Computer Arithmetic Training

Generic Implementation & Comparison of Adders Algorithms

By: Youssef Gamal Eldein

Introduction

Why Adders Matter:

- Fundamental building blocks in digital systems
- Critical for arithmetic operations in DSP, space applications, processors, and embedded systems
- Enable efficient computation in CPUs, GPUs, and FPGAs
- Impact performance, power, and area in hardware designs

Introduction

Challenges with Ripple Carry Adder (RCA):

- Slow propagation delay due to carry chain
- Linear delay increases with bit width, limiting performance
- Inefficient for high-speed applications

The + Operator Issue:

- Commonly used in RTL designs without optimization awareness
- Relies on synthesis tools, but are they optimal?
- Lack of insight into whether faster or more efficient adder architectures exist
- Motivation: Investigate if alternatives outperform the default + operator (Golden Model)

Introduction

Project Flow:

- Studied adder architectures to develop generic, synthesizable RTL patterns (hardest part)
- Implemented 10 adder architectures & Golden Model (+ operator) in RTL
- Performed functional simulation for verification
- Extracted Power, Performance, Area (PPA) metrics (Adder Sizes Tested 8-bit, 16-bit, 32-bit, 64-bit to analyze growth trends)
- Compared all adders across FPGA and ASIC technologies

Bold Assumptions:

- All adders assume cin = 0 (no external carry-in) to simplify generic algorithm development
- Prefix adders use only Black Cells for carry generation units (no Gray Cells) for simplicity

Introduction

Types of Adders Explored:

- Basic Adders: Carry Ripple Adder (RCA) (1)
- Carry Skip Adder (2)
- Carry Increment Adder: Optimized version of Carry Select (3)
- Carry Lookahead Adder: (CLA) Efficient for medium bit widths (4)
- Fast Prefix Adders: Kogge-Stone, Brent-Kung, Sklansky (5,6,7)
- Hybrid Prefix Adders: Ladner-Fischer, Han-Carlson, Knowles (8,9,10)

Comparison Sections:

- Section 1: Fast Prefix Adders (Kogge-Stone, Brent-Kung, Sklansky, Ladner-Fischer, Han-Carlson, Knowles) vs. Golden Model
- Section 2: Carry Increment, CLA, Carry Ripple, Carry Skip vs. Golden Model

Introduction

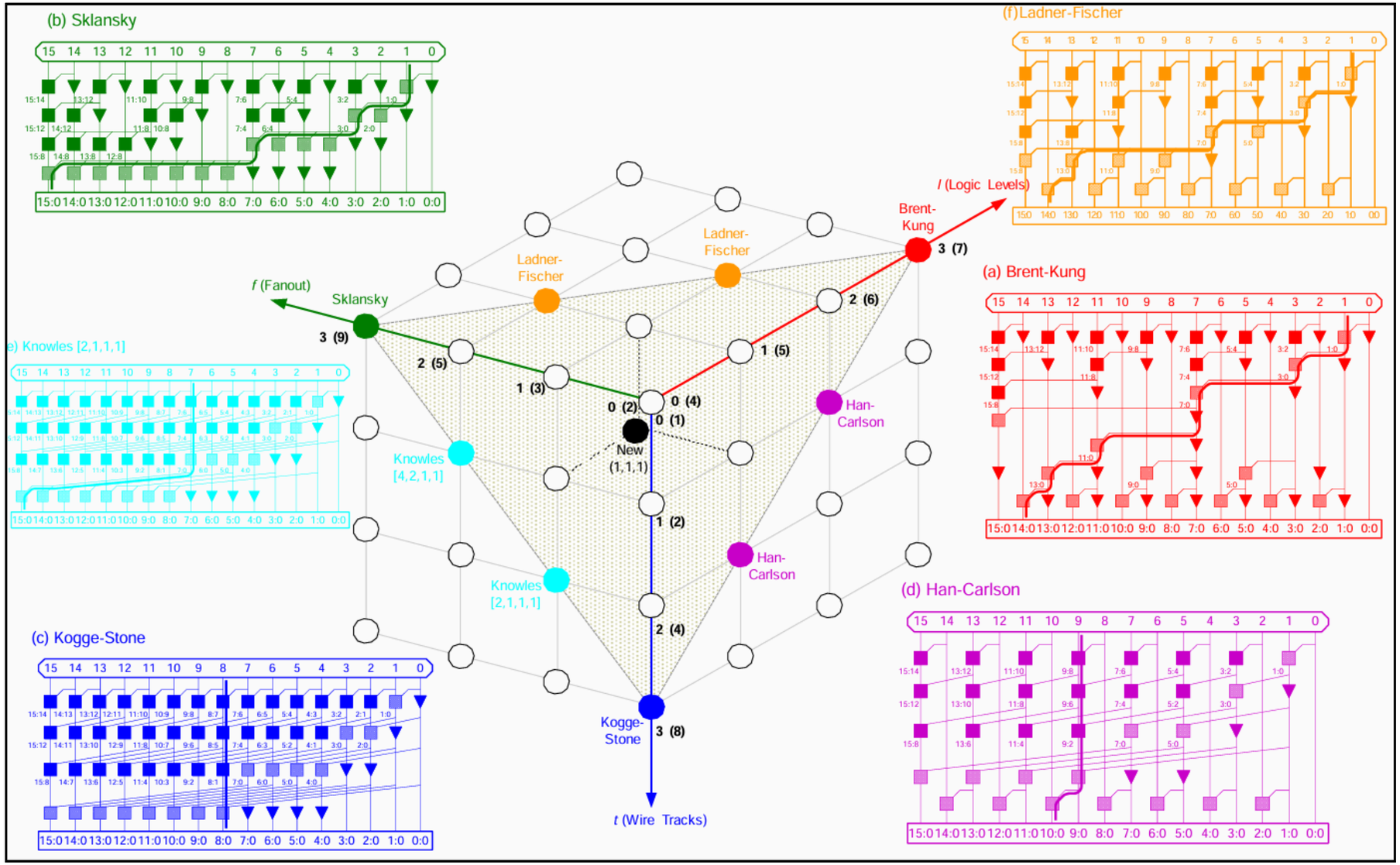
Power Analysis Methodology:

