Exploring Int-8 Optimization Methods for Multiply-Accumulation in Neural Networks

By Jan Silva and Oleh Kondratyuk
June 2021

Timing Analysis of Multiplier

Files: int8.sv
IP Modules used: clk_wiz_0, xbip_dsp48_macro_0

Using one instance of the DSP 48 Macro and a PLL generated by a clock wizard IP module, the highest clock frequency for one multiplication is 645 MHz. Two test weights and one test input were hard-coded.

```
// in_i * (w_k + w_j)
// A and D are inputs to pre-adder
xbip_dsp48_macro_0 dsp_slice (
  .CLK(clk),                      // input wire CLK
  .A({weight_j, {17{1'b0}}}),     // input wire [24 : 0] A - weight
  .B({{10{1'b0}}, input_i}),      // input wire [17 : 0] B - input
  .C(accum),                      // input wire [47 : 0] C - 3rd input -
                                  // used as another addition here for accumulation
  .D({{10{1'b0}}, weight_k}),     // input wire [17 : 0] D - weight
  .P(P)                           // output wire [47 : 0] P
);
```

Figure 1: DSP Macro Instantiation

Highest clock frequency = 645 Hz
Counter pipeline stages = 3
Output pipeline stages = 2

Fine-Tuning the Packed-Weight Multiplication

Files: macc.sv
IP Modules: xbip_dsp48_macro_0

Note: This report mainly focuses on signed numbers.

The figure below shows the implementation of packed-weight multiplication using the DSP48 macro's pre-adder, multiplier, and accumulator.

```
if (SIGN == 1)
    xbip_dsp48_macro_0 dsp_slice (
        .CLK(clk),
        // Move weight_j to upper bits
        .A($signed({weight_j[WIDTH-1], weight_j, {(24 - WIDTH){1'b0}}})),
        // Single input - sign extended
        .B($signed({{(18-WIDTH){input_i[WIDTH-1]}}, input_i})),
        // C port if no internal correction
        //.C($signed(accum)),
        // Apply internal correction - if lower bit term is negative, add 1 to upper bit term
        .C((weight_k[WIDTH-1] ^ input_i[WIDTH-1]) ? ({{(48-2*WIDTH-1){1'b0}}, 1'b1, {(2*WIDTH){1'b0}}}) : 0),
        // Move weight_k to lower bits
        .D($signed({{(25-WIDTH){weight_k[WIDTH-1]}}, weight_k})),
        // Output P
        .P({rep_sign, ji, ki})
    );
```

Figure 2: macc.sv. The two C port options will be further explained in the section “Four Methods of Correction and Accumulation”

The module macc.sv takes in two weights and an input, and returns results ji and ki. The module performs the operation `(wj + wk) * (i) = {ji, ki}` with `wj` and `wk` being the weights, `i` the input, and `ji` and `ki` the separated products. This method packs two weights into one multiplication port, effectively completing two input x weight multiplications in one DSP slice. Because of this packing, there is an upper term and a lower term in both the weight input to the multiplier, and the output of the multiplier. The following process outlines various situations in which overflow may occur and affect the padding bits in between these lower and upper terms.

For 8-bit two’s-complement numbers, we consider two values that can cause worst-case overflow: -128 (8’b1000000) and 127 (8’b01111111). We then tested several combinations of these values in the equation: `(wj + wk) * (i)` to observe the output behavior. One example hand-calculation is provided below:

```
wj = 8’d127, wk = 8’d127, i = 8’d127
127 * 127 =
                0000000000000000000000000000000011111100000001
(127 + 127) * 127 =
                0000000000000000001111110000000100111111000000001
                     extra bits     wj*i  padding bit    wk*i
-128 * -128 =
                0000000000000000000000000000000010000000000000000
(-128 + -128) * -128 =
                0000000000000000010000000000000001000000000000000
                     extra bits     wj*i  padding bit    wk*i
```

Using the method above for different bit-widths and worst-case values, we found the estimated number of padding for worst-case packed multiplication. The table below shows the calculated amount of padding bits for each input bit-width. See Appendix section II for more details on how these values were calculated.

