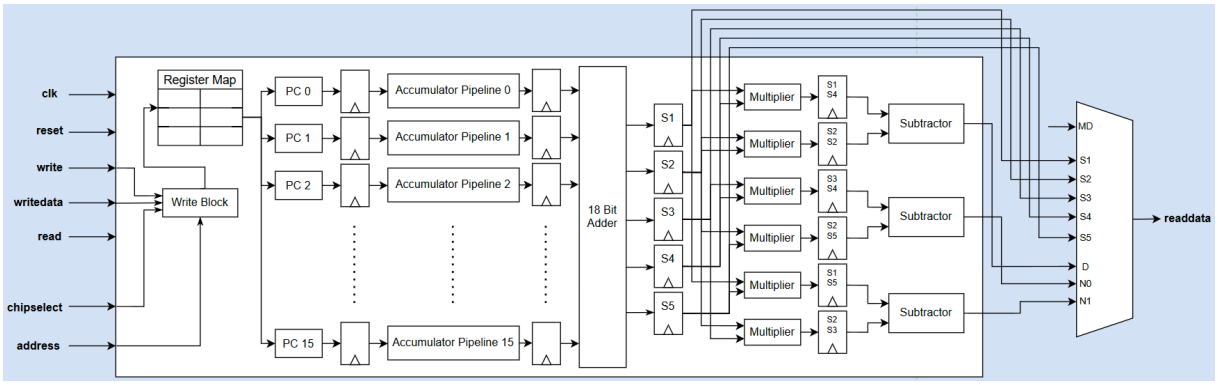


Figure 6: Linear Regression Accelerator 8b Block 512

LR Accumulator 8b 512

Fig. 6 illustrates a detailed architecture of a **Linear Regression Accelerator** designed for hardware implementation, typically on an FPGA. The accelerator performs *feature accumulation*, *intermediate statistical computation* (S_1–S_5), and *final coefficient evaluation* using a pipelined datapath composed of **accumulators**, **adders**, **multipliers**, and **subtractors**. The module is memory-mapped, allowing interaction with a processor through a standard Avalon or AXI-like interface using control and status registers.

The module operates under the control of a clock signal (`clk`) and is reset by a synchronous reset signal (`reset`). Memory-mapped I/O signals, such as `address`, `writedata`, `read`, `write`, and `chipselect`, facilitate communication between the processor and the accelerator. At the core of the design are a **Register Map** and a **Write Block**, which receive data from the processor and route it to 16 parallel **Accumulator Pipelines**, labeled from `PC0` to `PC15`. These pipelines accumulate partial sums from incoming training data across multiple cycles.

The accumulated outputs from all pipelines feed into an **18-bit Adder Tree**, which computes global statistics such as:

$$S_1 = \sum x, \quad S_2 = \sum y, \quad S_3 = \sum x^2, \quad S_4 = \sum xy, \quad S_5 = \sum y^2$$

These intermediate statistical values are then passed through a series of **multipliers** and **subtractors** to calculate the higher-level regression terms:

$$N_0, \quad N_1, \quad \text{and} \quad D$$

where N_0 and N_1 are the numerators used in the computation of the slope and intercept of the regression line, and D is the common denominator.

Finally, all computed values, along with a status signal `MD` (Master Done), are routed to a **Multiplexer (MUX)**. This MUX connects to the `readdata` bus, enabling the processor to selectively access internal results based on the supplied address.

Final Stage — Root Addition:

- Final adder computes:

$$\texttt{result} = \text{SumA} + \text{SumB}$$

- The result is stored in a final register before output

Key Design Features

- **Fully Pipelined:** Every adder is followed by a register, enabling high-throughput processing.
- **Balanced Tree Structure:** Reduces fanout and critical path delay, ideal for timing-optimized implementations.
- **Uniform Bit Width:** All operations are performed with 18-bit unsigned values, maintaining consistency.
- **FPGA-Friendly Design:** Optimized for synthesis and implementation in modern FPGAs using DSP slices and logic blocks.

Combinational 18-bit Adder

Fig. 7 shows a **pipelined binary adder tree** designed to compute the sum of 16 parallel 18-bit unsigned input values, labeled from `op0` to `opF`. Each input is first registered to align timing and introduce pipelining. The architecture groups these inputs into adjacent pairs and sums them using multiple stages of binary adders. The tree structure ensures that additions are performed with *logarithmic depth*, reducing the 16 inputs down to a single final output in four addition stages.

Each addition operation is followed by a pipeline register, allowing every stage to operate in a fully pipelined manner. This design ensures a high-throughput datapath in which one sum result can be computed on every clock cycle once the pipeline is filled. The final 18-bit result is produced at the `result` output port, making the design suitable for applications in real-time signal processing or statistical summation workloads.

The adder tree employs a balanced binary reduction strategy with:

- **Stage 1**: 8 parallel adders for 8 pairwise additions.
- **Stage 2**: 4 adders for the second-level reduction.
- **Stage 3**: 2 adders to compute two intermediate sums.
- **Stage 4**: Final adder to compute the total sum.

By fully pipelining each stage, the design achieves:

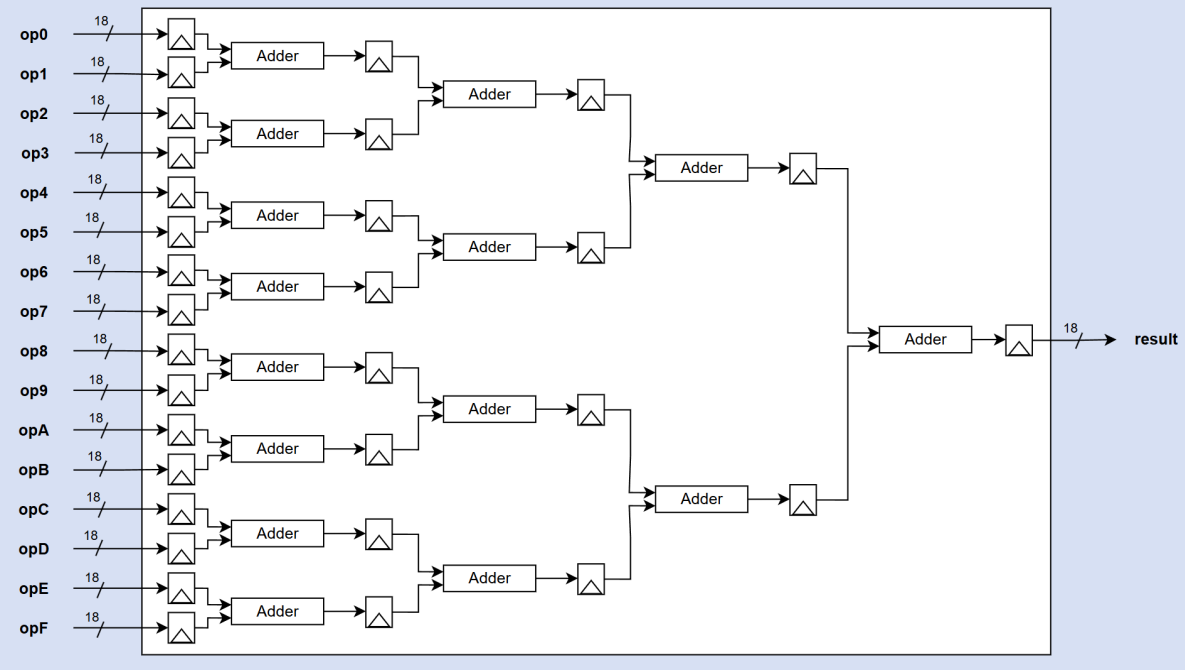


Figure 7: Combinational 18-bit Adder

- Minimal critical path delay between stages.
- High clock frequency compatibility for FPGA implementations.
- Improved timing closure with reduced combinational fanout.
- Scalability to wider bit-widths or more input operands with minor structural changes.

This module is used within the Linear Regression Accelerator as a core summation unit to compute statistical accumulations such as $\sum x$, $\sum y$, $\sum x^2$, and $\sum xy$, leveraging its efficient structure for fast, large-scale accumulation operations in hardware.

Register Map Tables

It defines how software (e.g., a CPU or host controller) can interact with hardware using memory-mapped registers. It acts as a bridge between the software control layer and the hardware datapath by exposing internal hardware states, controls, and outputs at specific memory addresses. This abstraction enables software to configure, monitor, and trigger hardware operations without direct intervention in signal-level control. It also facilitates easy integration into driver frameworks and high-level control applications.

a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
0	0	0	0	0	0	0	0	0	0	Master Done (inform completion of operation)
0	0	0	0	0	0	0	0	0	1	N_0 (Used for w_0 computation)
0	0	0	0	0	0	0	0	1	0	N_1 (Used for w_1 computation)
0	0	0	0	0	0	0	0	1	1	D (Used for both w_0 and w_1)
0	0	0	0	0	0	0	1	0	0	S_1 (Provided for users who want all outputs)
0	0	0	0	0	0	0	1	0	1	S_2 (Provided for users who want all outputs)
0	0	0	0	0	0	0	1	1	0	S_3 (Provided for users who want all outputs)
0	0	0	0	0	0	0	1	1	1	S_4 (Provided for users who want all outputs)
0	0	0	0	0	0	1	0	0	0	S_5 (Provided for users who want all outputs)

Figure 8: Read Address Mapping

Read Address Mapping

Fig 8 shows the 10-bit address encoding used to access the results. Each address corresponds to a specific pipeline output or flag:

- Address `0x000` selects the `MASTER_DONE` flag, used to indicate completion of the regression computation.
- Address `0x001` maps to `N0`, the partial sum relevant for w_0's calculation.
- Address `0x002` selects `N1`, used for computing w_1.
- Address `0x003` accesses `D`, a value reused by both coefficient computations.
- Addresses `0x004` through `0x008` represent stages `S1` through `S5`, and are mainly provided for debug or verbose mode where all pipeline outputs are required.

Each of these registers is mapped via a read multiplexer, controlled by address bits a_3 to a_0, and the result is driven onto the `READDATA` bus. The output is valid only when the CPU initiates a read operation (i.e., when `WRITE = 0`) to one of these addresses with `CHIPSELECT` asserted. Since the accelerator module is memory-mapped, the CPU communicates via standard read/write protocols, ensuring synchronization without interrupt handling.

To support the complete address decoding logic, the accelerator treats all addresses with $a_9 = 0$ and `WRITE = 0` as read operations.

Read Register Map

fig. 9 provides a memory map of readable output registers of the accelerator, showing how each pipeline stage's result or status signal is placed in memory. These offsets represent addresses accessible via the Avalon interface by the processor.
The table clearly distinguishes between control/status bits (like `Master Done`) and result values (like N_0, N_1, D, and S_1–S_5). The `Master Done` flag is encoded at offset 0, bit 0, with

Offset (Hex)	Bit	Description
0	31:1	Unused
	0	Master Done
1	31:0	Signed N_0
2	31:0	Signed N_1
3	31:0	Signed D
4	31:0	Unsigned S_1
5	31:0	Unsigned S_2
6	31:0	Unsigned S_3
7	31:0	Unsigned S_4
8	31:0	Unsigned S_5

Figure 9: Read Register Map

the remaining bits unused. The rest of the offsets (1 through 8) contain signed or unsigned 32-bit results from various computation stages of the accelerator.