- **FPGA:**
  - **Used .SAIF files for post-implementation functional simulation**
  - **Calculated power per operation by dividing total power by number of operations**
  - **Enhanced accuracy and confidence in results**
- **ASIC (Sky130nm):**
  - **Relied on default power calculation reports from OpenLane Flow**

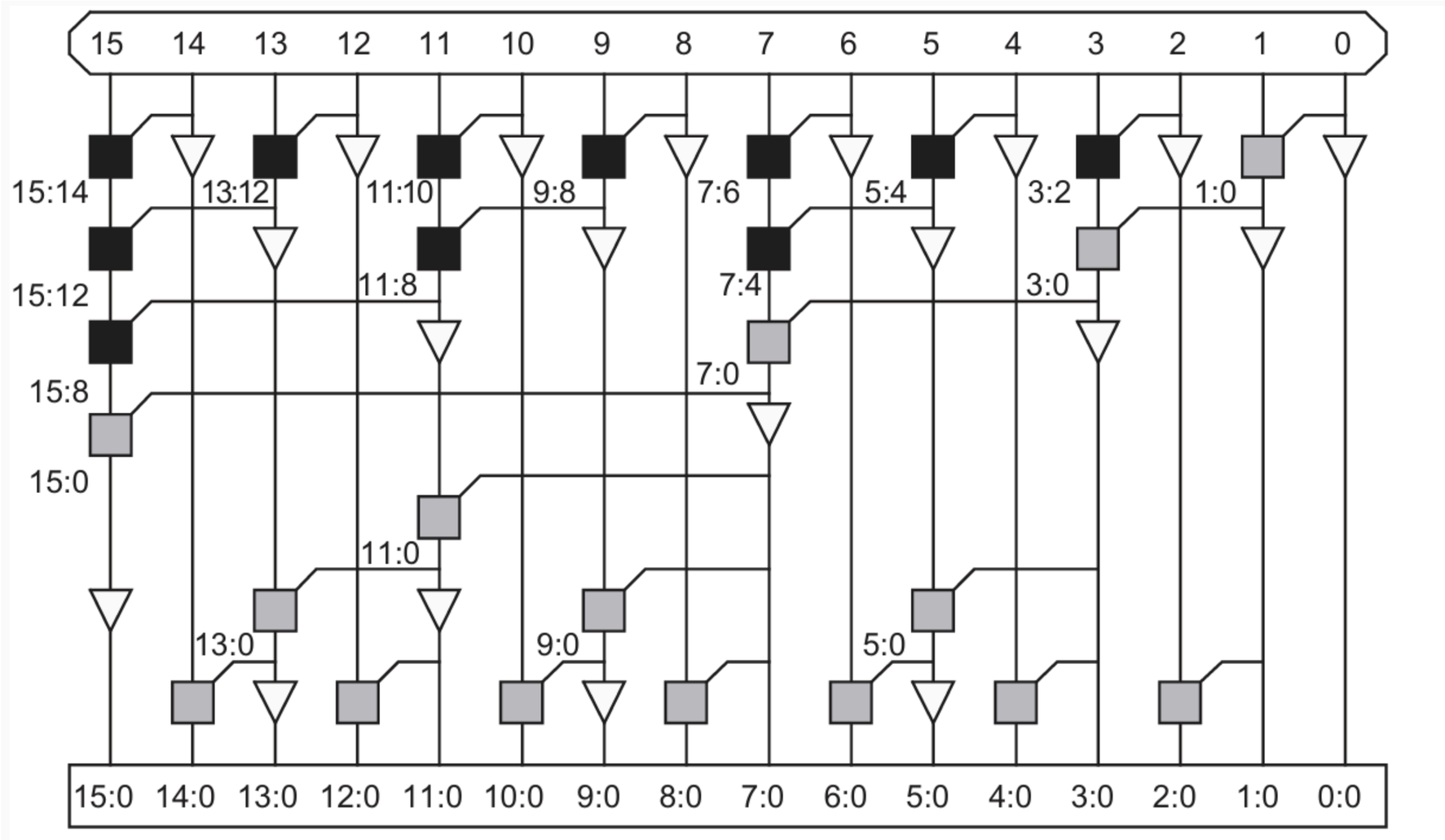
Comparison Units:

- **FPGA**
  - **(Area in LUTS) (Timing in Nanoseconds) (Power in Watt x 10^-4)**
- **ASIC**
  - **(Area in micrometer square) (Timing in Nanoseconds) (Power in Watt x10^-4)**