Table 1: Padding Analysis

Width	**Range (worst-case values)**	**Padding (bits)**
Int-9	-256, -256	-1
Int-8	-128, -128	0
Int-7	-64, -64	3
Int-6	-32, -32	6
Int-5	-16, -16	9

Four Methods of Correction and Accumulation

Files: macc.sv, chain_adder_tree.sv, chain.sv
IP Modules: xbip_dsp48_macro_0

During the padding analysis, we found that there are slight quirks when certain combinations of negative and positive two's-complement numbers are multiplied together. For example, when the lower-bit term *wk* is negative and the upper-bit term and input *wj* and *i* are positive, the resulting product term *ji* is one bit lower than expected.

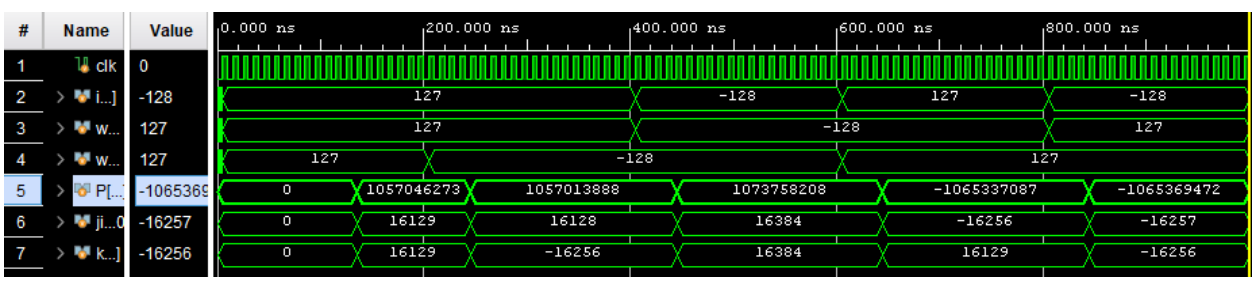


Figure 3: 127 x (127 + 127) results in ji = ki = 16129, but 127 * (127 + -128) results in ji = 16128 instead of ji = 16129.

To correct these quirks, we explored four different methods of multiply-accumulation. First, we examined the difference between using an internal correction and an external adder tree, and an external correction and internal adder tree. We also tested methods

with both correction and no correction to investigate the effects of corrections on resource utilization and timing. If the effects of the corrections are close to negligible, it may be worth it to manually manipulate the weights to avoid such quirks.

- Internal Correction, External Adder Tree
    - The correction for the one-bit off phenomenon is applied through a simple conditional statement in the C port of the DSP. The C port can no longer be used to accumulate results, and an external adder tree can be implemented to accumulate multiplications. See chain.sv for this implementation.
- External Correction, Internal Adder Tree
    - A separate adder tree, external to the DSP slices was needed to accumulate the errors that occur in the process of chaining DSP slices together. Whether a chained pair of DSP slices required a correction (a +1 or -1 to the result) was determined by some logic taking in the sign bit of the 1) Input, 2) `wk` 3) preceding DSP result and 4) overall DSP result. Some of these depend on each other, so some pipelining was required to properly apply the correction logic. The final sum of the adder tree is added to the result of the overall adder tree to come to a final, correct, result.
- No correction, External Adder Tree
- No correction, Internal Adder Tree

We tested these cases with 5, 8-bit inputs first, then ran synthesis and implementation with 1024 inputs. This required a new top-level module called test_use_case that read 1024 input values and weights from memory files.

Resource Utilization Analysis

Files: macc.sv, chain_adder_tree.sv, chain.sv, test_use_case.sv
IP Modules: xbip_dsp48_macro_0

Note: The results below are slightly outdated. As of 6/8/2021, we implemented a new binary adder tree which raised the LUT usage for both of the modules significantly.

