COLUMBIA ENGINEERING
The Fu Foundation School of Engineering and Applied Science

AccelReg: Accelerator for Linear Regression

CSEE 4840: Embedded Systems

Pranav Asuri

pa2708@columbia.edu

Venkat Suprabath Bitra

vsb2127@columbia.edu

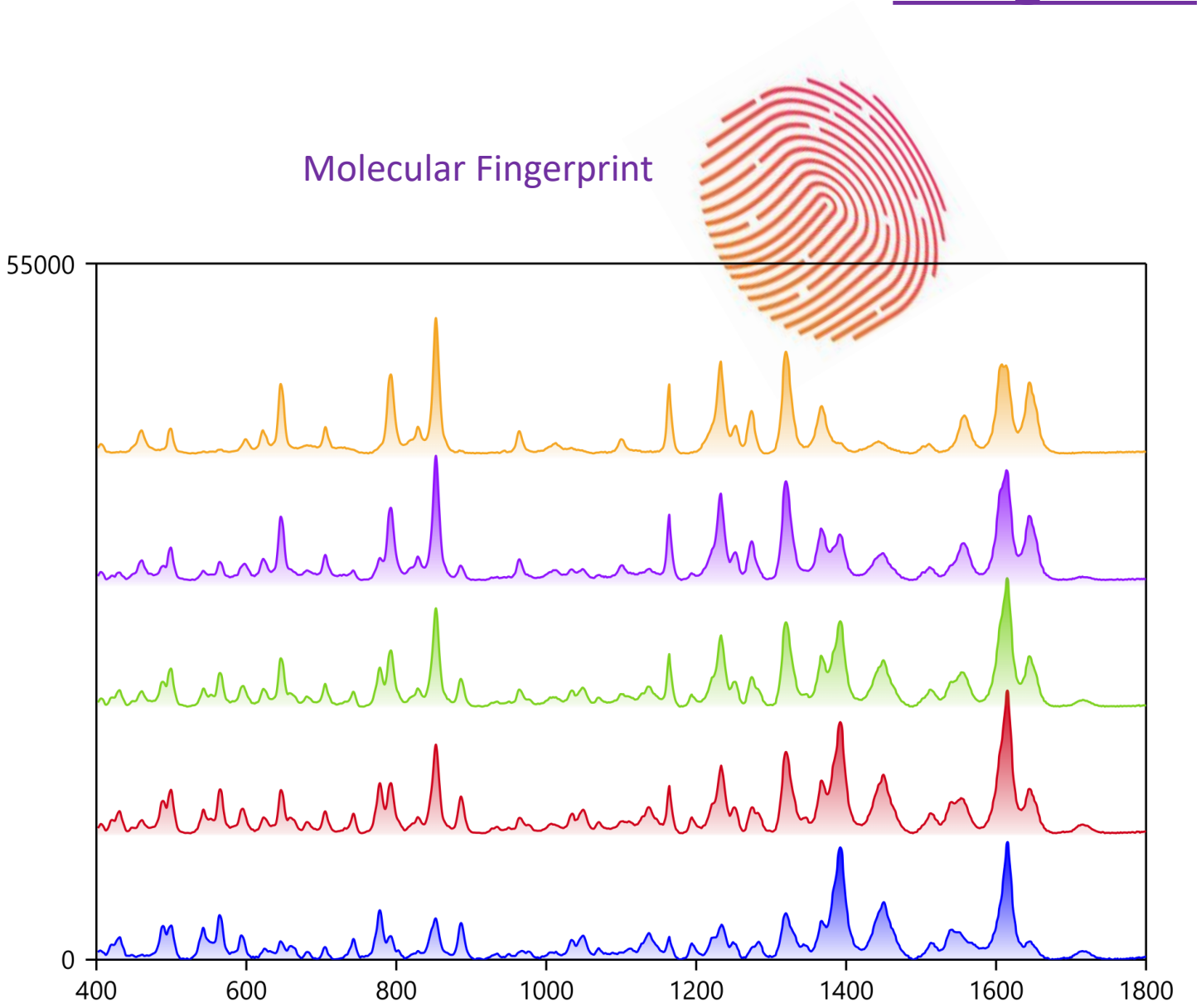
Varsha Keshava Prasad

vk2550@columbia.edu

Doreen Sisanalli

ds4371@columbia.edu

Background



- Unique spectral "fingerprints" for identifying chemical compounds and their compositions
- Concentration of components can be determined by analyzing the intensity ratios of their prominent Raman peaks
- Ratio of these characteristic peaks exhibits a near-linear relationship with the concentration

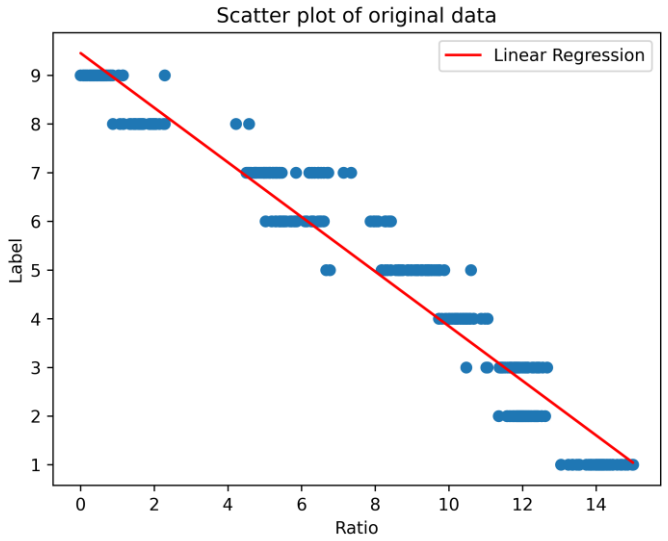
Motivation

- Instrument makers need on-board hardware for direct quantitative analysis, avoiding external PCs
- Portable spectrometers gather data fast, requires quick, real-time parameters for point-of-care applications
- Scientists prefer automated tools, typically avoiding to work directly with code for ease of use

Develop an FPGA linear regression module for portable Raman spectrometers, enabling rapid, on-the-fly, user-friendly chemical concentration analysis

4-Bit Integer Quantization for Linear Regression

- Quantized 4-bit integers reduce FPGA resource usage and power
- Training with representative dataset ensures robust quantized model performance
- Representative Dataset: reflects statistical properties and variations of the original full precision data.
- 4-bit quantization yields results closely matching the original full-precision model



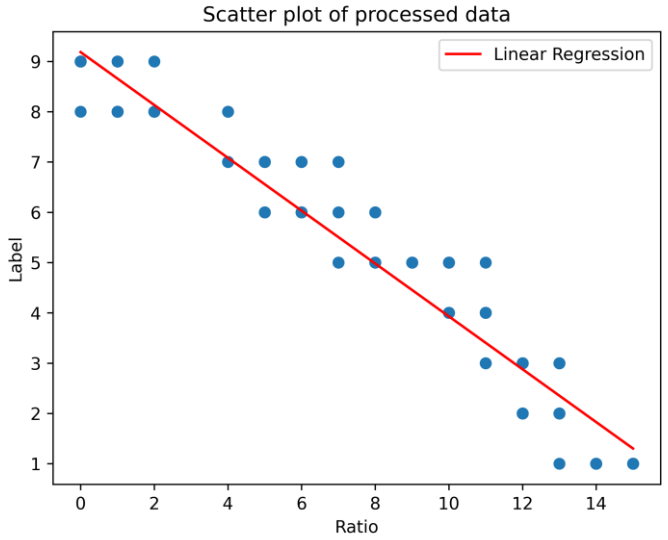
Fit Metrics

Slope : -0.560855

Intercept : 9.455583

R2 Score : 0.951513

MSE : 0.323244



Fit Metrics

Slope : -0.525380

Intercept : 9.183580

R2 Score : 0.948967

MSE : 0.340218

Simplification for Linear Regression

n observations (x_i, y_i) $\quad y = w_0 + w_1 x$

$$y_i = \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \qquad \mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