Prefix Adders Taxonomy



Brent-Kung Adder



Brent-Kung Adder

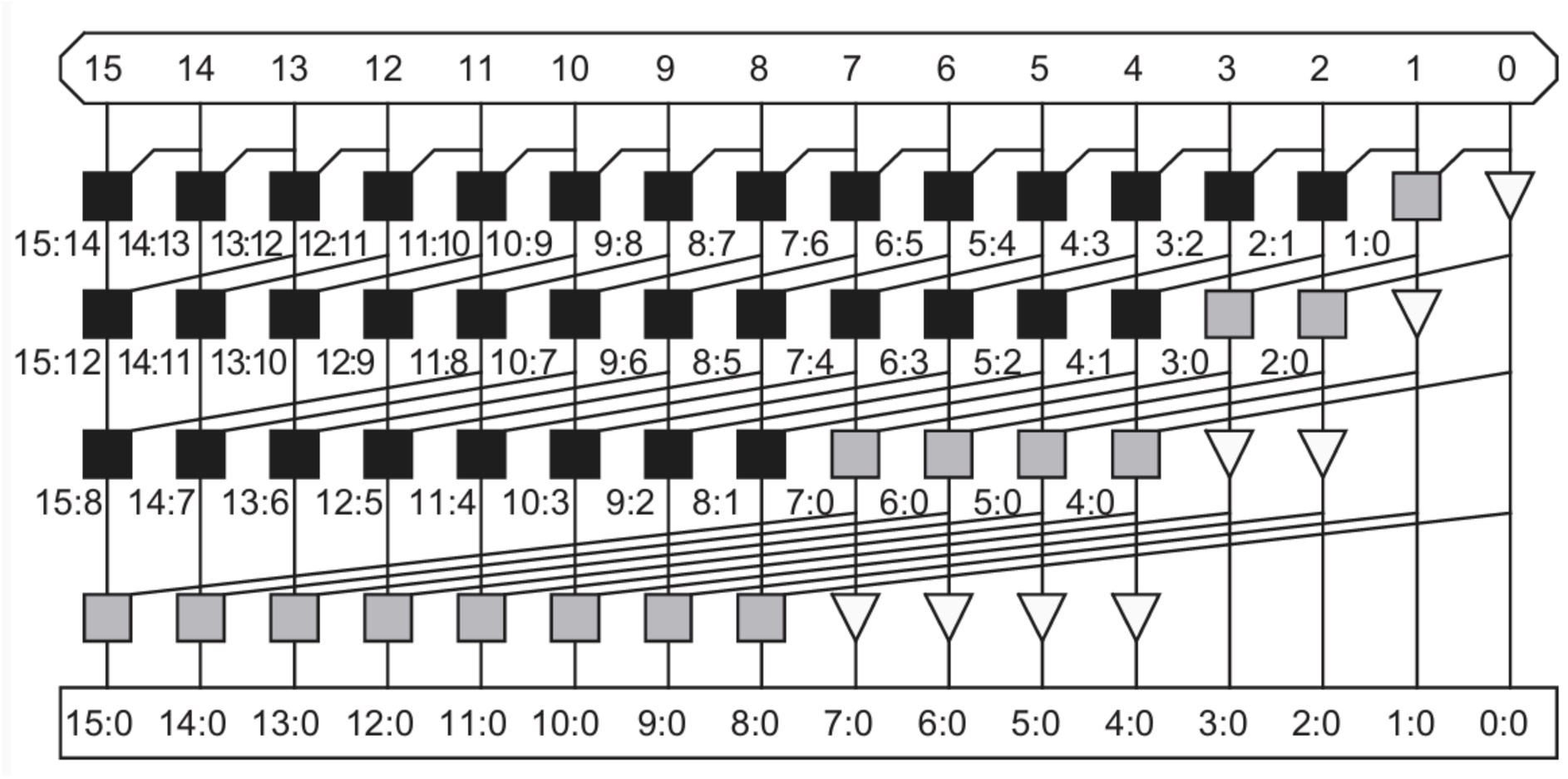
Upper Half Algorithm

```
for(stage = 0; stage < $clog2(N) ; stage = stage + 1) begin
    g_mem[stage + 1] = g_mem[stage];
    p_mem[stage + 1] = p_mem[stage];

  for(i = (2**stage -1); i < N ; i = i + (2**(stage+1))) begin : upper_half
    g_mem[stage+1][i+(1 << stage)] = g_mem[stage][i+(1 << stage)] | (g_mem[stage][i] &  p_mem[stage][i+(1 << stage)]);
    p_mem[stage+1][i+(1 << stage)] = p_mem[stage][i] & p_mem[stage][i+(1 << stage)];

  end
end
```

Lower Half Algorithm

```
 for(stage = $clog2(N)-2 ; stage >= 0 ; stage = stage - 1) begin
     offset = $clog2(N) + $clog2(N/2**2);
     g_mem[offset-stage + 1] = g_mem[offset-stage];
     p_mem[offset-stage + 1] = p_mem[offset-stage];

   for(j = (2**(stage+1)-1); j < N-1 ; j = j + (2**(stage+1))) begin : lower_half
     g_mem[offset-stage+1][j+(1 << stage)] = g_mem[offset-stage][j+(1 << stage)] |
         (g_mem[offset-stage][j] &  p_mem[offset-stage][j+(1 << stage)]);

     p_mem[offset-stage+1][j+(1 << stage)] = p_mem[offset-stage][j] & p_mem[offset-stage][j+(1 << stage)];

   end
 end
```