- **Offset 0:**
  - Bit 0: `Master Done` (set to 1 if the accelerator has completed computation).
  - Bits 31–1: Unused.
- **Offsets 1–3 (Signed Values):**
  - Offset 1: Signed result of N_0 (used in computing w_0).
  - Offset 2: Signed result of N_1 (used in computing w_1).
  - Offset 3: Signed result D (common to both weights).
- **Offsets 4–8 (Unsigned Values):**
  - Represent unsigned outputs S_1 through S_5.
  - Provided primarily for full-output tracing or debugging purposes.

a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
0	0	0	0	0	0	0	0	0	0	Write to Address 0 in Memory ($a_{8:0} = 1$)
0	0	0	0	0	0	0	0	0	1	Write to Address 1 in Memory ($a_{8:0} = 1$)
0	0	0	0	0	0	0	0	1	0	Write to Address 2 in Memory ($a_{8:0} = 2$)
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	1	1	1	1	1	1	1	1	1	Write to Address 511 in Memory ($a_{8:0} = 511$)
1	0	0	0	0	0	0	0	0	0	Write Reset
1	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	Write Go with Count of Data y

Figure 10: Write Address Encoding

Write Address Encoding

Fig. 10 illustrates the address decoding scheme for write operations in the Linear Regression Accelerator. The 10-bit address field (a_9 to a_0) determines whether a memory address is being written to, or a control operation (reset or go) is being issued. When the most significant bit $a_9 = 0$, the operation targets the training memory and stores input data (typically X and Y pairs). The address lines $a_8 : 0$ define the specific memory location, allowing for up to 512 entries. When $a_9 = 1$, the address is interpreted as a control command—either to reset the accelerator or to trigger the computation (go), where the lower 8 bits specify the number of input samples (count).

- **Data Write Operations:**
  - $a_9 = 0 \rightarrow$ Memory write mode.
  - $a_8 : 0$ defines the memory address (range: 0 to 511).
  - Used to write training data into internal memory.
- **Control Operations:**
  - $a_9 = 1$ and all lower bits $= 0 \rightarrow$ Write Reset.
  - $a_9 = 1$ and remaining bits = count value $y[7 : 0] \rightarrow$ Write Go (start operation with input count y).

Write Register Map

Fig. 11 defines the memory layout for training data in the Linear Regression Accelerator. The range from offset `0x000` to `0x1FF` is used to store input pairs (x, y) for regression computation. Each 32-bit word is partially used: only the lower 8 bits are valid, where

Offset (Hex)	Bit	Description
$0 - 1FF$	31:8	Unused
	7:4	y
	3:0	x
$200 - 2FF$	31:0	Unused

Figure 11: Write Register Map

bits `[3:0]` encode x and bits `[7:4]` encode y. The upper bits `[31:8]` are unused and may be written as zeros. The subsequent address space from `0x200` to `0x2FF` is completely unused and is likely reserved for padding, alignment, or future extension.

This compact representation is optimized for 4-bit quantized input values, minimizing memory usage while maintaining high data throughput. The structure also allows software to stream input pairs into the accelerator sequentially, simplifying buffer management and reducing logic complexity. Such a layout is particularly well-suited for FPGA-based accelerators, where memory efficiency and deterministic access patterns are critical.

- **Offset Range** `0x000 { 0x1FF`**:**
  - Total of 512 entries.
  - Bits `[3:0]`: Encodes input x.
  - Bits `[7:4]`: Encodes input y.
  - Bits `[31:8]`: Unused.
- **Offset Range** `0x200 { 0x2FF`**:**
  - All bits `[31:0]` are unused.
  - Reserved for future use or address padding.

Timing Diagrams

8-bit Block Simulation Analysis

The `lr_acc_8b` module implements a pipelined datapath designed to accumulate essential statistical values for linear regression, using 8-bit input streams. The included

simulation results validate the correctness and efficiency of the internal datapath.

Fitter Summary

<<Filter>>

Fitter Status	Successful - Tue May 13 02:15:41 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	lr_acc_8b_wrapper
Top-level Entity Name	lr_acc_8b_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	74 / 32,070 (< 1 %)
Total registers	193
Total pins	76 / 457 (17 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total RAM Blocks	0 / 397 (0 %)
Total DSP Blocks	2 / 87 (2 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 12: Fitter Analysis

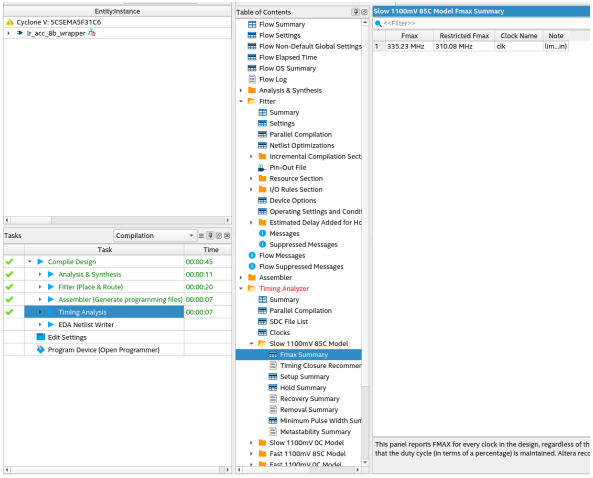


Figure 13: Fmax Analysis

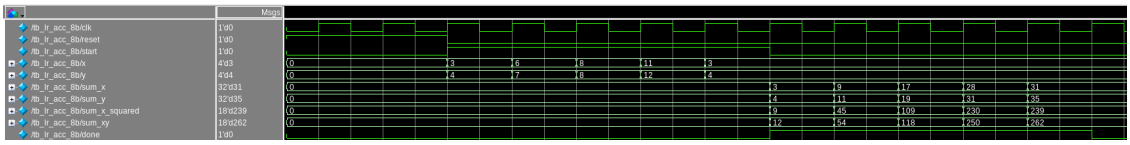


Figure 14: Timing Diagram

The `lr_acc_8b` module implements a pipelined datapath designed to accumulate essential statistical values for linear regression, using 8-bit input streams. The included waveform captures a representative simulation cycle-by-cycle, validating correct temporal behavior and functional correctness.

- **Inputs** (x, y): Sequential data pairs applied after a reset. Timing is aligned with the testbench vectors to ensure synchronization with pipeline stages.
- **sum_x / sum_y**: These signals accumulate the total of all observed x and y values, respectively, and are used to calculate the mean and other statistical moments.
- **sum_x_squared**: Validates the squaring and accumulation of each x sample, representing $\sum x^2$, needed in slope denominator computation.
- **sum_XY**: Captures the accumulated sum of products $x \cdot y$, forming the numerator for the slope ($m = \frac{\sum(xy) - n\bar{x}\bar{y}}{\sum(x^2) - n\bar{x}^2}$).

This block is essential for real-time regression analysis, offering throughput-optimized, low-bitwidth accumulation suitable for embedded and resource-constrained environments. The successful simulation confirms the block's readiness for integration into the full datapath of the regression accelerator.

Pipelined Multiplier Simulation Analysis

Fitter Summary

<<Filter>>

Fitter Status	Successful - Tue May 13 01:01:02 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	multiplier_wrapper
Top-level Entity Name	multiplier_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	19 / 32,070 (< 1 %)
Total registers	64
Total pins	39 / 457 (9 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total RAM Blocks	0 / 397 (0 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 15: Fitter Analysis

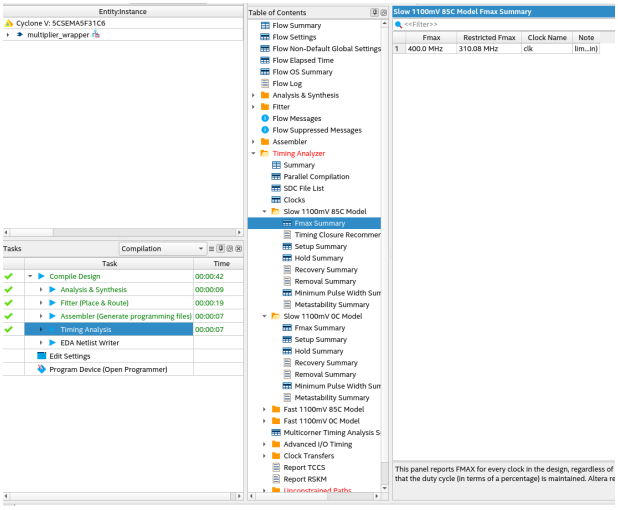


Figure 16: Fmax Analysis

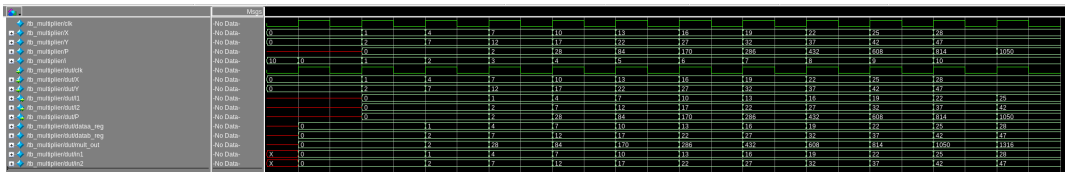


Figure 17: Timing Diagram

This simulation verifies the behavior of a pipelined multiplier module (`tb_multiplier`) receiving sequential inputs `X` and `Y`. Key signals demonstrate staged pipeline computation and throughput efficiency.

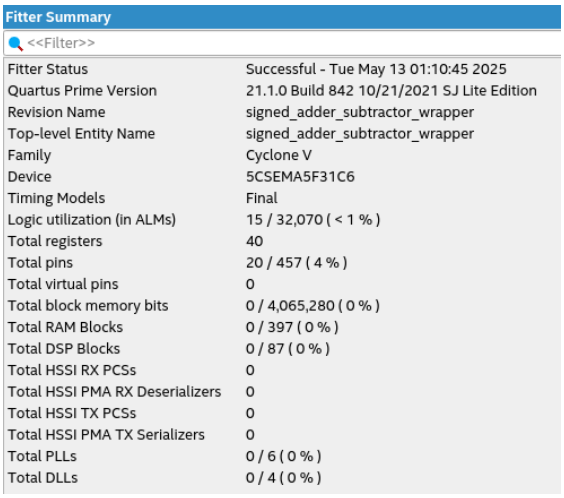
- **Inputs:** `X` and `Y` increment per clock cycle, latched on rising edges.
- **Pipelining:** Internal signals (`dut1`, `dut2`, `data_reg`) confirm multi-stage buffering.
- **Output:** `mult_out` delivers $X \cdot Y$ with $\sim$3-cycle latency. Examples:
  - $X = 4,\ Y = 7 \Rightarrow$ `mult_out` $= 28$
  - $X = 25,\ Y = 42 \Rightarrow$ `mult_out` $= 1050$
  - $X = 38,\ Y = 34.6 \Rightarrow$ `mult_out` $= 1316$
- **Conclusion:** Accurate pipelined multiplication with consistent throughput and result stability.