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

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

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

$$S_1 = n \qquad \text{(count of observations)}$$

$$S_2 = \sum_{i=1}^{n} x_i \qquad \text{(sum of } x_i\text{)}$$

$$S_3 = \sum_{i=1}^{n} y_i \qquad \text{(sum of } y_i\text{)}$$

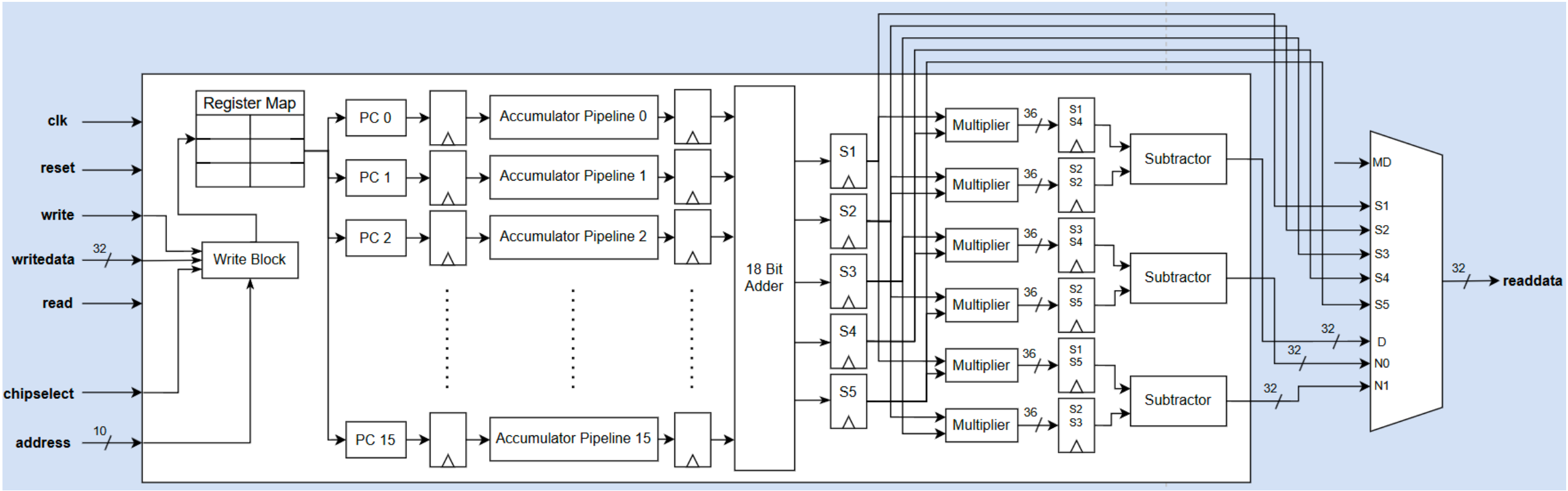
$$S_4 = \sum_{i=1}^{n} x_i^2 \qquad \text{(sum of } x_i^2\text{)}$$

$$S_5 = \sum_{i=1}^{n} x_i y_i \qquad \text{(sum of } x_i y_i\text{)}$$

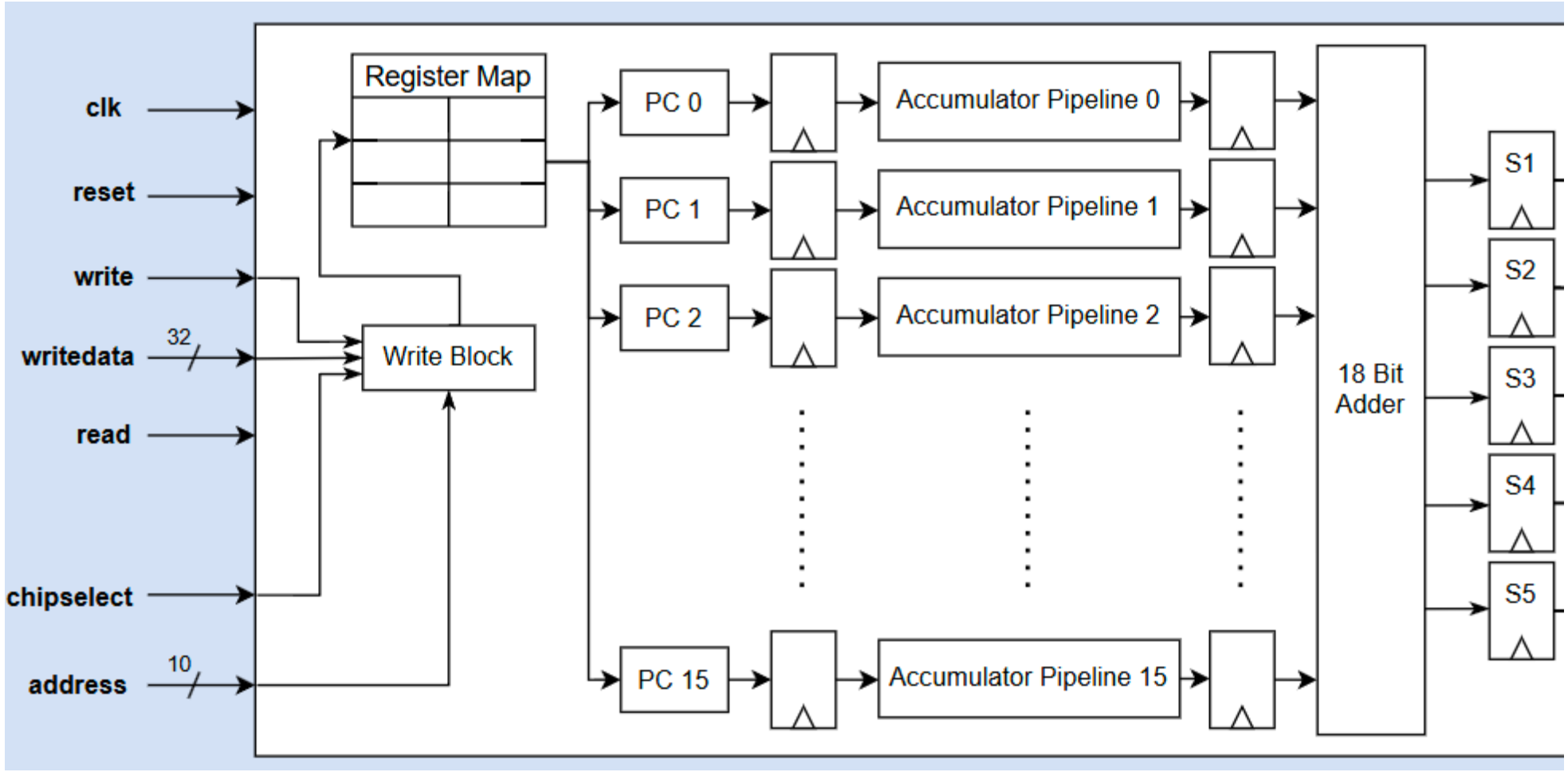
- Use accumulators and multipliers in int4.
- Convert to float at the end for precision.