Kogge-Stone Adder

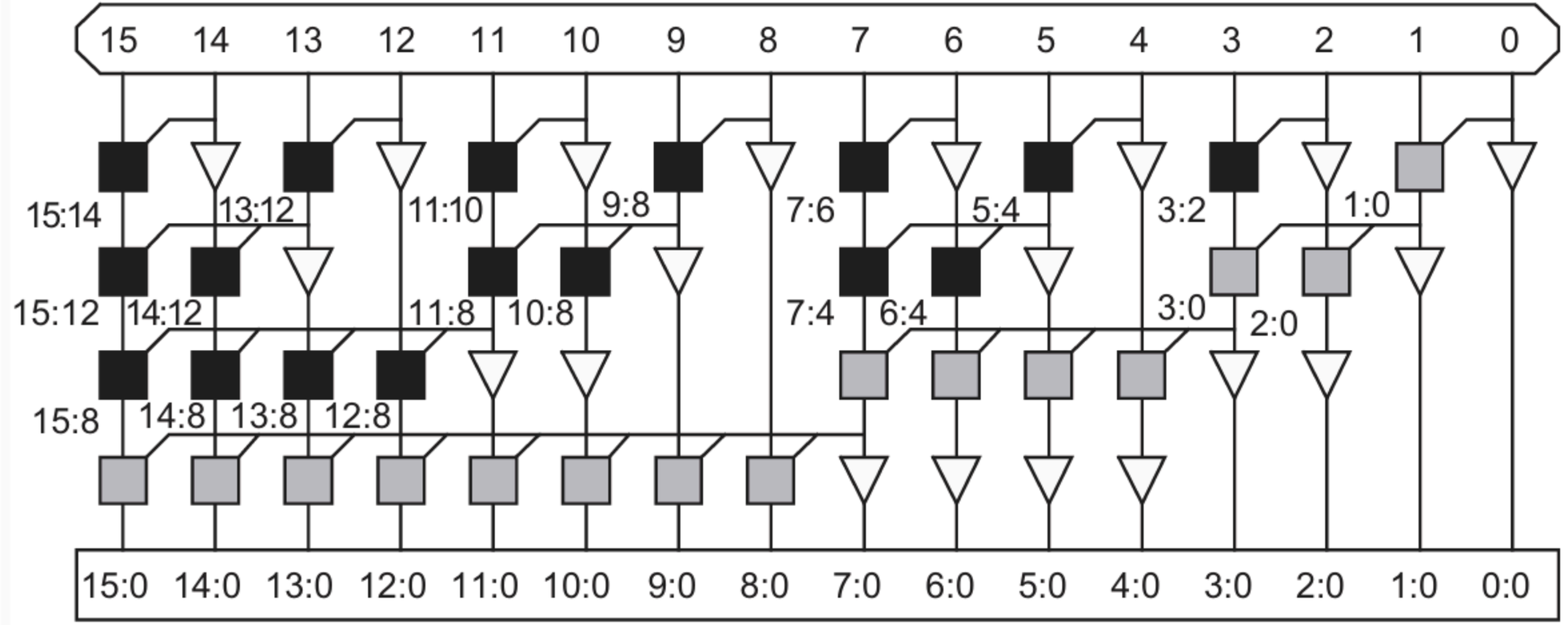


Kogge-Stone Adder

Core algorithm

```
for(stage = 0; stage < $clog2(N); stage = stage + 1) begin

  g_mem[stage + 1] = g_mem[stage];
  p_mem[stage + 1] = p_mem[stage];

  for(i = 0; i < (N - 2**stage) ; i = i + 1) begin

    g_mem[stage + 1][i+(2**stage)] = g_mem[stage][i+(2**stage)] | (g_mem[stage][i] & p_mem[stage][i+(2**stage)]);
    p_mem[stage + 1][i+(2**stage)] = p_mem[stage][i] & p_mem[stage][i+(2**stage)];

  end
end
```