Signed Adder/Subtractor Simulation Analysis

The simulation waveform in Figure . illustrates the behavior of the `tb_signed_adder_subtractor` module, which performs signed arithmetic based on a control signal. The inputs `dataa` and `datab` are 16-bit signed values, and the `add_sub` signal determines whether an addition (`add_sub = 0`) or subtraction (`add_sub = 1`) is performed. The `result` output reflects the correct operation for each input pair and control signal setting.

For example, when `dataa = 62756`, `datab = 40577`, and `add_sub = 1`, the module correctly computes the result `22179` (`dataa - datab`). The internal DUT signals (`dut/dataa`, `dut/datab`, and `dut/result`) match the top-level signals, verifying proper signal propagation. Additionally, the `expected` output matches `result` across all cycles, confirming functional correctness.

All signal transitions are aligned with the rising clock edge, demonstrating proper synchronous design behavior. The pipeline exhibits no glitches or misalignments, validating the timing integrity of the module. This confirms that the signed adder/subtractor operates reliably under pipelined conditions with accurate arithmetic and control logic.



Fitter Summary	
Fitter Status	Successful - Tue May 13 01:10:45 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	signed_adder_subtractor_wrapper
Top-level Entity Name	signed_adder_subtractor_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	15 / 32,070 (< 1 %)
Total registers	40
Total pins	20 / 457 (4 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total RAM Blocks	0 / 397 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 18: Fitter Analysis

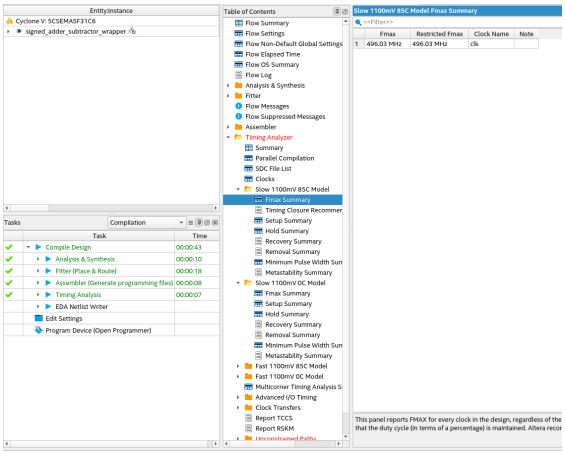


figure 19: Fmax Analysis

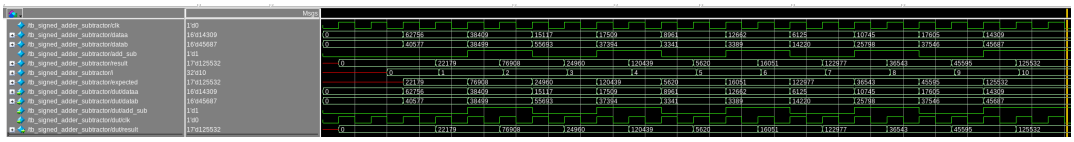


Figure 20: Timing Diagram

Single-Port RAM Timing Analysis

The timing diagram in Fig. shows the behavior of the `tb_single_port_ram` module during sequential read operations. The address line increments with each rising clock

edge, and the corresponding data appears at the output `q` after a one-cycle delay, confirming typical synchronous single-port RAM behavior. The output values align with the initialized memory contents, validating functional correctness.

- **1-cycle latency:** Output data appears exactly one clock after the address input.
- **Synchronous operation:** All transitions are clock-aligned.
- **Full address sweep:** Addresses range from 0 to 511, covering the entire memory space.
- **Correct outputs:** Data values match memory initialization, proving proper read access.
- **Clean waveform:** No glitches or misaligned transitions are observed.

Fitter Summary

<<Filter>>

Fitter Status	Successful - Tue May 13 00:24:14 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	single_port_ram_with_init
Top-level Entity Name	memory_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	5 / 32,070 (< 1 %)
Total registers	12
Total pins	11 / 457 (2 %)
Total virtual pins	0
Total block memory bits	256 / 4,065,280 (< 1 %)
Total RAM Blocks	1 / 397 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 21: Fitter Analysis

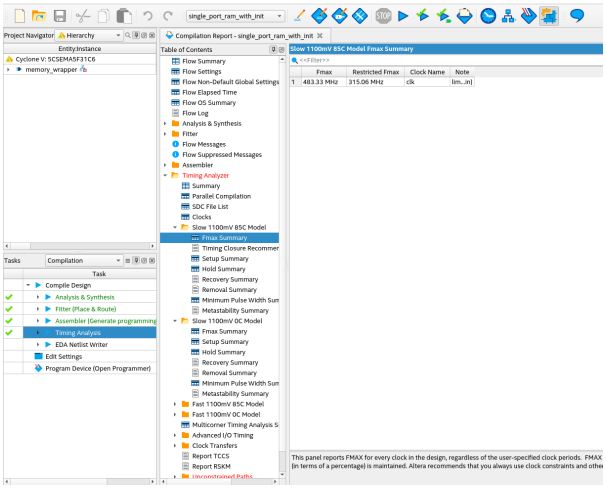


Figure 22: Fmax Analysis

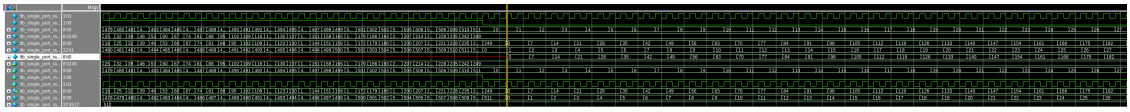


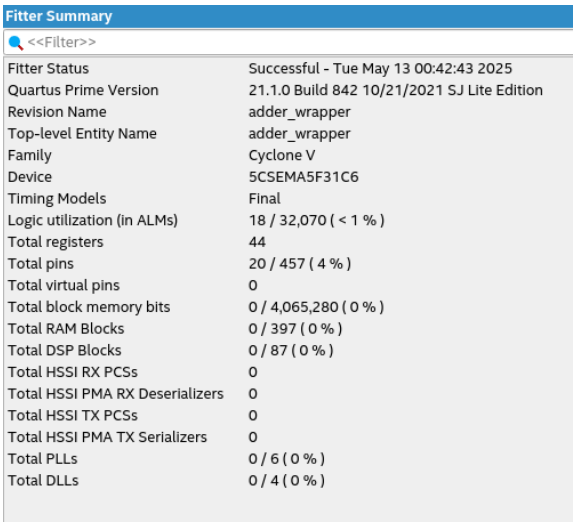
Figure 23: Timing Diagram of Sequential Read Access

Unsigned Adder Timing Analysis

The waveform in Fig. validates the behavior of the `tb_unsigned_adder` module under continuous stimulus. The input operands `dataa` and `datab` increment on each rising

clock edge, and the output `result` reflects their unsigned sum. The output is synchronized to the clock and verified against `expected` values, ensuring correctness. With the carry-in `cin` signal held at zero throughout, this test validates simple unsigned addition.

- **Operand Sweep:** Inputs range from 0–155, covering a wide span of test cases.
- **Correct Output:** `result` matches the expected sum at every clock cycle.
- **Clock Sync:** All transitions are aligned with rising edges of `clk`.
- **No Carry-In Effect:** With `cin = 0`, the test checks pure two-input addition.
- **DUT Integrity:** `dut/result` and top-level `result` signals match exactly.



Fitter Summary

<<Filter>>

Fitter Status	Successful - Tue May 13 00:42:43 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	adder_wrapper
Top-level Entity Name	adder_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	18 / 32,070 (< 1 %)
Total registers	44
Total pins	20 / 457 (4 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total RAM Blocks	0 / 397 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 24: Fitter Analysis

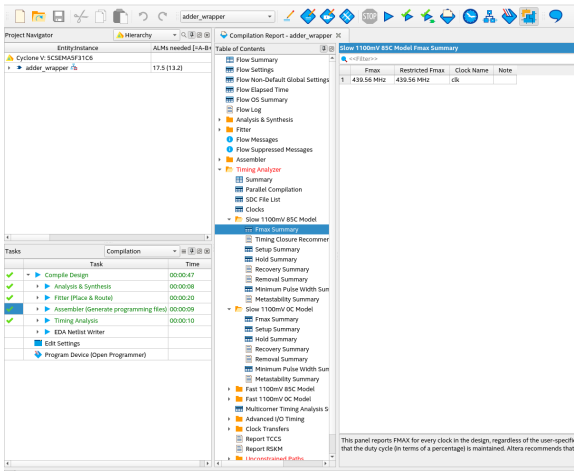


Figure 25: Fmax Analysis

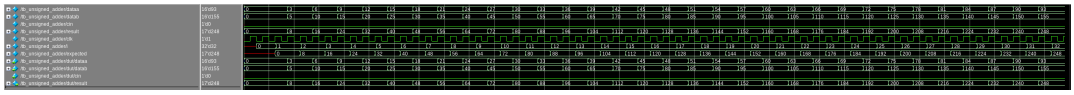


Figure 26: Waveform of unsigned adder showing sum result and DUT agreement

18-bit Combinational Adder Timing Analysis

The `comb_adder_18b` module implements a fully combinational 18-bit adder tree, accepting sixteen unsigned operands (`op0` to `opF`) as inputs. These are grouped pairwise and summed hierarchically in multiple levels—first forming intermediate sums such as `op01`, `op23`, and so on—until a total result is produced. The final output signal matches both `expected_sum` and `captured_result`, confirming functional correctness.

Since the output is computed within the same cycle as the `adder_start` signal, the design demonstrates true combinational behavior with zero latency. This structure is particularly beneficial in datapaths where low-latency parallel accumulation is required.

Since the output is computed within the same cycle as the `adder_start` signal, the design demonstrates true combinational behavior with zero latency. This structure is particularly beneficial in datapaths where low-latency parallel accumulation is required.

- **Zero-Latency Result:** Output is computed in the same cycle as the start signal.
- **Adder Tree Hierarchy:** Pairwise additions build intermediate results up to a 240-sum.
- **Waveform Matching:** All output signals match expected and captured values.
- **Synthesis Success:** Design meets timing and resource constraints based on fitter and Fmax analysis.