LR Accelerator Component (Block Diagram)

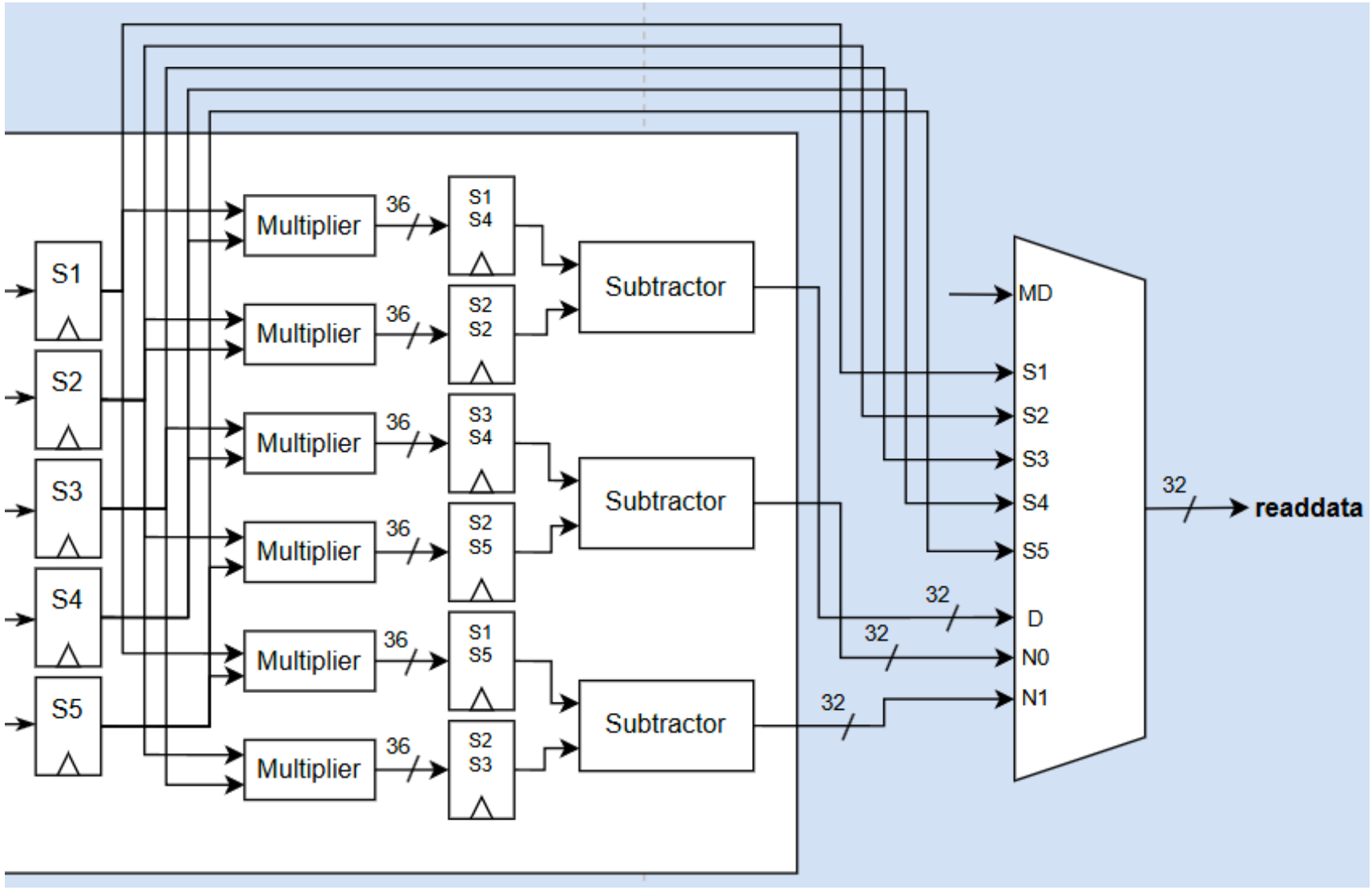
- 16 parallel pipelines accumulate data
- Adder sums pipeline outputs
- Processing unit computes products
- Register interface controls data flow



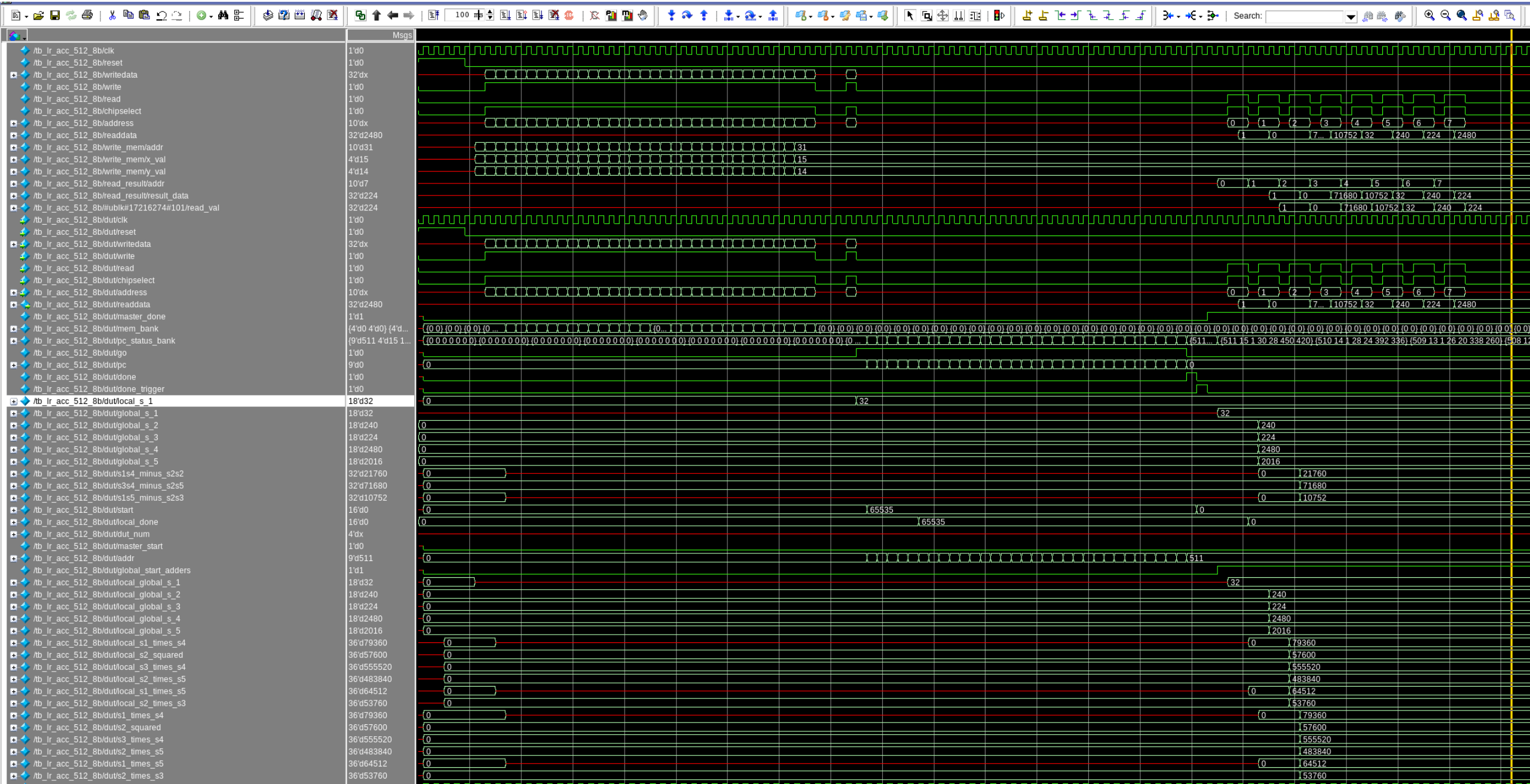
LR Accelerator Component (Accumulator Block)