Sklansky Adder

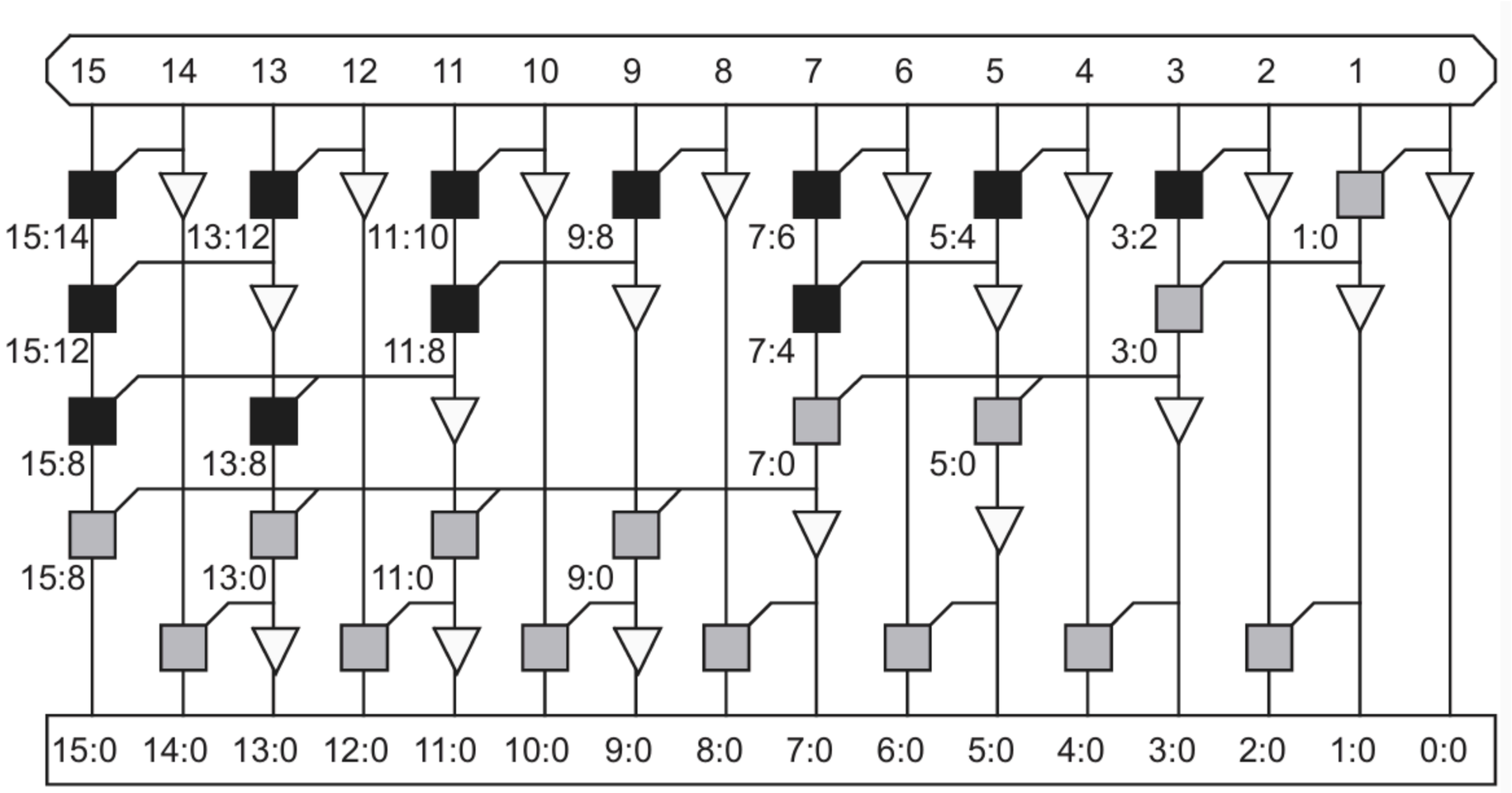


Sklansky Adder

Core algorithm

```
for(stage = 0; stage < $clog2(N) ; stage = stage + 1) begin    // number of levels
    g_mem[stage + 1] = g_mem[stage];
    p_mem[stage + 1] = p_mem[stage];

  for(k = 0 ; k < N/2 ; k = k + 1) begin                       // number of carry operators
    for(i = (2**stage -1); i < N ; i = i + (2**(stage+1))) begin
      for(j = 0 ; j < 2**stage ; j = j + 1) begin               //number of repeatitions of first operand
        g_mem[stage + 1][i+1+j] = g_mem[stage][i+1+j] | (g_mem[stage][i] & p_mem[stage][i+1+j]);
        p_mem[stage + 1][i+1+j] = p_mem[stage][i] & p_mem[stage][i+1+j];
      end
    end
  end
end
```

Ladner-Fischer Adder



Ladner-Fischer Adder

First 2 stages always are brent kung

```
  if(stage == 0 || stage == 1) begin    // brent kung upper half algorithm
    for(i = (2**stage -1); i < N ; i = i + (2**(stage+1))) begin : upper_half

      g_mem[stage+1][i+(1 << stage)] = g_mem[stage][i+(1 << stage)] | (g_mem[stage][i] &  p_mem[stage][i+(1 << stage)]);
      p_mem[stage+1][i+(1 << stage)] = p_mem[stage][i] & p_mem[stage][i+(1 << stage)];

    end
  end
```