Fitter Summary

<<Filter>>

Fitter Status	Successful - Tue May 13 03:01:05 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	comb_adder_18b_wrapper
Top-level Entity Name	comb_adder_18b_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	112 / 32,070 (< 1 %)
Total registers	231
Total pins	22 / 457 (5 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total RAM Blocks	0 / 397 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 27: Fitter Analysis

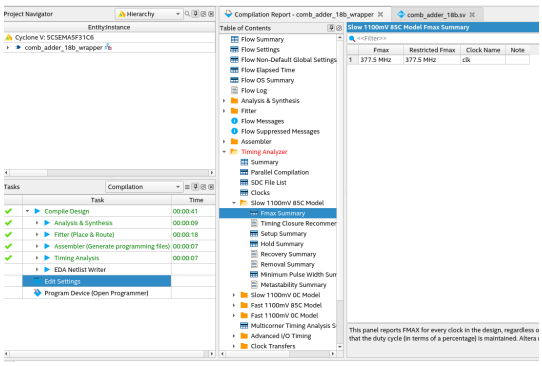


Figure 28: Fmax Analysis

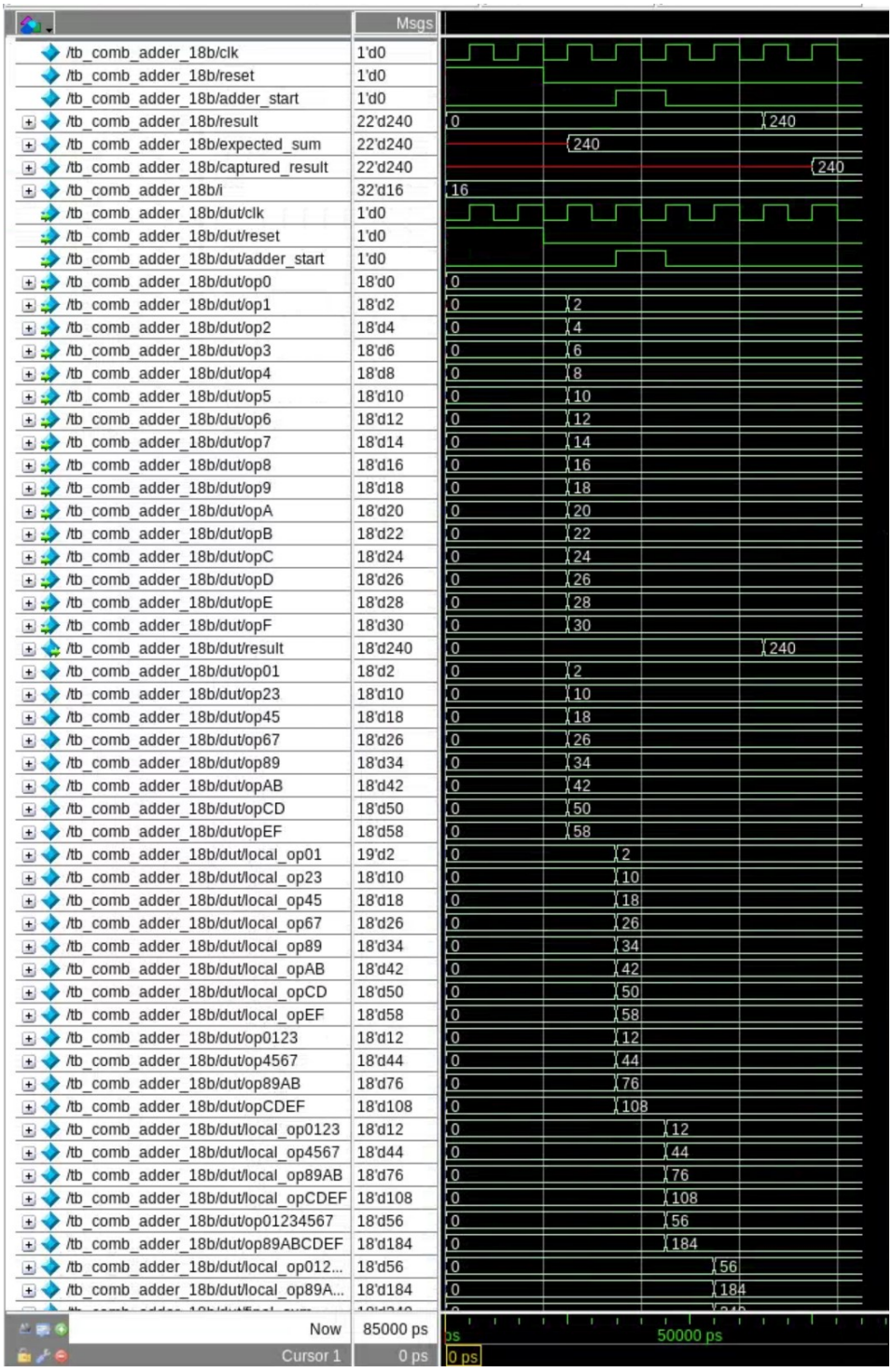


Figure 29: Timing diagram of 18-bit combinational adder tree showing intermediate and final sums

Verilog Code

comb_adder_18b.sv

```
`include "lr_acc.svh"


module comb_adder_18b (

    //Port Structure
    input clk,
    input reset,
    input logic adder_start,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op0,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op1,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op2,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op3,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op4,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op5,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op6,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op7,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op8,
    input logic [`ADDER_IN_SIZE_18b-1:0]   op9,
    input logic [`ADDER_IN_SIZE_18b-1:0]   opA,
    input logic [`ADDER_IN_SIZE_18b-1:0]   opB,
    input logic [`ADDER_IN_SIZE_18b-1:0]   opC,
    input logic [`ADDER_IN_SIZE_18b-1:0]   opD,
    input logic [`ADDER_IN_SIZE_18b-1:0]   opE,
    input logic [`ADDER_IN_SIZE_18b-1:0]   opF,

    output logic [`ADDER_OUT_SIZE_18b-1:0] result
);

    logic [`ADDER_IN_SIZE_18b-1:0]   op01;
    logic [`ADDER_IN_SIZE_18b-1:0]   op23;
    logic [`ADDER_IN_SIZE_18b-1:0]   op45;
    logic [`ADDER_IN_SIZE_18b-1:0]   op67;
    logic [`ADDER_IN_SIZE_18b-1:0]   op89;
    logic [`ADDER_IN_SIZE_18b-1:0]   opAB;
    logic [`ADDER_IN_SIZE_18b-1:0]   opCD;
    logic [`ADDER_IN_SIZE_18b-1:0]   opEF;

    unsigned_adder #(.WIDTH(18)) adder_inst0 (
        .dataa(op0),
        .datab(op1),
        .cin('0),
        .result(op01)
    );

    unsigned_adder #(.WIDTH(18)) adder_inst1 (
        .dataa(op2),
```

```
    .datab(op3),
    .cin('0),
    .result(op23)
);

unsigned_adder #(.WIDTH(18)) adder_inst2 (
    .dataa(op4),
    .datab(op5),
    .cin('0),
    .result(op45)
);

unsigned_adder #(.WIDTH(18)) adder_inst3 (
    .dataa(op6),
    .datab(op7),
    .cin('0),
    .result(op67)
);

unsigned_adder #(.WIDTH(18)) adder_inst4 (
    .dataa(op8),
    .datab(op9),
    .cin('0),
    .result(op89)
);

unsigned_adder #(.WIDTH(18)) adder_inst5 (
    .dataa(opA),
    .datab(opB),
    .cin('0),
    .result(opAB)
);

unsigned_adder #(.WIDTH(18)) adder_inst6 (
    .dataa(opC),
    .datab(opD),
    .cin('0),
    .result(opCD)
);

unsigned_adder #(.WIDTH(18)) adder_inst7 (
    .dataa(opE),
    .datab(opF),
    .cin('0),
    .result(opEF)
);

logic [`ADDER_IN_SIZE_18b:0]   local_op01;
logic [`ADDER_IN_SIZE_18b-1:0]   local_op23;
logic [`ADDER_IN_SIZE_18b-1:0]   local_op45;
logic [`ADDER_IN_SIZE_18b-1:0]   local_op67;
```

```
logic [`ADDER_IN_SIZE_18b-1:0]   local_op89;
logic [`ADDER_IN_SIZE_18b-1:0]   local_opAB;
logic [`ADDER_IN_SIZE_18b-1:0]   local_opCD;
logic [`ADDER_IN_SIZE_18b-1:0]   local_opEF;

 always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        local_op01 <= '0;
        local_op23 <= '0;
        local_op45 <= '0;
        local_op67 <= '0;
        local_op89 <= '0;
        local_opAB <= '0;
        local_opCD <= '0;
        local_opEF <= '0;
    end else begin
        local_op01 <= op01;
        local_op23 <= op23;
        local_op45 <= op45;
        local_op67 <= op67;
        local_op89 <= op89;
        local_opAB <= opAB;
        local_opCD <= opCD;
        local_opEF <= opEF;
    end
end

logic [`ADDER_IN_SIZE_18b-1:0]   op0123;
logic [`ADDER_IN_SIZE_18b-1:0]   op4567;
logic [`ADDER_IN_SIZE_18b-1:0]   op89AB;
logic [`ADDER_IN_SIZE_18b-1:0]   opCDEF;

unsigned_adder #(.WIDTH(18)) adder_inst00 (
    .dataa(local_op01),
    .datab(local_op23),
    .cin('0),
    .result(op0123)
);

unsigned_adder #(.WIDTH(18)) adder_inst01 (
    .dataa(local_op45),
    .datab(local_op67),
    .cin('0),
    .result(op4567)
);

unsigned_adder #(.WIDTH(18)) adder_inst02 (
    .dataa(local_op89),
    .datab(local_opAB),
    .cin('0),
    .result(op89AB)
```

```
);

unsigned_adder #(.WIDTH(18)) adder_inst03 (
    .dataa(local_opCD),
    .datab(local_opEF),
    .cin('0),
    .result(opCDEF)
);

logic [`ADDER_IN_SIZE_18b-1:0]   local_op0123;
logic [`ADDER_IN_SIZE_18b-1:0]   local_op4567;
logic [`ADDER_IN_SIZE_18b-1:0]   local_op89AB;
logic [`ADDER_IN_SIZE_18b-1:0]   local_opCDEF;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        local_op0123  <= '0;
        local_op4567  <= '0;
        local_op89AB  <= '0;
        local_opCDEF  <= '0;
    end else begin
        local_op0123  <= op0123;
        local_op4567  <= op4567;
        local_op89AB  <= op89AB;
        local_opCDEF  <= opCDEF;
    end
end

logic [`ADDER_IN_SIZE_18b-1:0]   op01234567;
logic [`ADDER_IN_SIZE_18b-1:0]   op89ABCDEF;

unsigned_adder #(.WIDTH(18)) adder_inst000 (
    .dataa(local_op0123),
    .datab(local_op4567),
    .cin('0),
    .result(op01234567)
);

unsigned_adder #(.WIDTH(18)) adder_inst001 (
    .dataa(local_op89AB),
    .datab(local_opCDEF),
    .cin('0),
    .result(op89ABCDEF)
);

logic [`ADDER_IN_SIZE_18b-1:0] local_op01234567;
logic [`ADDER_IN_SIZE_18b-1:0] local_op89ABCDEF;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        local_op01234567 <= '0;
```

```
            local_op89ABCDEF <= '0;
        end else begin
            local_op01234567 <= op01234567;
            local_op89ABCDEF <= op89ABCDEF;
        end
    end

    logic [`ADDER_IN_SIZE_18b-1:0]   final_sum;


    // Final addition
    unsigned_adder #(.WIDTH(18)) adder_final0000 (
        .dataa(local_op01234567),
        .datab(local_op89ABCDEF),
        .cin('0),
        .result(final_sum)
    );

    // Registered final result
    logic [`ADDER_IN_SIZE_18b-1:0] final_sum_reg;

    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            final_sum_reg <= '0;
        else
            final_sum_reg <= final_sum;
    end

    assign result = final_sum_reg;

endmodule
```