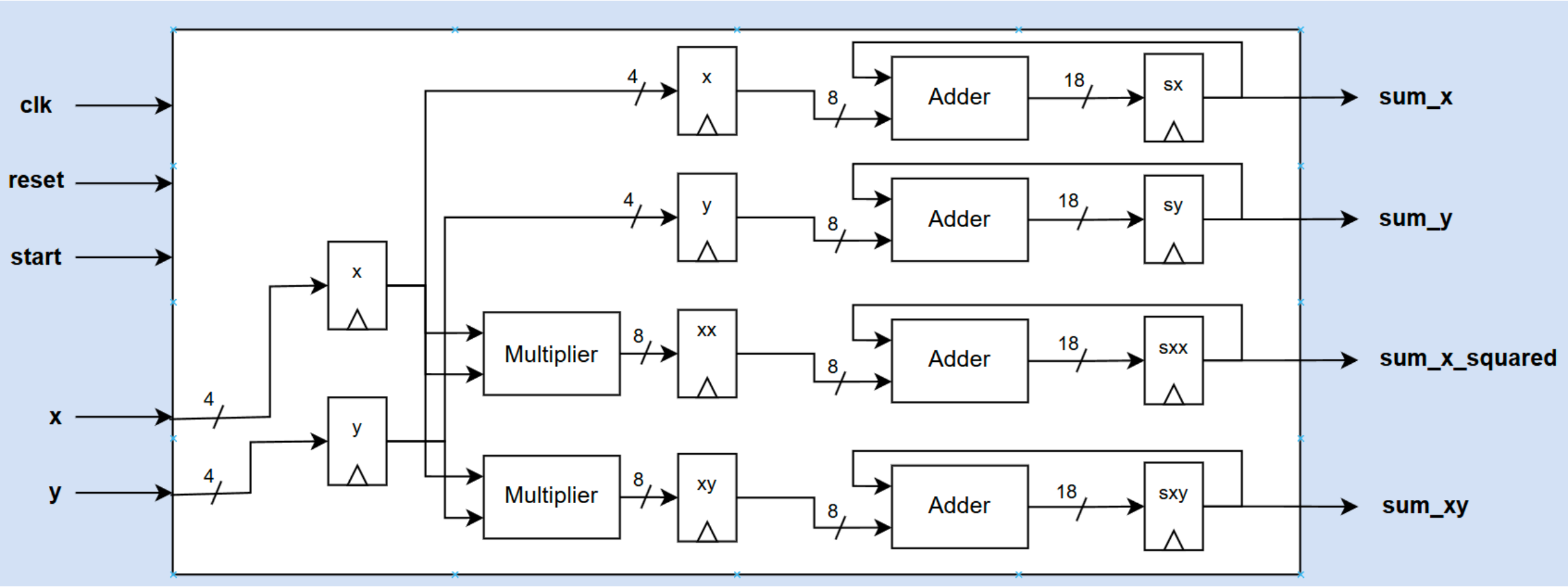
LR Accelerator Component (Weight Computation Block)



Timing Diagram for LR Accelerator Component



Accumulator Pipeline (Block Diagram)



Accumulator Pipeline (Analysis)

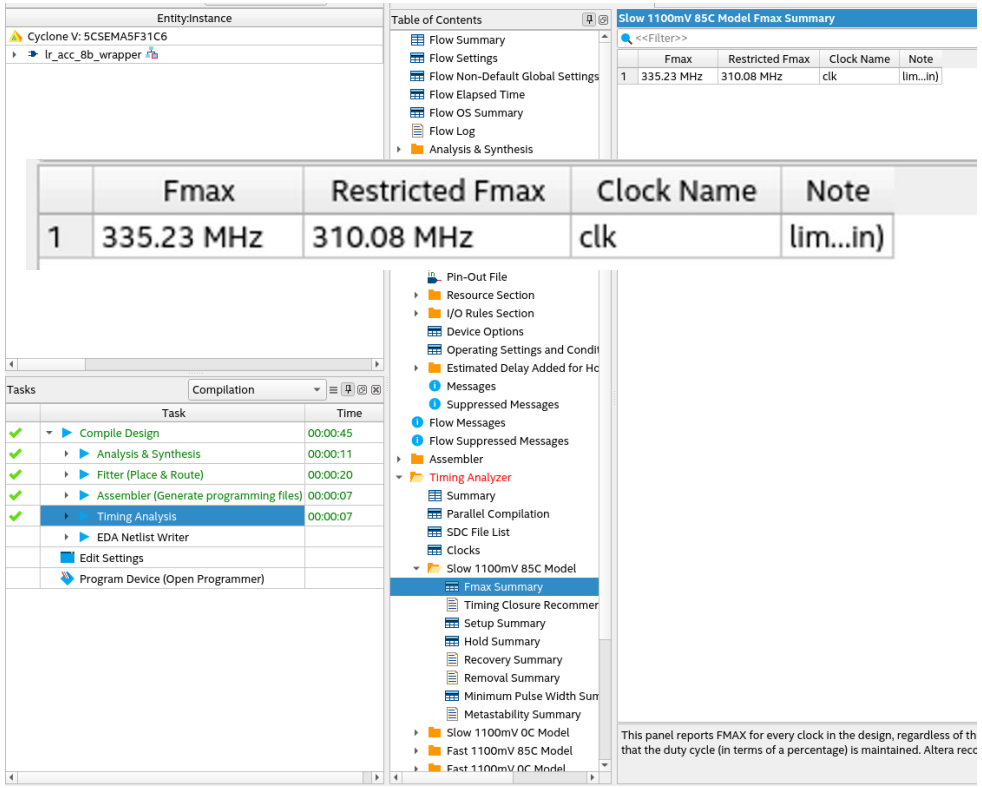
Fitter Summary

Fitter Summary

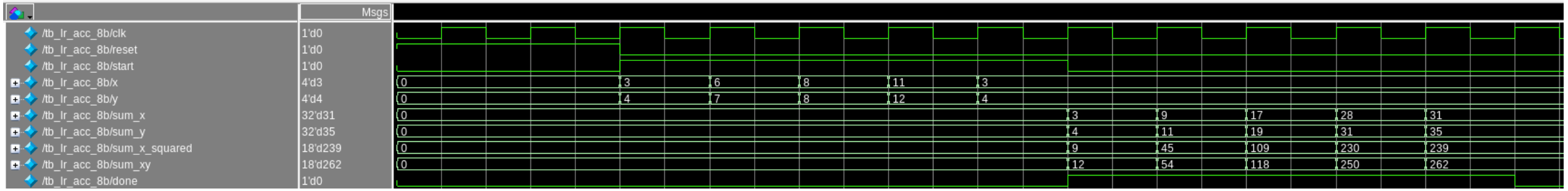
<<Filter>>

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 %)