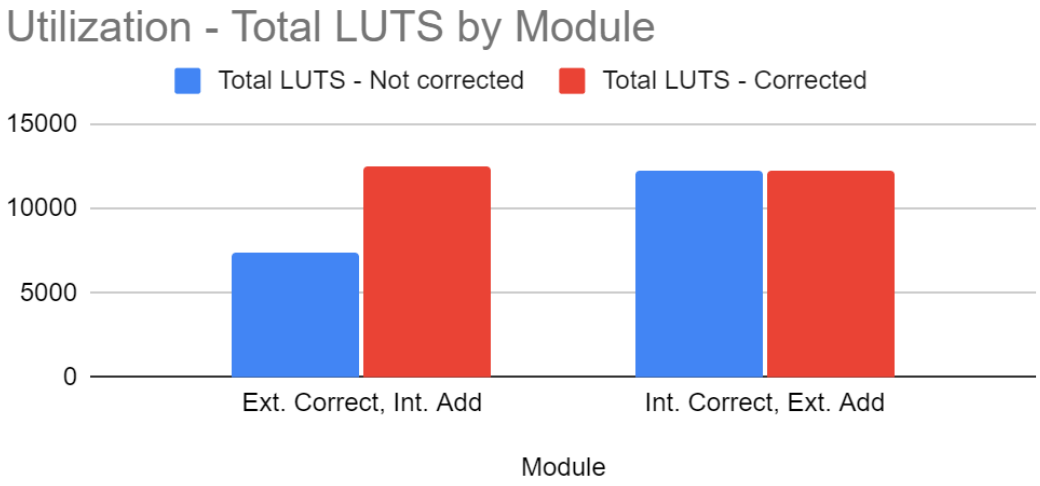


Figure 4: Total LUTs by Module

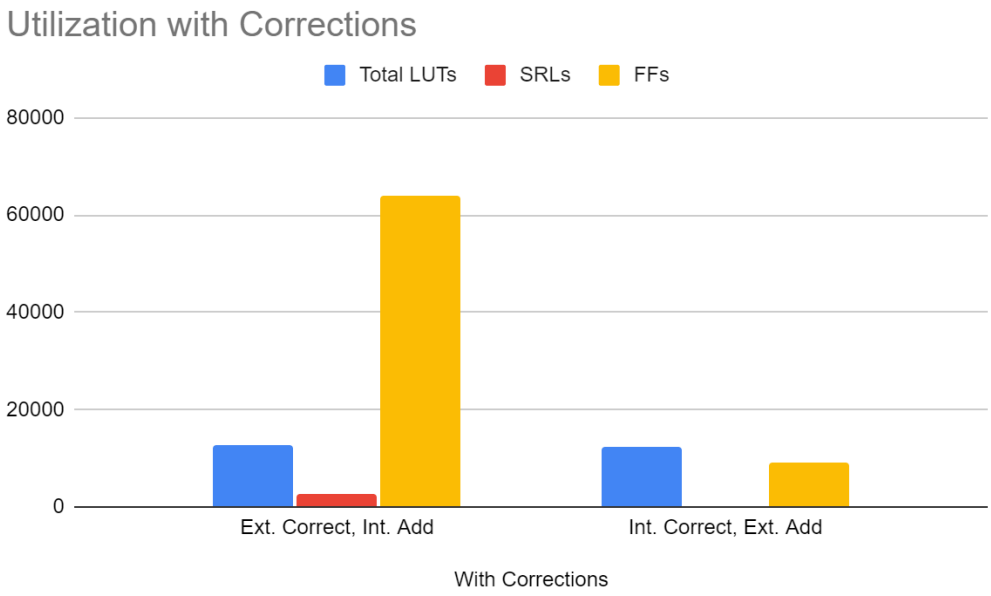


Figure 5: Utilization of each method with corrections

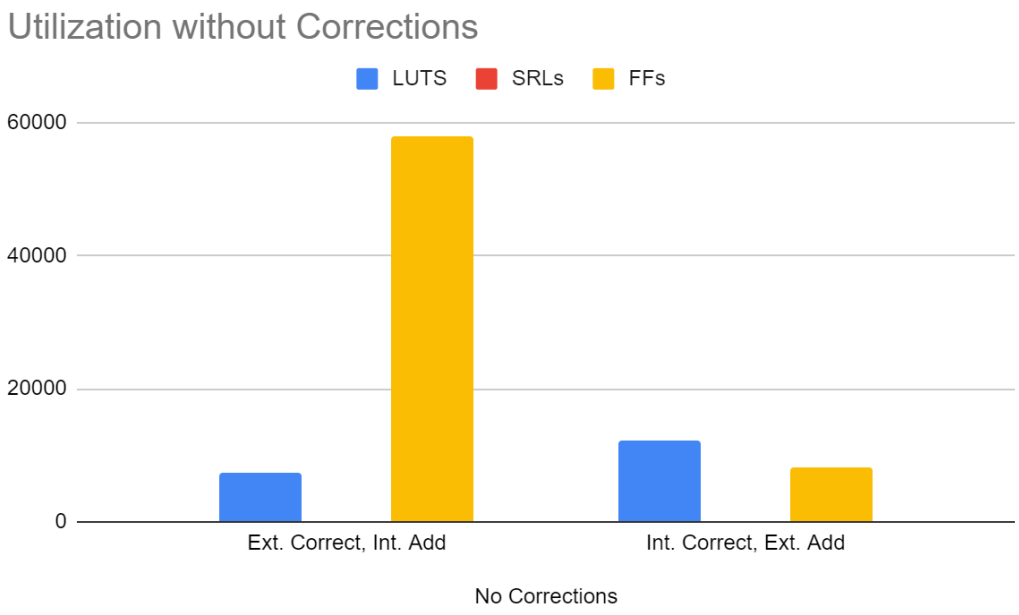


Figure 6: Utilization of each method without corrections

<u>Timing Analysis</u>

Files: macc.sv, chain_adder_tree.sv, chain.sv, test_use_case.sv, timed_case.sv
IP Modules: xbip_dsp48_macro_0, clk_wiz_0

For the timing analysis, we created a new top-level module called timed_case with a PLL to generate a 400 MHz clock. This module contains an instance of test_use_case, which can be modified to implement each of the four methods.

400 MHz

File	Method	Synthesis	Implementation
chain_adder_tree	External correction, internal addition	WNS = 0.778 ns WHS = 0.043 ns WPWS = 0.607 ns	WNS = -0.113 ns WHS = 0.043 ns WPWS = 0.607 ns (Failed timing)
chain	Internal correction, external addition	WNS = 0.855 ns WHS = 0.049 ns WPWS = 0.899 ns	WNS = 0.097 WHS = 0.023 WPWS = 0.849

WNS = Worst Negative Slack, WHS = Worst Hold Slack, WPWS = Worst Pulse Width Slack

Conclusion

From the above resource utilization reports and timing analyses, the internally-corrected multiply-accumulate implementation can handle faster clock speeds. While the total LUT usage of the uncorrected internal adder method is much lower, the method still uses a much higher number of flip-flops. The corrected versions of both the internal adder and external adder methods have similar LUT usage, but the latter uses much less FFs. Thus, the internally corrected method with the external adder tree is more resource-efficient, and along with the potential to go faster than 400MHz, this method seems to be the better one out of the four.

References

https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

Appendices

I. Multiplier Timing Spreadsheet

II. Screenshots of Simulations for analyzing Int-5 to Int-9

A. Calculated binary values

1. d'127 x d'127 =
0000000000000000000000000000000000 011111100000001

2. D'127 x -d'128 =
0000000000000000000000000000000000 11111110000000

3. -d'128 x d'127 =
0000000000000000000000000000000000 11111110000000
(-1111111000000)