comb_mult_4b.sv

```
`include "lr_acc.svh"


module comb_mult_4b (
    input  [`MULT_INPUT_SIZE_4b-1:0] opa,
    input  [`MULT_INPUT_SIZE_4b-1:0] opb,
    output [`MULT_OUTPUT_SIZE_8b-1:0] result
);
    wire [`MULT_OUTPUT_SIZE_8b-1:0] p0 = opb[0] ? {4'b0, opa      } : 8'b0;
    wire [`MULT_OUTPUT_SIZE_8b-1:0] p1 = opb[1] ? {3'b0, opa, 1'b0} : 8'b0;
    wire [`MULT_OUTPUT_SIZE_8b-1:0] p2 = opb[2] ? {2'b0, opa, 2'b0} : 8'b0;
    wire [`MULT_OUTPUT_SIZE_8b-1:0] p3 = opb[3] ? {1'b0, opa, 3'b0} : 8'b0;

    assign result = p0 + p1 + p2 + p3;

endmodule
```

lr_acc_8b.sv

```
`include "lr_acc.svh"

module lr_acc_8b (
    input logic                                 clk,
    input logic                                 reset,
    input logic                                 start,

    input logic [`SPLIT_DATA_LEN-1:0]           x,
    input logic [`SPLIT_DATA_LEN-1:0]           y,

    output logic [`MULT_DATA_IN_LEN-1:0]        sum_x,
    output logic [`MULT_DATA_IN_LEN-1:0]        sum_y,
    output logic [`MULT_DATA_IN_LEN-1:0]        sum_x_squared,
    output logic [`MULT_DATA_IN_LEN-1:0]        sum_xy,

    output logic                                done
);

    // ----------- Pipeline Stage 1: Register Inputs -------------
    logic [`SPLIT_DATA_LEN-1:0] local_x, local_y;
    logic                       local_start;

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            local_x     <= '0;
            local_y     <= '0;
            local_start <= 0;
        end else if (start) begin
            local_x     <= x;
            local_y     <= y;
            local_start <= 1;
        end else begin
            local_start <= 0;
        end
    end

    // ---------- Pipeline Stage 2: Multiplier -------------------
    logic [`MULT_OUTPUT_SIZE_8b-1:0] local_x_squared, local_x_times_y;
    logic [`SPLIT_DATA_LEN-1:0] inp1, inp2;

    multiplier #(.WIDTH(4)) mult_x_x (
        .clk(clk),
        .X(local_x),
        .Y(local_x),
        .I1(),
        .I2(),
        .P(local_x_squared)
    );
```

```
multiplier #(.WIDTH(4)) mult_x_y (
    .clk(clk),
    .X(local_x),
    .Y(local_y),
    .I1(inp1),
    .I2(inp2),
    .P(local_x_times_y)
);

// ---------- Delay Stage to Capture Multiplier Outputs ------
logic                           delay_reg;
logic                           local_local_start;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        delay_reg           <= 0;
        local_local_start   <= 0;
    end else begin
        delay_reg           <= local_start;
        local_local_start   <= delay_reg;
    end
end

// --------- Pipeline Register After Multipliers ------------
logic [`MULT_OUTPUT_SIZE_8b-1:0] x_squared, x_times_y;
logic [`SPLIT_DATA_LEN-1:0]      inp_x, inp_y;
logic                            start_next_stage;

logic [`SPLIT_DATA_LEN-1:0] x_pipe, y_pipe;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        inp_x           <= '0;
        inp_y           <= '0;
        x_squared       <= '0;
        x_times_y       <= '0;
        x_pipe          <= '0;
        y_pipe          <= '0;
        start_next_stage <= 0;
    end else if (local_local_start) begin
        x_pipe          <= local_x;
        y_pipe          <= local_y;
        x_squared       <= local_x_squared;
        x_times_y       <= local_x_times_y;
        inp_x           <= inp1;     // corrected: use proper pipelined x
        inp_y           <= inp2;     // corrected: use proper pipelined y
        start_next_stage <= 1;
    end else begin
        start_next_stage <= 0;
    end
end
```

```
    // ---------- Pipeline Stage 3: Accumulator ------------------
    logic [`MULT_DATA_IN_LEN-1:0] local_sum_x, local_sum_y;
    logic [`MULT_DATA_IN_LEN-1:0] local_sum_xx, local_sum_xy;
    logic                         sum_done;

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            local_sum_x  <= '0;
            local_sum_y  <= '0;
            local_sum_xx <= '0;
            local_sum_xy <= '0;
            sum_done     <= 0;
        end else if (start_next_stage) begin
            local_sum_x  <= local_sum_x + inp_x;
            local_sum_y  <= local_sum_y + inp_y;
            local_sum_xx <= local_sum_xx + x_squared;
            local_sum_xy <= local_sum_xy + x_times_y;
            sum_done     <= 1;
        end else begin
            sum_done <= 0;
        end
    end

    // ------------------- Output Assignment ---------------------
    assign sum_x         = local_sum_x;
    assign sum_y         = local_sum_y;
    assign sum_x_squared = local_sum_xx;
    assign sum_xy        = local_sum_xy;
    assign done          = sum_done;

endmodule
```