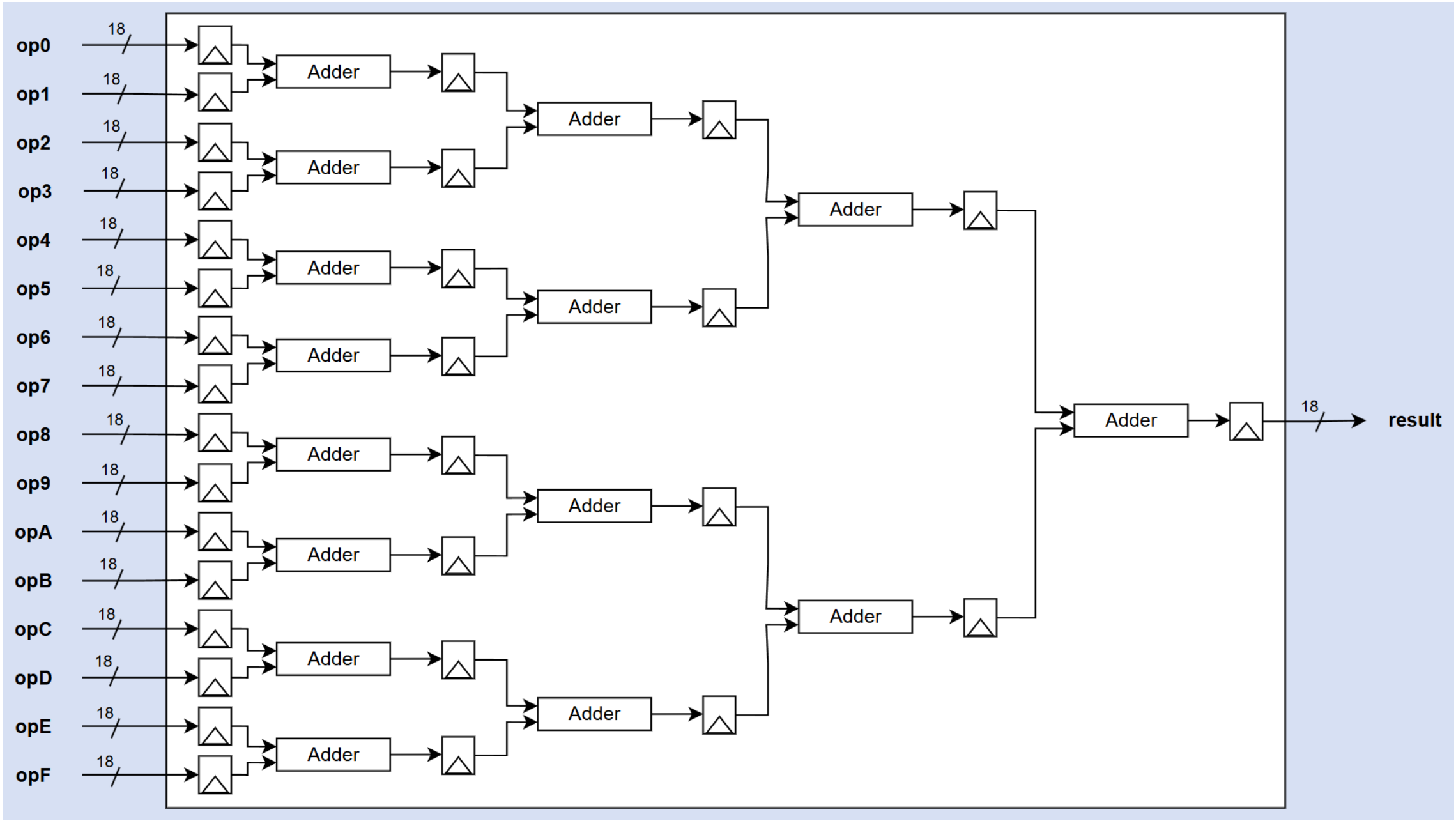
Timing Analyzer (Fmax)