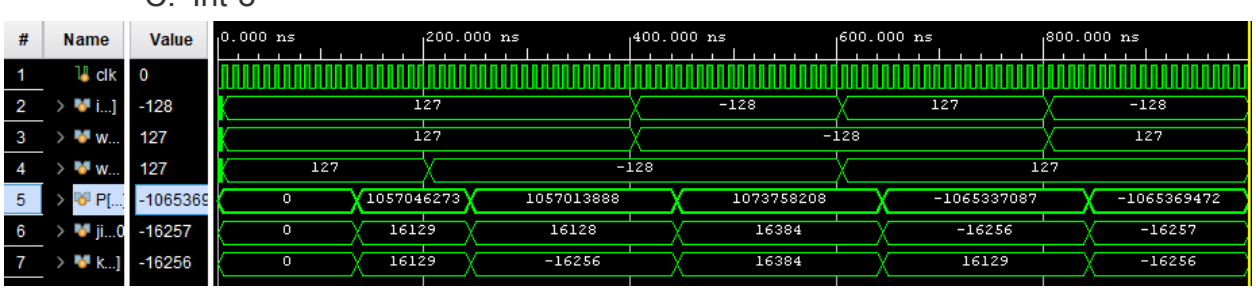
4. -d'128 x -d'128 =
0000000000000000000000000000000000 0100000000000000

B. Int-9

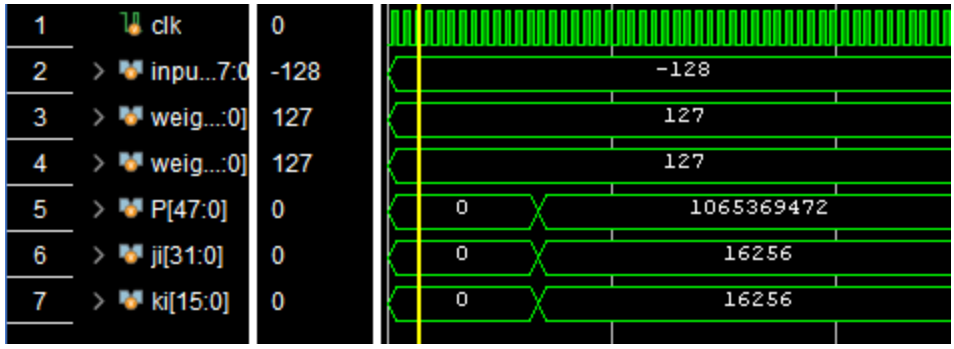
1. 0000000000000000011111100000000101111110000000001
1111111000000001

2.

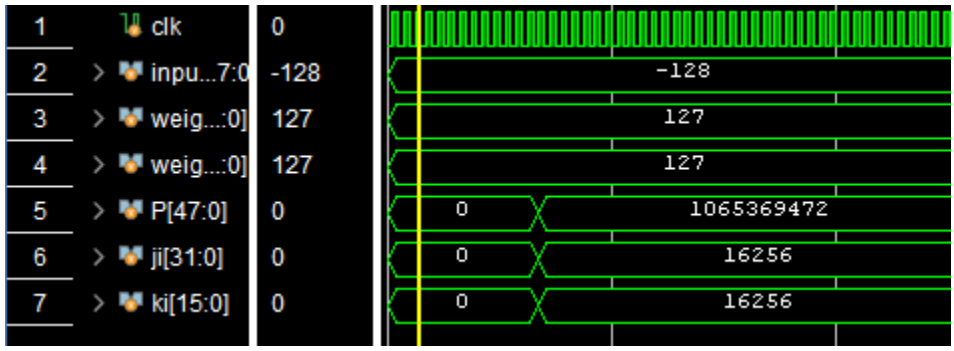
C. Int-8



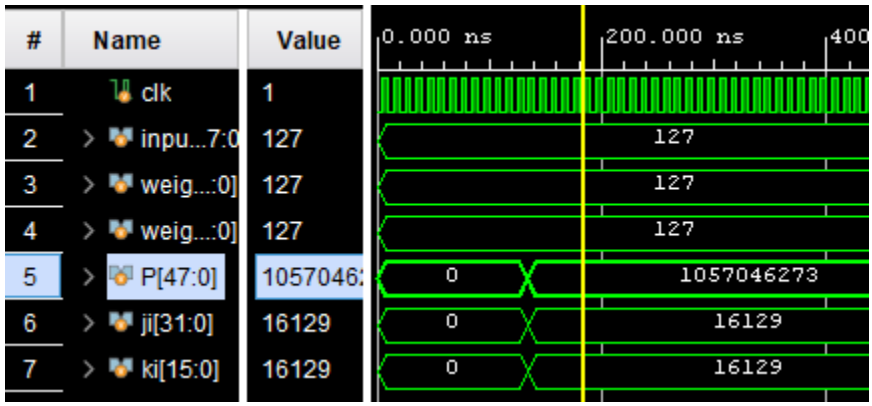
1. -128 x (127 + 127)