lr_acc_512_8b.sv

```
    `include "lr_acc.svh"


module lr_acc_512_8b (
    //Port Structure
    input logic                             clk,
    input logic                             reset,

    input logic [31:0]                      writedata,
    input logic                             write,
    input logic                             read,
    input                                   chipselect,
    input logic [9:0]                       address,

    output logic [31:0]                     readdata
```

```
);

logic master_done;

memory [`DATA_SET_SIZE-1:0] mem_bank;
pc_tracker [`PIPELINE_SETS-1:0] pc_status_bank;
logic go;
logic [`ADDRESS_BITS_SIZE-1:0] pc;
logic done;
logic done_trigger;
logic [`MULT_DATA_IN_LEN-1:0] local_s_1;

logic [`MULT_DATA_IN_LEN-1:0] global_s_1, global_s_2, global_s_3, global_s_4,
↪  global_s_5;

logic [`OUTPUT_DATA_LEN-1:0] s1s4_minus_s2s2;
logic [`OUTPUT_DATA_LEN-1:0] s3s4_minus_s2s5;
logic [`OUTPUT_DATA_LEN-1:0] s1s5_minus_s2s3;

always_ff @(posedge clk)
begin      //Get data from SW
  if (reset)
  begin
    go <= 1'b0;
    master_done <= '0;
    done <= '0;
    done_trigger <= '0;
    local_s_1 <= '0;
    // global_s_1 <= '0;
    // global_s_2 <= '0;
    // global_s_3 <= '0;
    // global_s_4 <= '0;
    // global_s_5 <= '0;
    // s1s4_minus_s2s2 <= '0;
    // s3s4_minus_s2s5 <= '0;
    // s1s5_minus_s2s3 <= '0;
    for (int i = 0; i < `DATA_SET_SIZE; i++)
    begin
      mem_bank[i].x <= '0;
      mem_bank[i].y <= '0;
    end
  end
  else if (chipselect && write)
  begin      //Read data only when past operation is complete
    case (address[`ADDRESS_BITS_SIZE])
      1'h0:
              begin
                if (!go)
                begin
                  mem_bank[address[8:0]].x <= writedata[3:0];
                  mem_bank[address[8:0]].y <= writedata[7:4];
```

```
                end
              end
    1'h1:
              begin
                if (address[8:0] == 0)
                begin
                  go <= 1'b0;
                  master_done <= '0;
                  done <= '0;
                  done_trigger <= '0;
                  local_s_1 <= '0;
    // global_s_1 <= '0;
    // global_s_2 <= '0;
    // global_s_3 <= '0;
    // global_s_4 <= '0;
    // global_s_5 <= '0;
    // s1s4_minus_s2s2 <= '0;
    // s3s4_minus_s2s5 <= '0;
    // s1s5_minus_s2s3 <= '0;
                  for (int i = 0; i < `DATA_SET_SIZE; i++)
                  begin
                    mem_bank[i].x <= '0;
                    mem_bank[i].y <= '0;
                  end
                end
                else
                begin
                  go <= 1'b1;
                  local_s_1 <= address[8:0];
                end
              end
  endcase
end
else if (chipselect && read)
begin      //Read data only when past operation is complete
  case (address)
    10'h0:
    begin
      readdata <= master_done;
    end
    10'h1:
    begin
      readdata <= s1s4_minus_s2s2;
    end
    10'h2:
    begin
      readdata <= s3s4_minus_s2s5;
    end
    10'h3:
    begin
      readdata <= s1s5_minus_s2s3;
```

```
      end
      10'h4:
      begin
        readdata <= global_s_1;
      end
      10'h5:
      begin
        readdata <= global_s_2;
      end
      10'h6:
      begin
        readdata <= global_s_3;
      end
      10'h7:
      begin
        readdata <= global_s_4;
      end
      10'h8:
      begin
        readdata <= global_s_5;
      end
    endcase
  end

  if (done)
  begin
    done_trigger <= '1;
    done <= '0;
  end
  else if (done_trigger)
  begin
    master_done <= '1;
    done_trigger <= '0;
  end
  else if (pc == 496)
  begin
    go <= '0;
    done <= '1;
  end
end

logic [`PIPELINE_SETS-1:0] start ;
logic [`SPLIT_DATA_LEN-1:0] x [`PIPELINE_SETS-1:0];
logic [`SPLIT_DATA_LEN-1:0] y [`PIPELINE_SETS-1:0];
logic [`MULT_DATA_IN_LEN-1:0] sum_x [`PIPELINE_SETS-1:0];
logic [`MULT_DATA_IN_LEN-1:0] sum_y [`PIPELINE_SETS-1:0];
logic [`MULT_DATA_IN_LEN-1:0] sum_x_squared [`PIPELINE_SETS-1:0];
logic [`MULT_DATA_IN_LEN-1:0] sum_xy [`PIPELINE_SETS-1:0];
logic [`PIPELINE_SETS-1:0] local_done ;
```

```
logic [`PC_BITS_SIZE-1:0] local_pc [`PIPELINE_SETS-1:0];
logic [`PC_BITS_SIZE-1:0] dut_num;

logic master_start;
logic [`ADDRESS_BITS_SIZE-1:0] addr;

genvar i;
generate
  for (i = 0; i < `PIPELINE_SETS; i = i + 1)
  begin : lr_acc_gen
    lr_acc_8b lr_acc_8b_0 (
                .clk(clk),
                .reset(reset),
                .start(start[i]),
                .x(x[i]),
                .y(y[i]),
                .sum_x(sum_x[i]),
                .sum_y(sum_y[i]),
                .sum_x_squared(sum_x_squared[i]),
                .sum_xy(sum_xy[i]),
                .done(local_done[i])
              );
  end
endgenerate

always_ff @( posedge clk )
begin
  if (reset)
  begin
    pc <= '0;
    master_start <= '0;
    for (int i = 0; i < `PIPELINE_SETS; i++)
    begin
      start[i] <= '0;
      x[i] <= '0;
      y[i] <= '0;
    end
    addr <= '0;
    // master_done <= '0;
  // end
  // else if (!master_start && go)
  // begin
  //   for (int i = 0; i < `PIPELINE_SETS; i++)
  //   begin
  //     start[i] <= '0;
  //   end
  //   master_start <= '1;
  // end
  end else if (go)
  begin
    for (int i = 0; i < `PIPELINE_SETS; i++)
```

```
    begin
      addr <= pc + i;
      x[i] <= mem_bank[pc + i].x;
      y[i] <= mem_bank[pc + i].y;
      start[i] <= '1;
      pc_status_bank[i].pc <= pc + i;
      pc_status_bank[i].dut_num <= pc + i;
      pc_status_bank[i].done <= '0;
    end
    if(pc < 512)
    begin
      pc <= pc + 16;
    end
    else
    begin
      pc <= '0;
    end
    master_start <= '0;
  end
  else if (!go)
  begin
    for (int i = 0; i < `PIPELINE_SETS; i++)
    begin
      start[i] <= '0;
    end
  end
end

logic global_start_adders;


always_ff @( posedge clk )
begin
  if(reset)
  begin
    for (int i = 0; i < `PIPELINE_SETS; i++)
    begin
      pc_status_bank[i].pc <= '0;
      pc_status_bank[i].dut_num <= '0;
      pc_status_bank[i].done <= '0;
      pc_status_bank[i].local_s_2 <= '0;
      pc_status_bank[i].local_s_3 <= '0;
      pc_status_bank[i].local_s_4 <= '0;
      pc_status_bank[i].local_s_5 <= '0;
    end
    global_start_adders <= '0;
  end
  else if (master_done)
  begin
    for (int i = 0; i < `PIPELINE_SETS; i++)
    begin
```

```
      pc_status_bank[i].done <= '1;
      pc_status_bank[i].local_s_2 <= sum_x[i];
      pc_status_bank[i].local_s_3 <= sum_y[i];
      pc_status_bank[i].local_s_4 <= sum_x_squared[i];
      pc_status_bank[i].local_s_5 <= sum_xy[i];
    end
    global_s_1 <= local_s_1;
    global_start_adders <= '1;
  end
end


comb_adder_18b sum_s2_calc (
                .clk(clk),
                .reset(reset),
                 .adder_start(global_start_adders),
                 .op0  (pc_status_bank[ 0].local_s_2),
                 .op1  (pc_status_bank[ 1].local_s_2),
                 .op2  (pc_status_bank[ 2].local_s_2),
                 .op3  (pc_status_bank[ 3].local_s_2),

                 .op4  (pc_status_bank[ 4].local_s_2),
                 .op5  (pc_status_bank[ 5].local_s_2),
                 .op6  (pc_status_bank[ 6].local_s_2),
                 .op7  (pc_status_bank[ 7].local_s_2),

                 .op8  (pc_status_bank[ 8].local_s_2),
                 .op9  (pc_status_bank[ 9].local_s_2),
                 .opA  (pc_status_bank[10].local_s_2),
                 .opB  (pc_status_bank[11].local_s_2),

                 .opC  (pc_status_bank[12].local_s_2),
                 .opD  (pc_status_bank[13].local_s_2),
                 .opE  (pc_status_bank[14].local_s_2),
                 .opF  (pc_status_bank[15].local_s_2),
                 .result(global_s_2)
               );

comb_adder_18b sum_s3_calc (
                .clk(clk),
                .reset(reset),
                 .adder_start(global_start_adders),
                 .op0  (pc_status_bank[ 0].local_s_3),
                 .op1  (pc_status_bank[ 1].local_s_3),
                 .op2  (pc_status_bank[ 2].local_s_3),
                 .op3  (pc_status_bank[ 3].local_s_3),

                 .op4  (pc_status_bank[ 4].local_s_3),
                 .op5  (pc_status_bank[ 5].local_s_3),
                 .op6  (pc_status_bank[ 6].local_s_3),
                 .op7  (pc_status_bank[ 7].local_s_3),
```

```
                .op8  (pc_status_bank[ 8].local_s_3),
                .op9  (pc_status_bank[ 9].local_s_3),
                .opA  (pc_status_bank[10].local_s_3),
                .opB  (pc_status_bank[11].local_s_3),

                .opC  (pc_status_bank[12].local_s_3),
                .opD  (pc_status_bank[13].local_s_3),
                .opE  (pc_status_bank[14].local_s_3),
                .opF  (pc_status_bank[15].local_s_3),
                .result(global_s_3)
              );
comb_adder_18b sum_s4_calc (
                .clk(clk),
                .reset(reset),
                .adder_start(global_start_adders),
                .op0  (pc_status_bank[ 0].local_s_4),
                .op1  (pc_status_bank[ 1].local_s_4),
                .op2  (pc_status_bank[ 2].local_s_4),
                .op3  (pc_status_bank[ 3].local_s_4),

                .op4  (pc_status_bank[ 4].local_s_4),
                .op5  (pc_status_bank[ 5].local_s_4),
                .op6  (pc_status_bank[ 6].local_s_4),
                .op7  (pc_status_bank[ 7].local_s_4),

                .op8  (pc_status_bank[ 8].local_s_4),
                .op9  (pc_status_bank[ 9].local_s_4),
                .opA  (pc_status_bank[10].local_s_4),
                .opB  (pc_status_bank[11].local_s_4),

                .opC  (pc_status_bank[12].local_s_4),
                .opD  (pc_status_bank[13].local_s_4),
                .opE  (pc_status_bank[14].local_s_4),
                .opF  (pc_status_bank[15].local_s_4),
                .result(global_s_4)
              );

comb_adder_18b sum_s5_calc (
                .clk(clk),
                .reset(reset),
                .adder_start(global_start_adders),
                .op0  (pc_status_bank[ 0].local_s_5),
                .op1  (pc_status_bank[ 1].local_s_5),
                .op2  (pc_status_bank[ 2].local_s_5),
                .op3  (pc_status_bank[ 3].local_s_5),

                .op4  (pc_status_bank[ 4].local_s_5),
                .op5  (pc_status_bank[ 5].local_s_5),
                .op6  (pc_status_bank[ 6].local_s_5),
                .op7  (pc_status_bank[ 7].local_s_5),
```

```
                .op8  (pc_status_bank[ 8].local_s_5),
                .op9  (pc_status_bank[ 9].local_s_5),
                .opA  (pc_status_bank[10].local_s_5),
                .opB  (pc_status_bank[11].local_s_5),

                .opC  (pc_status_bank[12].local_s_5),
                .opD  (pc_status_bank[13].local_s_5),
                .opE  (pc_status_bank[14].local_s_5),
                .opF  (pc_status_bank[15].local_s_5),
                .result(global_s_5)
              );

logic [`MULT_DATA_IN_LEN-1:0] local_global_s_1, local_global_s_2,
↪  local_global_s_3, local_global_s_4, local_global_s_5;


always_ff @( posedge clk )
begin
  if(reset)
  begin
    local_global_s_1 <= '0;
    local_global_s_2 <= '0;
    local_global_s_3 <= '0;
    local_global_s_4 <= '0;
    local_global_s_5 <= '0;
  end
  else
  begin
    local_global_s_1 <= global_s_1;
    local_global_s_2 <= global_s_2;
    local_global_s_3 <= global_s_3;
    local_global_s_4 <= global_s_4;
    local_global_s_5 <= global_s_5;
  end
end

//assign s1_times_s4 = global_s_1 * global_s_4;
//assign s2_squared = global_s_2 * global_s_2;
//assign s3_times_s4 = global_s_3 * global_s_4;
//assign s2_times_s5 = global_s_2 * global_s_5;
//assign s1_times_s5 = global_s_1 * global_s_5;
//assign s2_times_s3 = global_s_2 * global_s_3;

logic [`DSP_MULT_OUT_SIZE-1:0] local_s1_times_s4;
logic [`DSP_MULT_OUT_SIZE-1:0] local_s2_squared;
logic [`DSP_MULT_OUT_SIZE-1:0] local_s3_times_s4;
logic [`DSP_MULT_OUT_SIZE-1:0] local_s2_times_s5;
logic [`DSP_MULT_OUT_SIZE-1:0] local_s1_times_s5;
logic [`DSP_MULT_OUT_SIZE-1:0] local_s2_times_s3;
```

```
multiplier m1 (
             .clk(clk),
             .X   (local_global_s_1),
             .Y   (local_global_s_4),
             .I1(),
             .I2(),
             .P   (local_s1_times_s4)
           );

multiplier m2 (
             .clk(clk),
             .X   (local_global_s_2),
             .Y   (local_global_s_2),
             .I1(),
             .I2(),
             .P   (local_s2_squared)
           );

multiplier m3 (
             .clk(clk),
             .X   (local_global_s_3),
             .Y   (local_global_s_4),
             .I1(),
             .I2(),
             .P   (local_s3_times_s4)
           );

multiplier m4 (
             .clk(clk),
             .X   (local_global_s_2),
             .Y   (local_global_s_5),
             .I1(),
             .I2(),
             .P   (local_s2_times_s5)
           );

multiplier m5 (
             .clk(clk),
             .X   (local_global_s_1),
             .Y   (local_global_s_5),
             .I1(),
             .I2(),
             .P   (local_s1_times_s5)
           );

multiplier m6 (
             .clk(clk),
             .X   (local_global_s_2),
             .Y   (local_global_s_3),
             .I1(),
             .I2(),
```

```
              .P   (local_s2_times_s3)
            );

  logic [`DSP_MULT_OUT_SIZE-1:0] s1_times_s4;
  logic [`DSP_MULT_OUT_SIZE-1:0] s2_squared;
  logic [`DSP_MULT_OUT_SIZE-1:0] s3_times_s4;
  logic [`DSP_MULT_OUT_SIZE-1:0] s2_times_s5;
  logic [`DSP_MULT_OUT_SIZE-1:0] s1_times_s5;
  logic [`DSP_MULT_OUT_SIZE-1:0] s2_times_s3;

  always_ff @( posedge clk )
  begin
    if(reset)
    begin
      s1_times_s4 <= '0;
      s2_squared <= '0;
      s3_times_s4 <= '0;
      s2_times_s5 <= '0;
      s1_times_s5 <= '0;
      s2_times_s3 <= '0;
    end
    else
    begin
      s1_times_s4 <= local_s1_times_s4;
      s2_squared <= local_s2_squared;
      s3_times_s4 <= local_s3_times_s4;
      s2_times_s5 <= local_s2_times_s5;
      s1_times_s5 <= local_s1_times_s5;
      s2_times_s3 <= local_s2_times_s3;
    end
  end

  assign s1s4_minus_s2s2 = s1_times_s4 - s2_squared;
  assign s3s4_minus_s2s5 = s3_times_s4 - s2_times_s5;
  assign s1s5_minus_s2s3 = s1_times_s5 - s2_times_s3;

endmodule
```