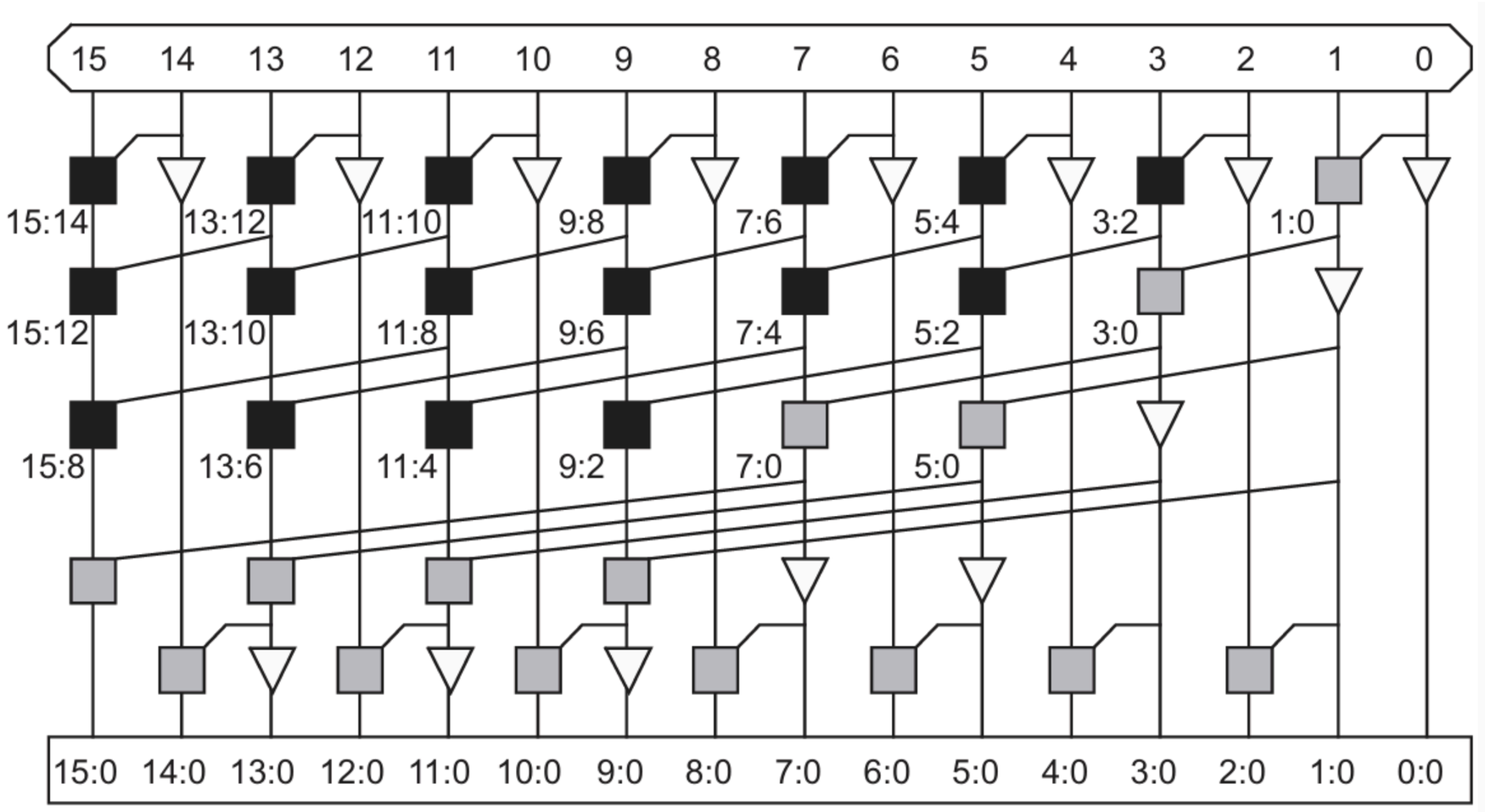
The rest is sklansky

```
if(stage >= 2 && stage <= $clog2(N)-1) begin  // sklansky algoritm
   for(k = 0 ; k < N/4; k = k + 1) begin                    // number of carry operators (squares in each line)
    for(i = (2**stage -1); i < N ; i = i + (2**(stage+1))) begin
      for(j = 0 ; j < 2**(stage-1) ; j = j + 1) begin                //number of repeatitions of first operand
        g_mem[stage + 1][i+2+2*j] = g_mem[stage][i+2+2*j] | (g_mem[stage][i] & p_mem[stage][i+2+2*j]);
        p_mem[stage + 1][i+2+2*j] = p_mem[stage][i] & p_mem[stage][i+2+2*j];
      end
    end
  end
end
```

Last stage is always brent kung

```
 if(stage == $clog2(N)) begin // brent kung lower half algorithm
   for(k = (2**0); k < N-1 ; k = k + (2**(0+1))) begin : lower_half

     g_mem[stage+1][k+(1 << 0)] = g_mem[stage][k+(1 << 0)] | (g_mem[stage][k] &  p_mem[stage][k+(1 << 0)]);
     p_mem[stage+1][k+(1 << 0)] = p_mem[stage][k] & p_mem[stage][k+(1 << 0)];

   end
 end
```

Han-Carlson Adder



Han-Carlson Adder

First stage always is brent kung

```
if(stage == 0) begin  // brent kung upper half algorithm
  for(i = (2**stage -1); i < N ; i = i + (2**(stage+1))) begin : upper_half

    g_mem[stage+1][i+(1 << stage)] = g_mem[stage][i+(1 << stage)] | (g_mem[stage][i] &  p_mem[stage][i+(1 << stage)]);
    p_mem[stage+1][i+(1 << stage)] = p_mem[stage][i] & p_mem[stage][i+(1 << stage)];

  end
end
```