2. 127 x (-128 + -128)

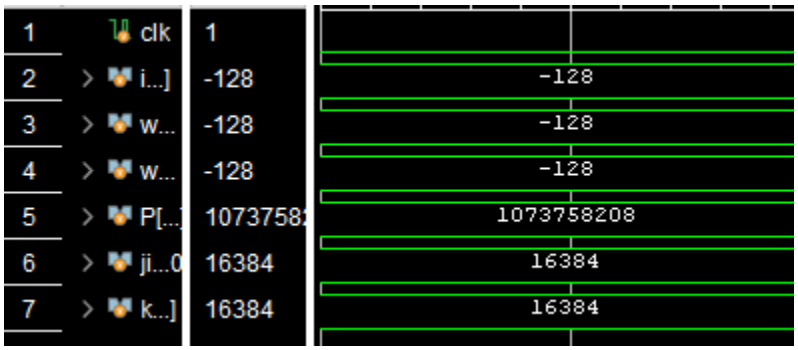


3. 127 x (127 + 127)



4. -128 x (-128 + -128) =

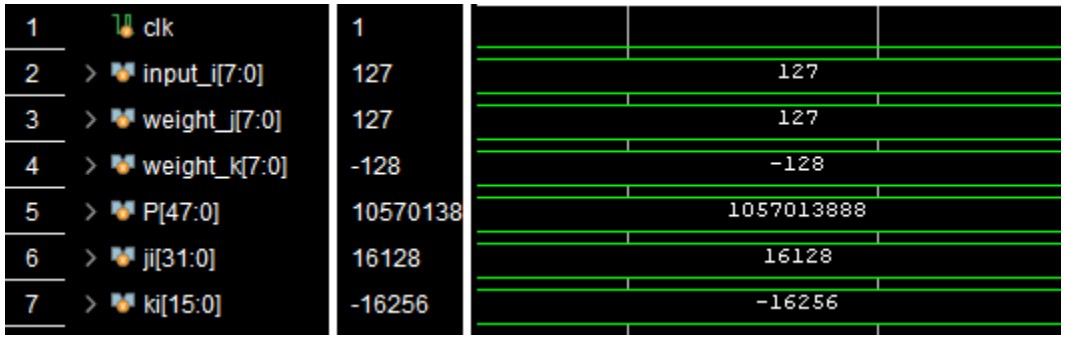
11111111111111111 100000000000000 0100000000000000



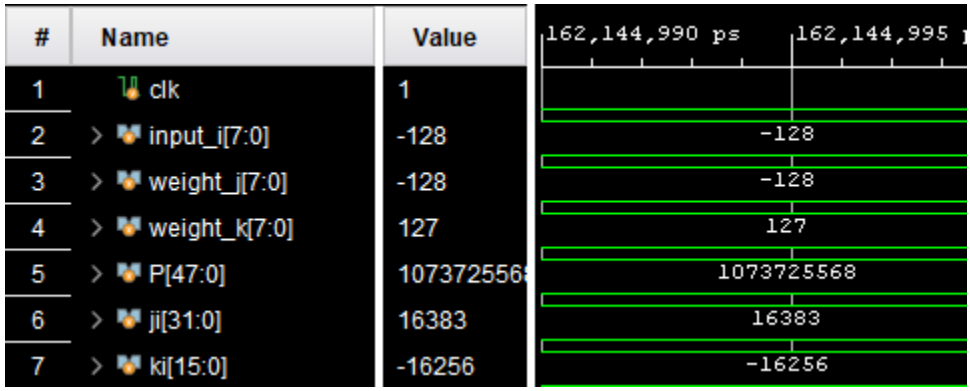
5. -128 x (127 + -128) =

00000000000000000011111110000000 0100000000000000

6. -128 x (-128 + 127) =
   11111111111111111100000000000000 00 11111110000000
7. `127 x (127 + -128)`