multiplier.sv

```
module multiplier
#(parameter WIDTH=18)
(
        input clk,
        input [WIDTH-1:0] X,
        input [WIDTH-1:0] Y,
        output reg [WIDTH-1:0] I1,
        output reg [WIDTH-1:0] I2,
        output reg [2*WIDTH-1:0] P
```

```
);

        // Declare input and output registers
        reg [WIDTH-1:0] dataa_reg;
        reg [WIDTH-1:0] datab_reg;
        wire [2*WIDTH-1:0] mult_out;
        wire [2*WIDTH-1:0] in1;
        wire [2*WIDTH-1:0] in2;

        // Store the result of the multiply
        assign mult_out = dataa_reg * datab_reg;
        assign in1 = dataa_reg;
        assign in2 = datab_reg;


        // Update data
        always @ (posedge clk)
        begin
                dataa_reg <= X;
                datab_reg <= Y;
                P <= mult_out;
                I1 <= in1;
                I2 <= in2;
        end

endmodule
```

unsigned_adder.sv

```
// Quartus Prime Verilog Template
// Unsigned Adder

module unsigned_adder
#(parameter WIDTH=16)
(
        input [WIDTH-1:0] dataa,
        input [WIDTH-1:0] datab,
        input cin,
        output [WIDTH:0] result
);

        assign result = dataa + datab + cin;

endmodule
```

lr_acc.svh

```
`ifndef __LR_ACC_SVH__
`define __LR_ACC_SVH__
```

```
`timescale 1ns/100ps


`ifndef CLOCK_PERIOD
`define CLOCK_PERIOD 10
`endif

`define SPLIT_DATA_LEN 4
`define INPUT_DATA_LEN 8
`define MULT_DATA_IN_LEN 18
`define OUTPUT_DATA_LEN 32


`define MEM_DATA_WIDTH 8
`define MEM_ADDR_WIDTH 9

`define MULT_4B_WIDTH 4

`define DATA_SET_SIZE 512

`define ADDRESS_BITS_SIZE $clog2(`DATA_SET_SIZE)

`define PIPELINE_SETS 16
`define PC_BITS_SIZE $clog2(`PIPELINE_SETS)

`define ADDER_IN_SIZE_18b 18
`define ADDER_OUT_SIZE_18b 18


`define ADDER_INPUT_SIZE_4b 4
`define ADDER_OUTPUT_SIZE_5b 5

`define MULT_INPUT_SIZE_4b 4
`define MULT_OUTPUT_SIZE_8b 8

`define DSP_MULT_IN_SIZE 18
`define DSP_MULT_OUT_SIZE 36

typedef struct packed {
    logic [`SPLIT_DATA_LEN-1:0] x;
    logic [`SPLIT_DATA_LEN-1:0] y;
} memory;

typedef struct packed {
    logic [`ADDRESS_BITS_SIZE-1:0] pc;
    logic [`PC_BITS_SIZE-1:0] dut_num;
    logic done;
    logic [`MULT_DATA_IN_LEN-1:0] local_s_2;
    logic [`MULT_DATA_IN_LEN-1:0] local_s_3;
    logic [`MULT_DATA_IN_LEN-1:0] local_s_4;
```

```
    logic [`MULT_DATA_IN_LEN-1:0] local_s_5;
} pc_tracker;

`endif
```

Software Code

lr_acc.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "lr_acc.h"

#define DRIVER_NAME "lr_acc"

/* Device registers */
#define BG_RED(x) (x)
#define BG_GREEN(x) ((x) + 1)
#define BG_BLUE(x) ((x) + 2)

#define VGA_BACKGROUND_OFFSET 0

/*
 * Information about our device
 */
struct lr_acc_dev
{
        struct resource res;        /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
        lr_acc_arg_t data;
        lr_acc_read_data_t read_data;
} dev;

/* Write background color */
static void write_data(lr_acc_arg_t *data)
{
        if (data->go)
```

```
        {
                iowrite32((u32)1, dev.virtbase + 4 * ((1 << 9) + data->address));
        }
        else
        {
                iowrite32((u32)data->data.data, dev.virtbase + 4 * data->address);
        }
}

static void read_data(lr_acc_read_data_t *data)
{
        int a = ioread32(dev.virtbase + 0);
        int b = ioread32(dev.virtbase + 4);
        int c = ioread32(dev.virtbase + 8);
        int d = ioread32(dev.virtbase + 12);
        int e = ioread32(dev.virtbase + 16);
        int f = ioread32(dev.virtbase + 20);
        int g = ioread32(dev.virtbase + 24);
        int h = ioread32(dev.virtbase + 28);
        int i = ioread32(dev.virtbase + 32);

        data->master_done = a;
        data->d = b;
        data->n0 = c;
        data->n1 = d;
        data->s1 = e;
        data->s2 = f;
        data->s3 = g;
        data->s4 = h;
        data->s5 = i;
}


/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long lr_acc_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        lr_acc_arg_t vla;
        lr_acc_read_data_t obj;

        switch (cmd)
        {
        case LR_ACC_WRITE_DATA:
                if (copy_from_user(&vla, (lr_acc_arg_t *)arg,
                                                sizeof(lr_acc_arg_t)))
                        return -EACCES;
                write_data(&vla);
                break;
```

```
        case LR_ACC_READ_DATA:
                read_data(&obj);

                if (copy_to_user((lr_acc_read_data_t *)arg, &obj,
                                                sizeof(lr_acc_read_data_t)))
                        return -EACCES;

                break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations lr_acc_fops = {
        .owner = THIS_MODULE,
        .unlocked_ioctl = lr_acc_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice lr_acc_misc_device = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = DRIVER_NAME,
        .fops = &lr_acc_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init lr_acc_probe(struct platform_device *pdev)
{
        // lr_acc_color_t beige = { 0xf9, 0xe4, 0xb7 };
        // lr_acc_color_t beige = { 0xff, 0xff, 0xff };

        int ret;

        /* Register ourselves as a misc device: creates /dev/lr_acc */
        ret = misc_register(&lr_acc_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret)
        {
                ret = -ENOENT;
                goto out_deregister;
        }
```

```
        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                                                    DRIVER_NAME) == NULL)
        {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL)
        {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        /* Set an initial color */
        // write_background(&beige);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&lr_acc_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int lr_acc_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&lr_acc_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id lr_acc_of_match[] = {
        {.compatible = "csee4840,lr_acc-1.0"},
        {},
};
MODULE_DEVICE_TABLE(of, lr_acc_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver lr_acc_driver = {
        .driver = {
                .name = DRIVER_NAME,
```

```
                .owner = THIS_MODULE,
                .of_match_table = of_match_ptr(lr_acc_of_match),
        },
        .remove = __exit_p(lr_acc_remove),
};

/* Called when the module is loaded: set things up */
static int __init lr_acc_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&lr_acc_driver, lr_acc_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit lr_acc_exit(void)
{
        platform_driver_unregister(&lr_acc_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(lr_acc_init);
module_exit(lr_acc_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Venkat Suprabath Bitra, Columbia University");
MODULE_DESCRIPTION("LR Accumulator Driver");
```

lr_acc.h

```
#ifndef _lr_acc_H
#define _lr_acc_H

#include <linux/ioctl.h>

typedef struct {
  char data;
} lr_acc_data_t;

typedef struct {
  lr_acc_data_t data;
  int address;
  char go;
} lr_acc_arg_t;

typedef struct {
  int master_done;
  int d, n0, n1, s1, s2, s3, s4, s5;
} lr_acc_read_data_t;
```

```
#define LR_ACC_MAGIC 'q'

/* ioctls and their arguments */
#define LR_ACC_WRITE_DATA _IOW(LR_ACC_MAGIC, 1, lr_acc_arg_t)
#define LR_ACC_READ_DATA  _IOR(LR_ACC_MAGIC, 2, lr_acc_read_data_t)

#endif
```

main.c

```
#include <stdio.h>
#include <time.h>
#include "lr_acc.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

int lr_acc_fd;

static const unsigned int raw_data[][2] = {
  {1, 15}, {1, 14}, {1, 15}, {1, 15}, {1, 15}, {1, 14}, {1, 15},
  {1, 15}, {1, 14}, {1, 15}, {1, 14}, {1, 15}, {1, 15}, {1, 13},
  {1, 15}, {1, 14}, {1, 15}, {1, 15}, {1, 15}, {1, 14}, {1, 14},
  {1, 15}, {1, 15}, {1, 15}, {1, 15}, {1, 15}, {1, 14}, {1, 15},
  {1, 15}, {1, 15}, {1, 15}, {1, 14}, {1, 15}, {1, 15}, {1, 15},
  {1, 14}, {1, 14}, {1, 15}, {1, 14}, {1, 15}, {1, 15}, {1, 15},
  {1, 14}, {1, 14}, {1, 14}, {1, 14}, {1, 14}, {1, 14}, {2, 12},
  {2, 12}, {2, 12}, {2, 13}, {2, 12}, {2, 12}, {2, 13}, {2, 12},
  {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 13}, {2, 12},
  {2, 13}, {2, 13}, {2, 13}, {2, 13}, {2, 13}, {2, 13}, {2, 13},
  {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 13}, {2, 13},
  {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 13}, {2, 12},
  {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 12},
  {2, 12}, {2, 12}, {2, 12}, {2, 12}, {2, 12}, {3, 12}, {3, 13},
  {3, 13}, {3, 12}, {3, 12}, {3, 12}, {3, 12}, {3, 12}, {3, 12},
  {3, 12}, {3, 12}, {3, 12}, {3, 11}, {3, 13}, {3, 12}, {3, 11},
```