Timing Diagram

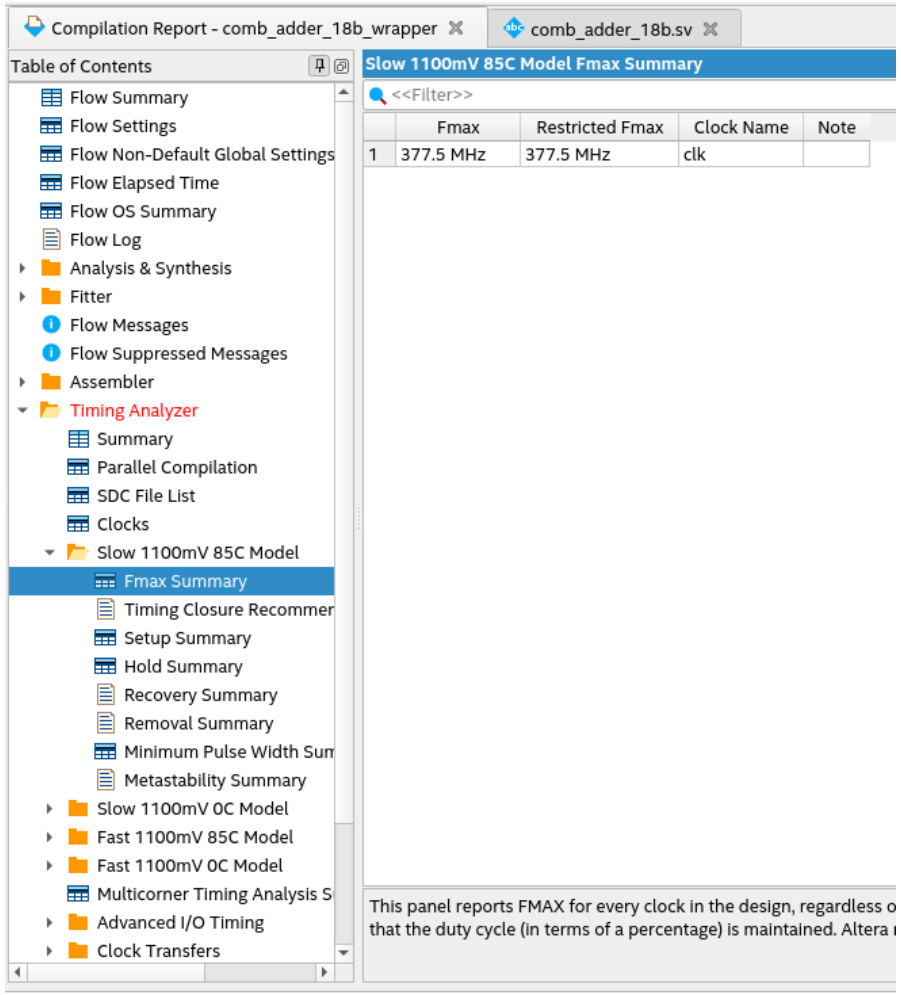


16 Number 18 Bit Adder Block

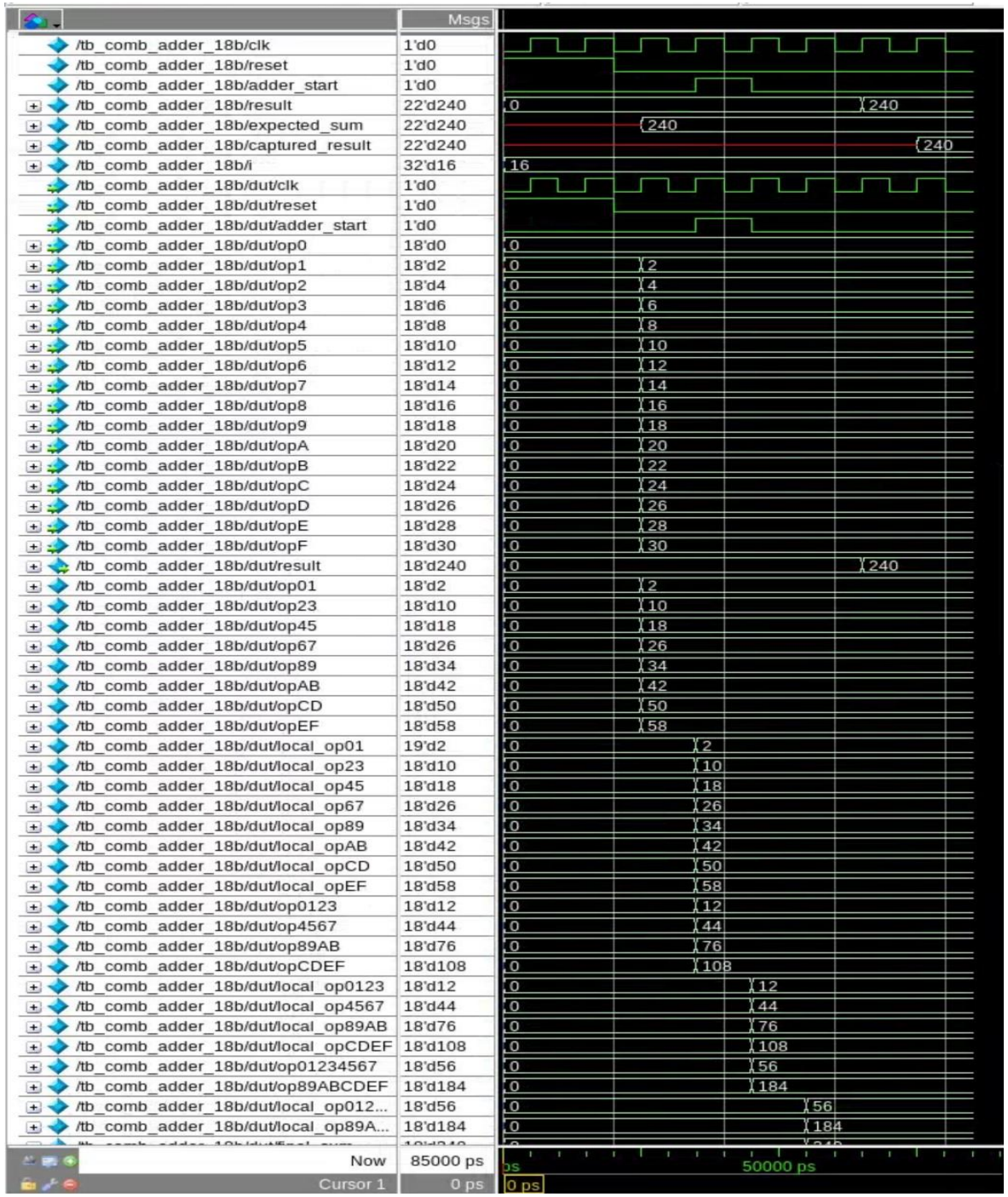


18 Bit Adder Block (Analysis)

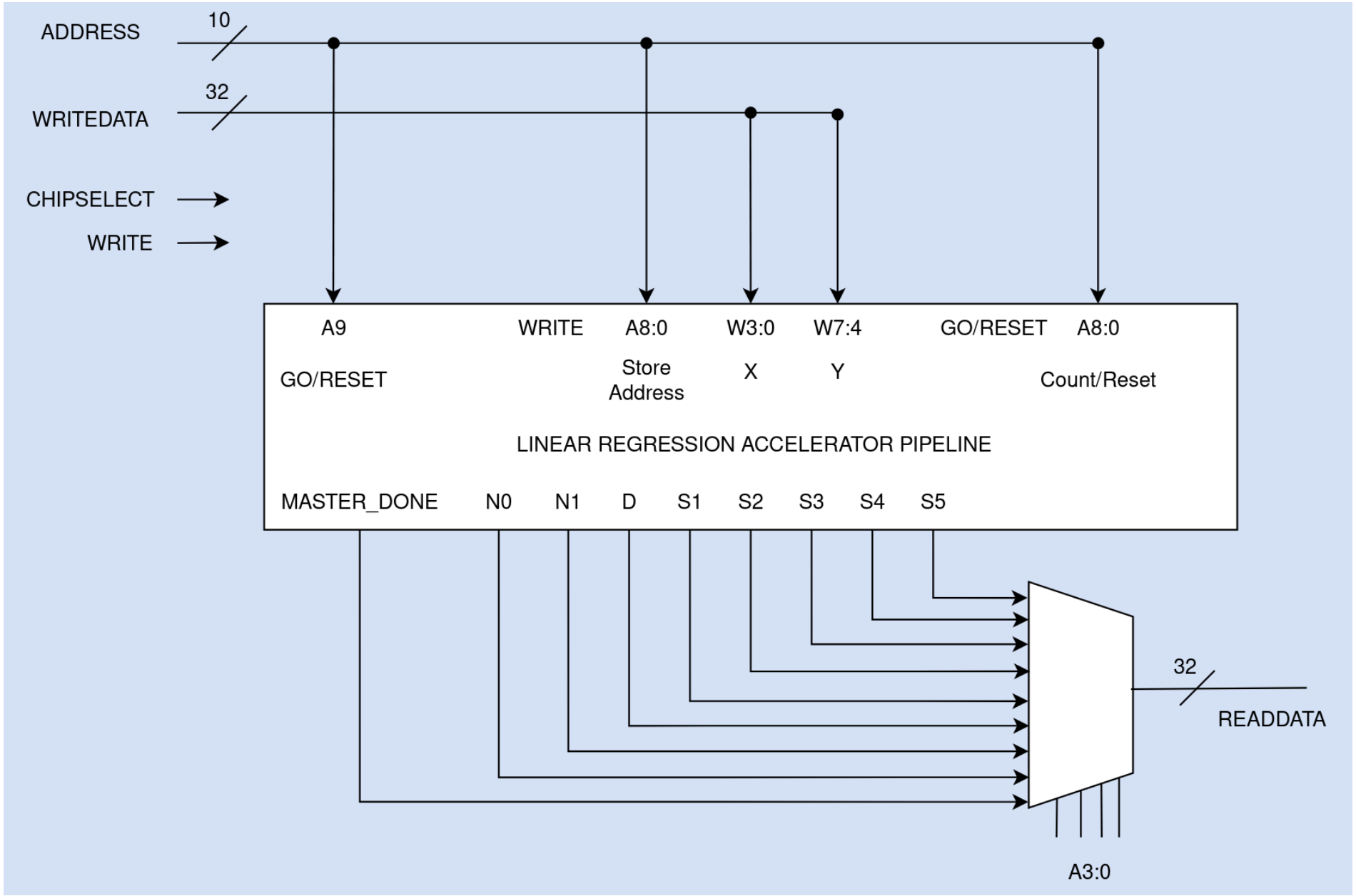
Timing Analyzer (Fmax)



Timing Diagram



LR Accelerator Component with Avalon MM Agent Interface



Address Encoding for Read Instructions

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)