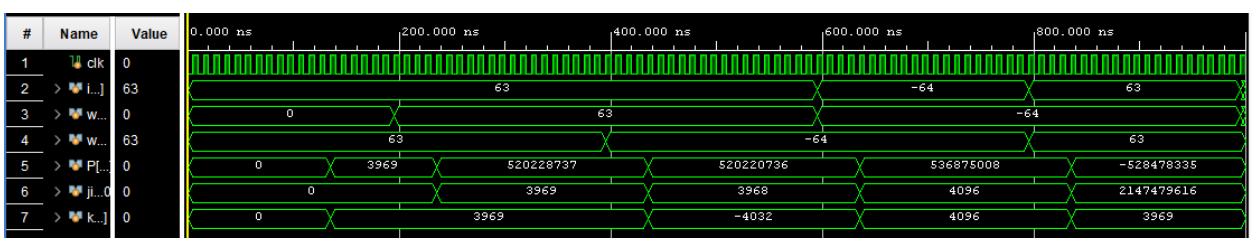


8. `127 x (-128 + 127)`



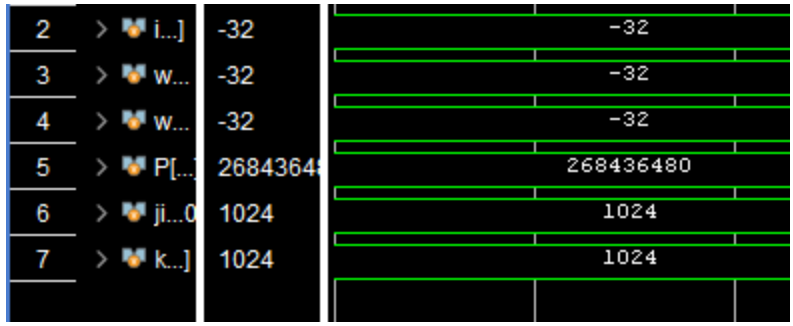
D. Int-7

1. `(-64 x (-64 + -64))`
2. `00000000000000000`
   `01 0000 0000 0000` **`000`** `01 0000 0000 0000`
   `assign ji = P[47:17]; assign ki = P[13:0];`



E. Int-6

1. `(-32 x (-32 + -32))`

2. 

3. 000000000000000000 010000000000 **000000** 010000000000
4. 6 bit
5. Assign ji = P[47:18], assign ki = P[11:0]

F. Int-5

1. 000000000000000000 **0100000000** 000000000 0100000000
2. 9 bit

G. Int-4

1. 7x(7+7)
2. 00000000000000000000000110001 **0000 0000 0000** 0011 0001
3. 12-bit

III. Timing Analysis Summaries at 400 MHz

A. Chain_adder_tree

1. Synthesis

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.778 ns	Worst Hold Slack (WHS):	0.043 ns	Worst Pulse Width Slack (WPWS):	0.607 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	238702	Total Number of Endpoints:	238702	Total Number of Endpoints:	178292

All user specified timing constraints are met.

2. Implementation

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0.113 ns	Worst Hold Slack (WHS):	0.043 ns	Worst Pulse Width Slack (WPWS):	0.607 ns
Total Negative Slack (TNS):	-0.113 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	1	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	110168	Total Number of Endpoints:	110168	Total Number of Endpoints:	86622

Timing constraints are not met.

B. Chain

1. Synthesis

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.855 ns	Worst Hold Slack (WHS):	0.049 ns	Worst Pulse Width Slack (WPWS):	0.899 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	232465	Total Number of Endpoints:	232465	Total Number of Endpoints:	191819

All user specified timing constraints are met.