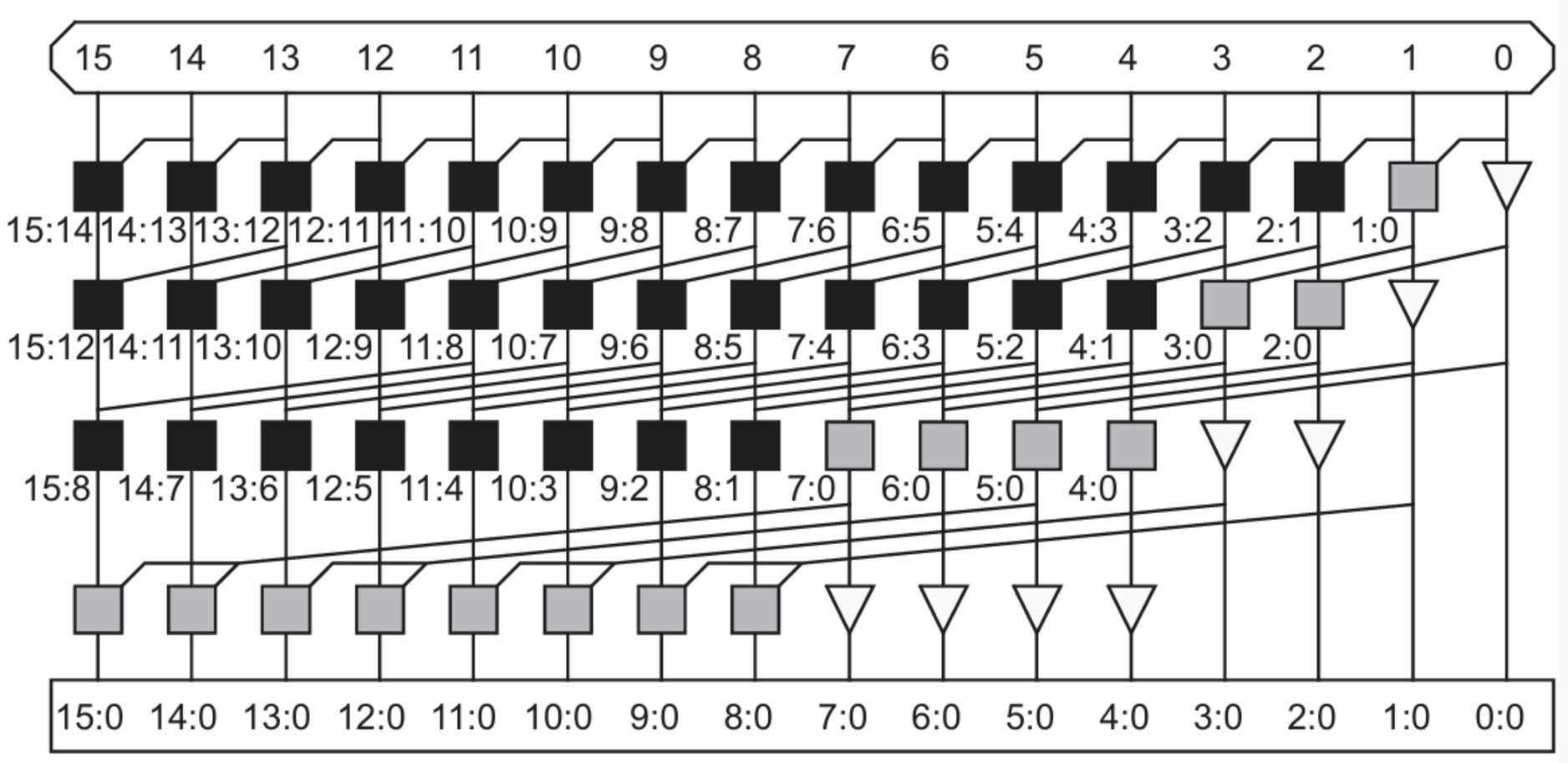
The rest is kogge stone

```
if(stage >= 1 && stage <= $clog2(N)-1) begin  // kogge stone algorithm
  for(j = 1; j < (N - 2**stage) ; j = j + 2) begin

    g_mem[stage + 1][j+(2**stage)] = g_mem[stage][j+(2**stage)] | (g_mem[stage][j] & p_mem[stage][j+(2**stage)]);
    p_mem[stage + 1][j+(2**stage)] = p_mem[stage][j] & p_mem[stage][j+(2**stage)];

  end
end
```

Last stage is always brent kung

```
if(stage == $clog2(N)) begin // brent kung lower half algorithm
  for(k = (2**0); k < N-1 ; k = k + (2**(0+1))) begin : lower_half

    g_mem[stage+1][k+(1 << 0)] = g_mem[stage][k+(1 << 0)] | (g_mem[stage][k] &  p_mem[stage][k+(1 << 0)]);
    p_mem[stage+1][k+(1 << 0)] = p_mem[stage][k] & p_mem[stage][k+(1 << 0)];

  end
end
```