```
  {3, 12}, {3, 12}, {3, 12}, {3, 11}, {3, 12}, {3, 12}, {3, 12}, {3, 13}, {3, 12},
  ↪  {3, 12}, {3, 12}, {3, 13}, {3, 13}, {3, 12}, {3, 12}, {3, 12}, {3, 12}, {3,
  ↪  12}, {3, 12}, {3, 12}, {3, 12}, {3, 13}, {3, 12}, {3, 12}, {3, 12}, {3, 12},
  ↪  {3, 12}, {3, 13}, {3, 12}, {3, 12}, {3, 12}, {3, 12}, {4, 10}, {4, 10}, {4,
  ↪  10}, {4, 10}, {4, 11}, {4, 11}, {4, 10}, {4, 10}, {4, 10}, {4, 10}, {4, 10},
  ↪  {4, 10}, {4, 10}, {4, 10}, {4, 10}, {4, 11}, {4, 10}, {4, 11}, {4, 11}, {4,
  ↪  11}, {4, 10}, {4, 11}, {4, 11}, {4, 11}, {4, 11}, {4, 10}, {4, 10}, {4, 11},
  ↪  {4, 10}, {4, 11}, {4, 11}, {4, 11}, {4, 10}, {4, 10}, {4, 10}, {4, 10}, {4,
  ↪  10}, {4, 11}, {4, 10}, {4, 11}, {4, 10}, {4, 11}, {4, 11}, {4, 11}, {4, 11},
  ↪  {4, 11}, {4, 11}, {4, 11}, {5, 7}, {5, 7}, {5, 8}, {5, 9}, {5, 9}, {5, 8},
  ↪  {5, 9}, {5, 9}, {5, 9}, {5, 9}, {5, 8}, {5, 9}, {5, 8}, {5, 10}, {5, 8}, {5,
  ↪  10}, {5, 9}, {5, 9}, {5, 10}, {5, 10}, {5, 11}, {5, 10}, {5, 10}, {5, 10},
  ↪  {5, 10}, {5, 9}, {5, 9}, {5, 9}, {5, 9}, {5, 9}, {5, 10}, {5, 10}, {5, 9},
  ↪  {5, 9}, {5, 9}, {5, 9}, {5, 9}, {5, 10}, {5, 10}, {5, 9}, {5, 10}, {5, 10},
  ↪  {5, 9}, {5, 10}, {5, 10}, {5, 9}, {5, 9}, {5, 9}, {6, 6}, {6, 6}, {6, 6}, {6,
  ↪  7}, {6, 7}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6},
  ↪  {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 5}, {6, 5}, {6, 5}, {6,
  ↪  6}, {6, 5}, {6, 5}, {6, 5}, {6, 5}, {6, 6}, {6, 5}, {6, 5}, {6, 5}, {6, 5},
  ↪  {6, 5}, {6, 5}, {6, 5}, {6, 6}, {6, 8}, {6, 8}, {6, 8}, {6, 8}, {6, 8}, {6,
  ↪  8}, {6, 8}, {6, 8}, {6, 8}, {6, 8}, {6, 8}, {6, 8}, {7, 5}, {7, 5}, {7, 5},
  ↪  {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7,
  ↪  7}, {7, 6}, {7, 6}, {7, 7}, {7, 7}, {7, 6}, {7, 6}, {7, 6}, {7, 7}, {7, 7},
  ↪  {7, 6}, {7, 6}, {7, 5}, {7, 5}, {7, 5}, {7, 4}, {7, 5}, {7, 5}, {7, 4}, {7,
  ↪  4}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 4}, {7, 5}, {7, 5}, {7, 5},
  ↪  {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {7, 5}, {8, 2}, {8, 2}, {8,
  ↪  1}, {8, 2}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 2}, {8, 1},
  ↪  {8, 4}, {8, 4}, {8, 2}, {8, 0}, {8, 2}, {8, 2}, {8, 1}, {8, 2}, {8, 2}, {8,
  ↪  2}, {8, 2}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 2}, {8, 1},
  ↪  {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 2}, {8, 1}, {8, 1}, {8, 2}, {8,
  ↪  1}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 1}, {8, 2}, {8, 1}, {8, 1}, {9, 0}, {9, 0},
  ↪  {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9,
  ↪  0}, {9, 0}, {9, 0}, {9, 0}, {9, 1}, {9, 0}, {9, 0}, {9, 0}, {9, 1}, {9, 0},
  ↪  {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 2}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9,
  ↪  0}, {9, 0}, {9, 0}, {9, 1}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0},
  ↪  {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}, {9, 0}
};

#define NUM_ENTRIES (sizeof(raw_data) / sizeof(raw_data[0]))

int read_data_from_array(char **data) {
    *data = (char *)malloc(NUM_ENTRIES * sizeof(char));
    if (*data == NULL) {
        perror("Error allocating memory");
        return -1;
    }

    for (int i = 0; i < NUM_ENTRIES; i++) {
        unsigned int y = raw_data[i][0];
        unsigned int x = raw_data[i][1];
        (*data)[i] = (y << 4) | x;
    }
```

```
    return NUM_ENTRIES;
}

void set_lr_data(const lr_acc_arg_t *d)
{

  if (ioctl(lr_acc_fd, LR_ACC_WRITE_DATA, d))
  {
    fprintf(stderr, "ioctl(LR_ACC_SET_DATA) failed");
    return;
  }
}

void read_lr_data(lr_acc_read_data_t *d)
{
  if (ioctl(lr_acc_fd, LR_ACC_READ_DATA, d))
  {
    fprintf(stderr, "ioctl(LR_ACC_GET_DATA) failed");
    return;
  }
}

int main()
{
  lr_acc_arg_t vla;
  lr_acc_read_data_t obj;
  int i;
  static const char filename[] = "/dev/lr_acc";

  if ((lr_acc_fd = open(filename, O_RDWR)) == -1)
  {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
  }

  char *data = NULL;
  int n = read_data_from_array(&data);
  if (n < 0) {
      fprintf(stderr, "Error reading data\n");
      return -1;
  }

  fprintf(stderr, "Read %d data points\n", n);

  lr_acc_data_t *d = (lr_acc_data_t *)malloc(sizeof(lr_acc_data_t));

  clock_t start, end;
  double cpu_time_used;

  start = clock();
```

```
vla.go = 1;
vla.address = 0;

set_lr_data(&vla);

for (int i = 0; i < n; i++) {
    d->data = data[i];
    vla.data = *d;
    vla.address = i;
    vla.go = 0;

    set_lr_data(&vla);
}

end = clock();

cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
fprintf(stderr, "Time taken to send data: %f microseconds\n", cpu_time_used *
↪  1e6);

start = clock();
vla.go = 1;
vla.address = n;
set_lr_data(&vla);

while (1)
{
  read_lr_data(&obj);

  if (obj.master_done == 1)
    break;

  usleep(1);
}

end = clock();

cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
fprintf(stderr, "Time taken to process and read data: %f microseconds\n",
↪  cpu_time_used * 1e6);

start = clock();

double w0 = (double)obj.n0 / (double)obj.d;
double w1 = (double)obj.n1 / (double)obj.d;

end = clock();


cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
```

```
  fprintf(stderr, "Time taken to calculate weights: %f microseconds\n",
  ↪  cpu_time_used * 1e6);

  fprintf(stderr, "Device finished processing\n");
  fprintf(stderr, "Results:\n");
  fprintf(stderr, "d: %d\n", obj.d);
  fprintf(stderr, "n0: %d\n", obj.n0);
  fprintf(stderr, "n1: %d\n", obj.n1);
  fprintf(stderr, "s1: %d\n", obj.s1);
  fprintf(stderr, "s2: %d\n", obj.s2);
  fprintf(stderr, "s3: %d\n", obj.s3);
  fprintf(stderr, "s4: %d\n", obj.s4);
  fprintf(stderr, "s5: %d\n", obj.s5);
  fprintf(stderr, "Weights:\n");
  fprintf(stderr, "w0: %f\n", w0);
  fprintf(stderr, "w1: %f\n", w1);

  fprintf(stderr, "Freeing allocated memory\n");
  free(data);
  free(d);
  close(lr_acc_fd);
  fprintf(stderr, "Closed the device file\n");

  fprintf(stderr, "LR Accumulator Userspace program terminating\n");
  return 0;
}
```

parsing.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_LINES 100000

int n;
int x_vals[MAX_LINES];
int y_vals[MAX_LINES];

void read_data(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        perror("File opening failed");
        exit(EXIT_FAILURE);
    }

    if (fscanf(fp, "%d", &n) != 1) {
        fprintf(stderr, "Invalid file format\n");
        fclose(fp);
```

```
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < n; i++) {
        if (fscanf(fp, "%d %d", &y_vals[i], &x_vals[i]) != 2) {
            fprintf(stderr, "Invalid data at line %d\n", i + 2);
            fclose(fp);
            exit(EXIT_FAILURE);
        }
    }

    fclose(fp);
}

void run(double *a, double *b) {
    long long s2 = 0, s3 = 0, s4 = 0, s5 = 0;

    for (int i = 0; i < n; i++) {
        int x = x_vals[i];
        int y = y_vals[i];

        s2 += x;
        s3 += y;
        s4 += (long long)x * x;
        s5 += (long long)x * y;
    }

    long long n0 = s4 * s3 - s2 * s5;
    long long n1 = s5 * n - s2 * s3;
    long long d  = n * s4 - s2 * s2;

    *a = (double)n0 / d;
    *b = (double)n1 / d;
}

int main() {
    read_data("preprocessed_data.txt");

    double a = 0, b = 0;

    clock_t start_clock = clock();

    for (int i = 0; i < 1000; i++) {
        run(&a, &b);
    }

    clock_t end_clock = clock();

    double cpu_time = (double)(end_clock - start_clock) / CLOCKS_PER_SEC;
    double avg_usec = (cpu_time / 1000.0) * 1e6;
```

```
    printf("Elapsed time (CPU): %.6f us per run\n", avg_usec);
    // printf("a: %.6f, b: %.6f\n", a, b);

    return 0;
}
```

Fmax Timing Summary

```
+-----------------------------------------------------------------------------
; Slow 1100mV 85C Model Fmax Summary
+-------------+-----------------+-----------------------------------------------
; Fmax        ; Restricted Fmax ; Clock Name
+-------------+-----------------+-----------------------------------------------
; 163.75 MHz  ; 163.75 MHz      ; clock_50_1
; 1184.83 MHz ; 717.36 MHz      ; soc_system:soc_system0|soc_system_hps_0:hps_0|soc_
+-------------+-----------------+-----------------------------------------------
```

Figure 18: Slow 1100mV 85°C Model Fmax Summary

The table in Figure 18 provides the maximum clock frequencies (**Fmax**) under the slow timing corner (1100 mV, 85°C):

- **clock_50_1** has a maximum frequency of **163.75 MHz**, which is also the restricted Fmax, indicating no additional constraints or timing bottlenecks were applied to it.
- The **HPS system clock** path (long hierarchical name: `soc_system:soc_system0|...`) achieves a significantly higher unrestricted Fmax of **1184.83 MHz**. However, the restricted Fmax is lowered to **717.36 MHz**, suggesting design or interface constraints limit this clock's usable frequency to avoid setup timing violations or to match external timing requirements.

These Fmax values help determine the safe and efficient operating frequencies for different clock domains within the design.