2. Implementation

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.097 ns	Worst Hold Slack (WHS):	0.023 ns	Worst Pulse Width Slack (WPWS):	0.849 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	49057	Total Number of Endpoints:	49057	Total Number of Endpoints:	50087

All user specified timing constraints are met.

IV. Registers as a Flip Flop

A. Chained_adder_tree

test_use_case	69937
chains (chained_adder_tree)	63655
Leaf Cells (5959)	5959
maccs[0].even_macc (chained_macc__xdcDup__1)	104
dsp_slice (xbip_dsp48_macro_0_HD1024)	104
U0 (xbip_dsp48_macro_0_xbip_dsp48_macro_v3_	104
i_synth (xbip_dsp48_macro_0_xbip_dsp48_mac	104
i_synth_option.i_synth_model (xbip_dsp48_m	104
i_c4 (xbip_dsp48_macro_0_xbip_pipe_v3_	48
i_has_c.i_c3 (xbip_dsp48_macro_0_xbip_	48
i_op3 (xbip_dsp48_macro_0_xbip_pipe_v3	4
i_op4 (xbip_dsp48_macro_0_xbip_pipe_v3	4

B. Chain

Register as Flip Flop

Name	Used
timed_case	49057
test (test_use_case)	49057
dut (chain)	49057
Leaf Cells (3986	39869
eachSlice[0].slic	10
dsp_slice (xb	10
U0 (xbip_c	10
eachSlice[1].slic	10
eachSlice[5].slic	10
eachSlice[6].slic	10
eachSlice[7].slic	10
eachSlice[9].slic	10

V. Resource Utilization Summaries

A. External adder tree with internal correction (chain)

Resource	Utilization	Available	Utilization %
LUT	83222	433200	19.21
FF	186880	866400	21.57
DSP	1280	3600	35.56
IO	30785	850	3621.76

B. Internal adder tree with external correction (chained_adder_tree)

Resource	Utilization	Available	Utilization %
LUT	81914	433200	18.91
FF	186880	866400	21.57
DSP	1280	3600	35.56
IO	30773	850	3620.35

C. Internal adder tree, no correction (chain)

Resource	Utilization	Available	Utilization %
LUT	64000	433200	14.77
FF	186880	866400	21.57
DSP	1280	3600	35.56
IO	30785	850	3621.76

VI. Early work on observing overflow

01111111 x 01111111

0000000000000000000000000000000 0011111100000001 term1
0000000000000000000000000000000 0111111000000010 term2
0000000000000000000000000000000 1011110100000011 term3
0000000000000000000000000000000 1111110000000100 term4
0000000000000000000000000000001 0011101100000101 term 5 << overflow

10000000 x 10000000

0000000000000000000000000000000 0000000000000000 term 0
0000000000000000000000000000000 0100000000000000 term 1
0000000000000000000000000000000 1000000000000000 term 2
0000000000000000000000000000000 1100000000000000 term 3
0000000000000000000000000000001 0000000000000000 term 4 << overflow

10000000 x 01111111

0000000000000000000000000000000 0000000000000000 term 0
0000000000000000000000000000000 0011111110000000 term 1
0000000000000000000000000000000 0111111100000000 term 2
0000000000000000000000000000000 1011111010000000 term 3
0000000000000000000000000000000 1111111000000000 term 4
0000000000000000000000000000001 0011110110000000 term 5 << overflow

Int 8 opt with 01111111

000000000000000 0000000000000000 0 0000000000000000 term0
000000000000000 0011111100000001 0 0011111100000001 term1
000000000000000 0111111000000010 0 0111111000000010 term2
000000000000000 1011110100000011 0 1011110100000011 term3
000000000000000 1111110000000100 0 1111110000000100 term4
000000000000001 0011101100000101 1 0011101100000101 term5 << overflow

11111111 x 11111111

0000000000000000000000000000000 0 0000000000000000 term0
0000000000000000000000000000000 0 1111111000000001 term1
0000000000000000000000000000000 1 1111110000000010 term2 << overflow