Address Encoding for Write Instructions

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

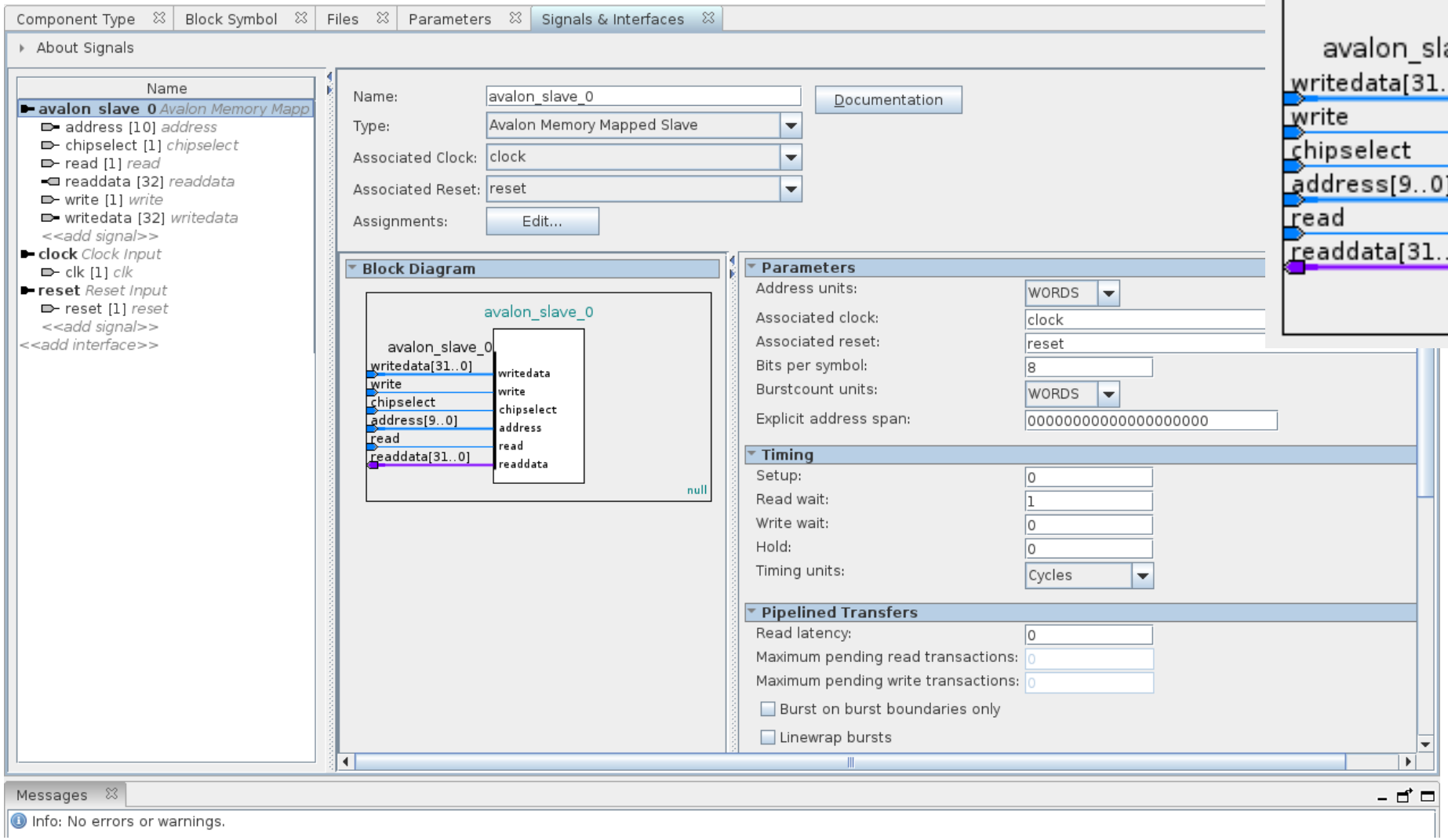
Memory Map for Read Instructions

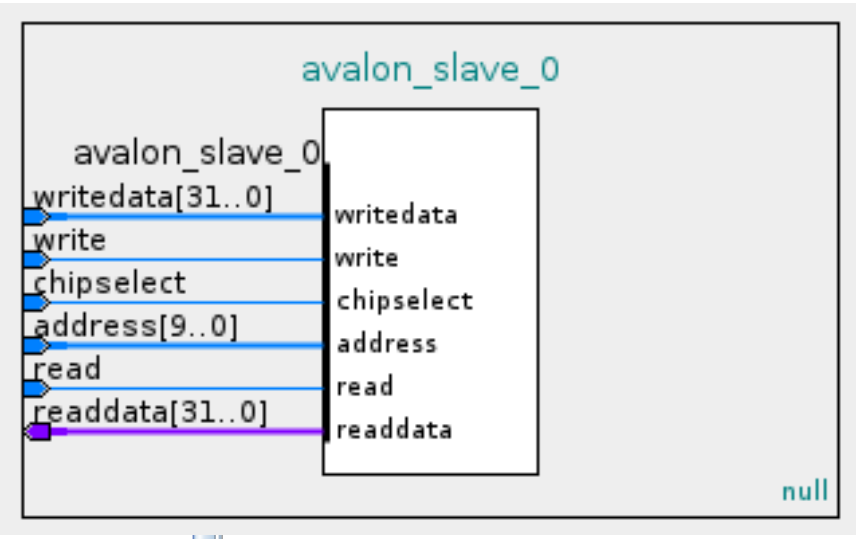
Offset (Hex)	Bit	Description
0	31:1	Unused
	0	Master Done
1	31:0	Signed N_0
2	31:0	Signed N_1
3	31:0	Signed D
4	31:0	Unsigned S_1
5	31:0	Unsigned S_2
6	31:0	Unsigned S_3
7	31:0	Unsigned S_4
8	31:0	Unsigned S_5

Memory Map for Write Instructions

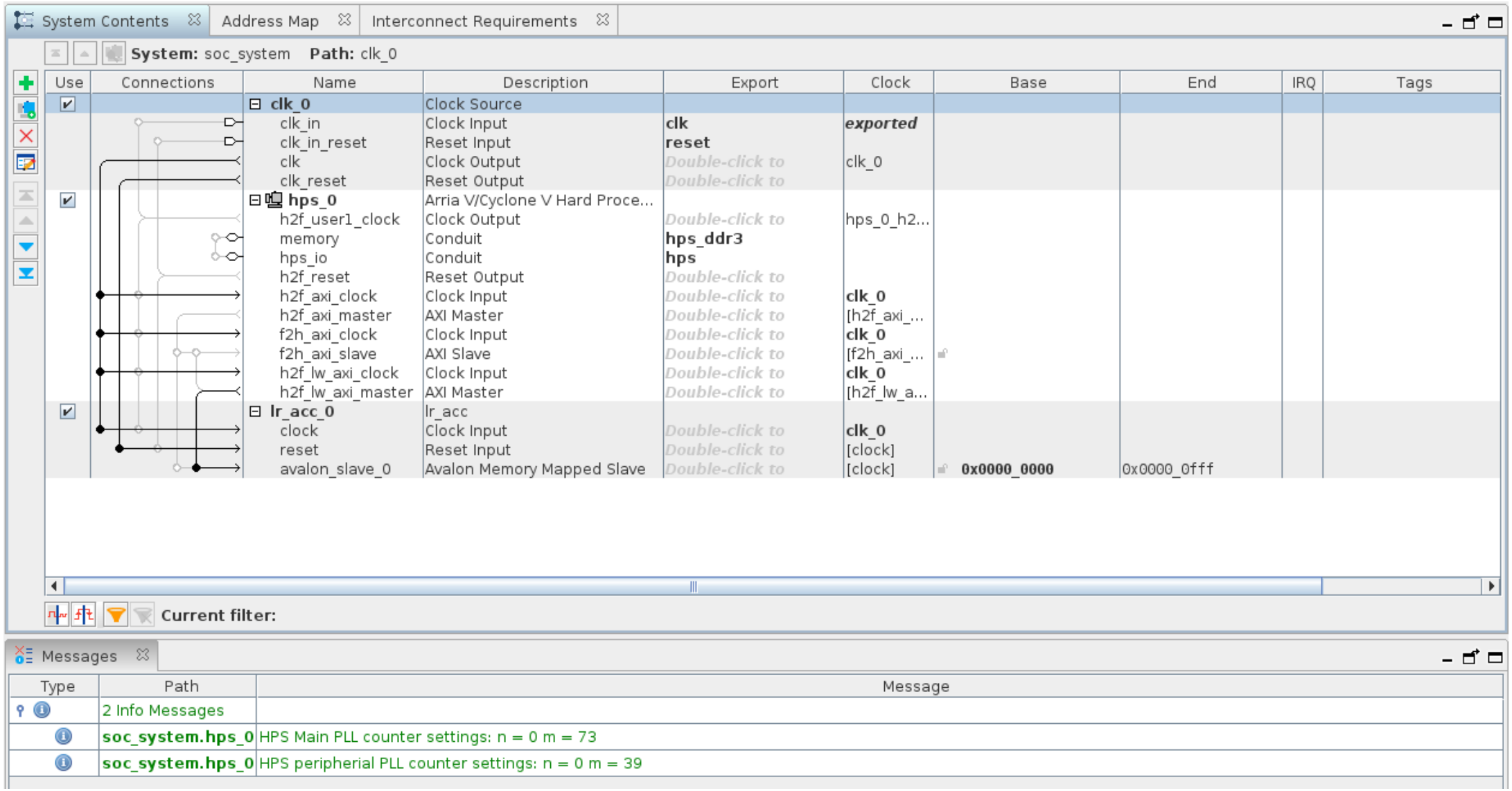
Offset (Hex)	Bit	Description
0 – 1FF	31:8	Unused
	7:4	y
	3:0	x
200 – 2FF	31:0	Unused

Signals and Interfaces





Platform Designer System



Software Code

```
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();
```

```
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();
```

```
start = clock();

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

end = clock();
```

```
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);
```

Software Code

```c
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;
  }
}
```

```c
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;
  }
}
```

```c
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;
}
```

Software Code

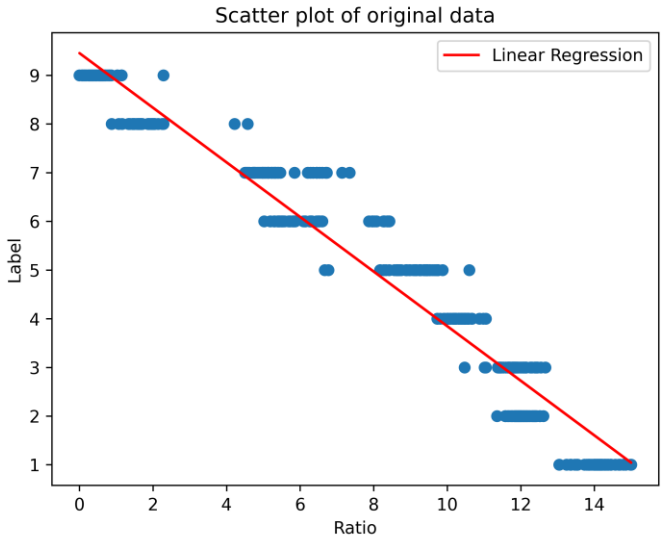
```
static void write_data(lr_acc_arg_t *data)
{
        if (data->go) {
                iowrite32((u32)1, dev.virtbase + 4 * ((1 << 9) + data->address));
        } else {
                iowrite32((u32)data->data.data, dev.virtbase + 4 * data->address);
        }

}
```

```
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;
```

Validation of Accelerator



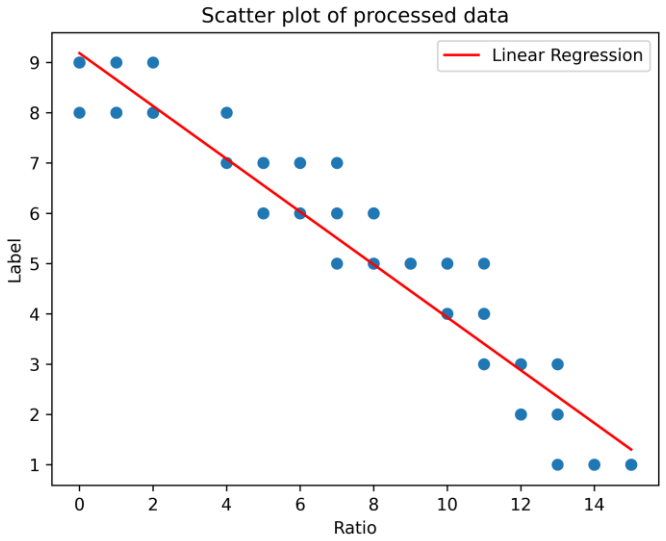
Fit Metrics

Slope : -0.560855

Intercept : 9.455583

R2 Score : 0.951513

MSE : 0.323244



Fit Metrics

Slope : -0.525380

Intercept : 9.183580

R2 Score : 0.948967

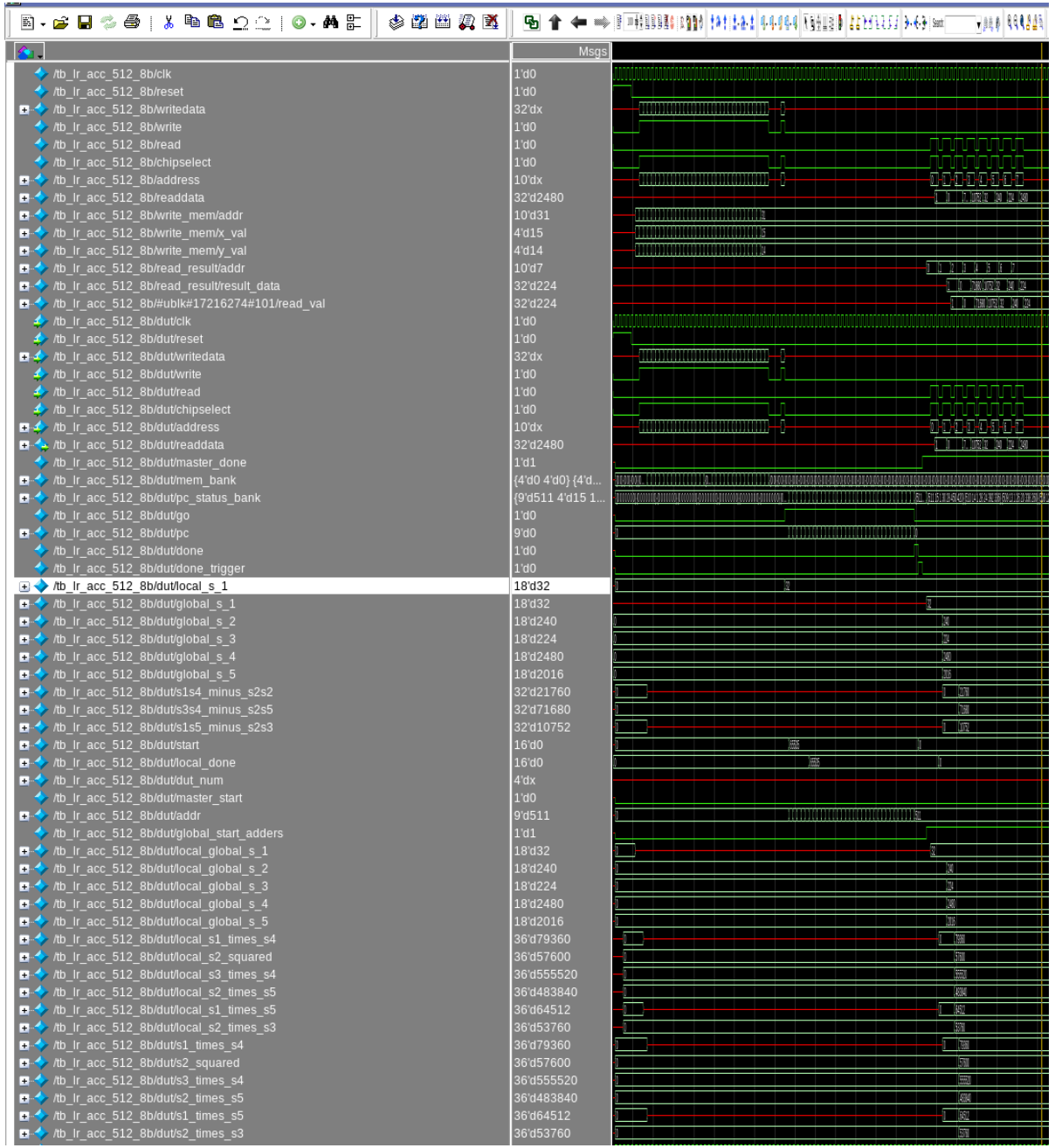
MSE : 0.340218

Param.	C Code (Local)	FGPA Accelerator
S1 :	432	432
S2 :	3440	3440
S3 :	2160	2160
S4 :	37294	37294
S5 :	11998	11998
N0 :	39281920	39281920
N1 :	-2247264	-2247264
D :	4277408	4277408
W0 :	9.183580	9.183580
W1 :	-0.525380	-0.525380

The various intermediate as well as final output generated by the accelerator match the ones generated by the baseline C code.

Runtime Analysis of Accelerator

- Runtime in Local Machine [Intel i9] (in Python): $154\mu s$
- Runtime in FPGA (in C): $28.9\mu s$
- Runtime with Accelerator:
  - Sending Data: $340\mu s$
  - Processing and Reading Data: $9\mu s$
  - Calculating Weights (Floating Operation): $2\mu s$



THANKS