Knowles Adder



Knowles Adder

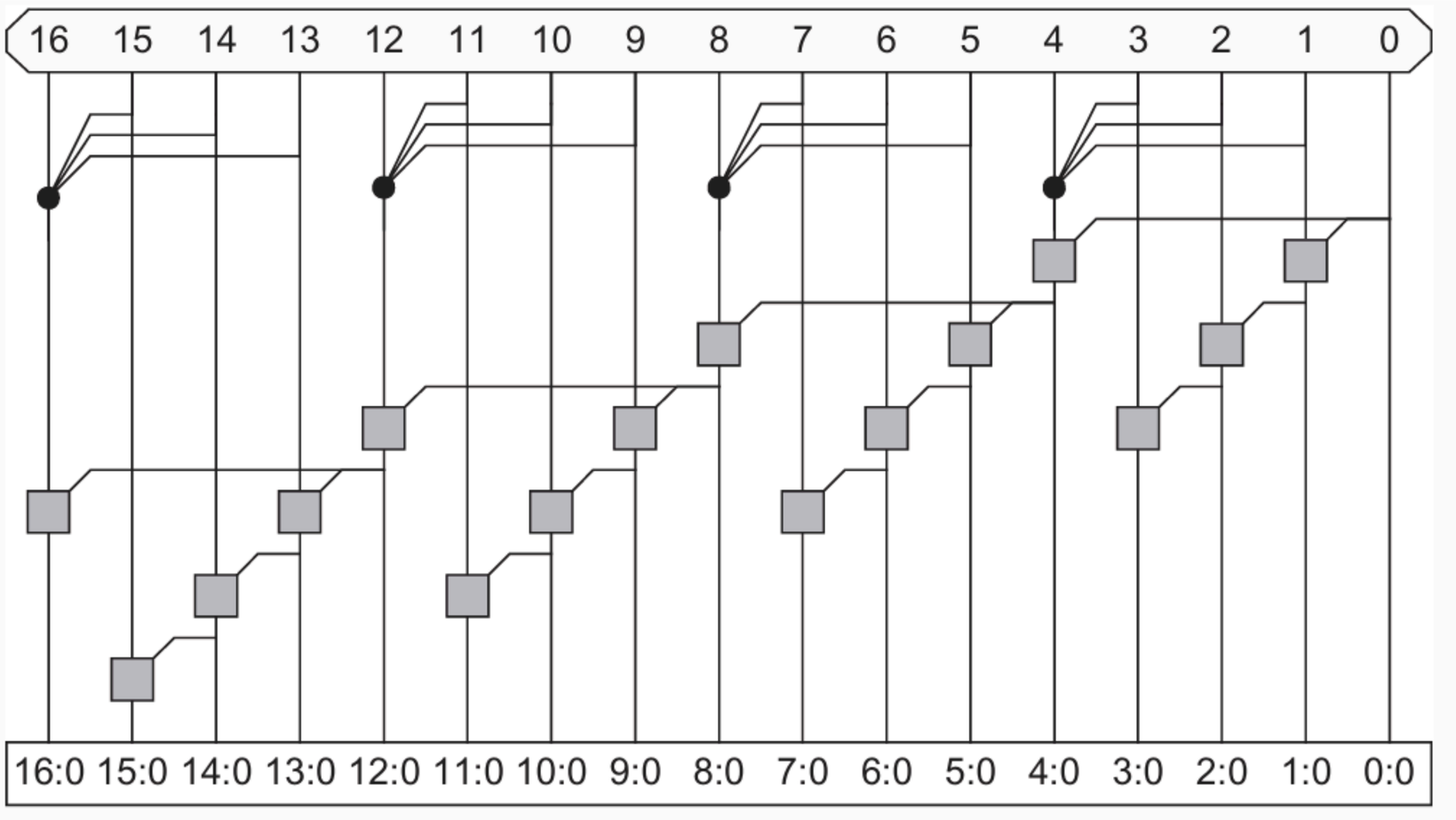
all stages from start till the the one before the last one are same as kogge stone adder

```
if(stage >= 0 && stage <= $clog2(N)-1) begin   // normal kogge stone algorithm
  for(i = 0; i < (N - 2**stage) ; i = i + 1) begin

    g_mem[stage + 1][i+(2**stage)] = g_mem[stage][i+(2**stage)] | (g_mem[stage][i] & p_mem[stage][i+(2**stage)]);
    p_mem[stage + 1][i+(2**stage)] = p_mem[stage][i] & p_mem[stage][i+(2**stage)];

  end
end
```

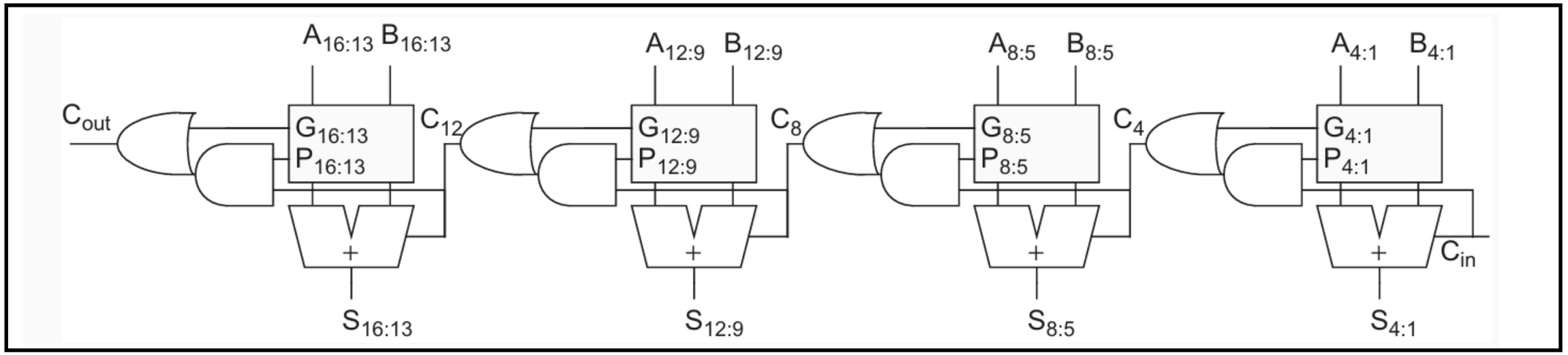
the last stage is kogge stone also but modified a little bit to be like sklansky this help in reduces number of wires to the half always

```
if(stage == $clog2(N)) begin   // kogge stone modified to be like sklansky in the final stage
  for(i = 1; i < (N - 2**stage) ; i = i + 2) begin
    for(j = 0 ; j < 2 ; j = j + 1) begin // sklansky repeatition algorithm
      g_mem[stage + 1][i+(2**stage)-1+j] = g_mem[stage][i+(2**stage)-1+j] | (g_mem[stage][i] & p_mem[stage][i+(2**stage)-1+j]);
      p_mem[stage + 1][i+(2**stage)-1+j] = p_mem[stage][i] & p_mem[stage][i+(2**stage)-1+j];
    end
  end
end
```

Carry Look-Ahead Adder



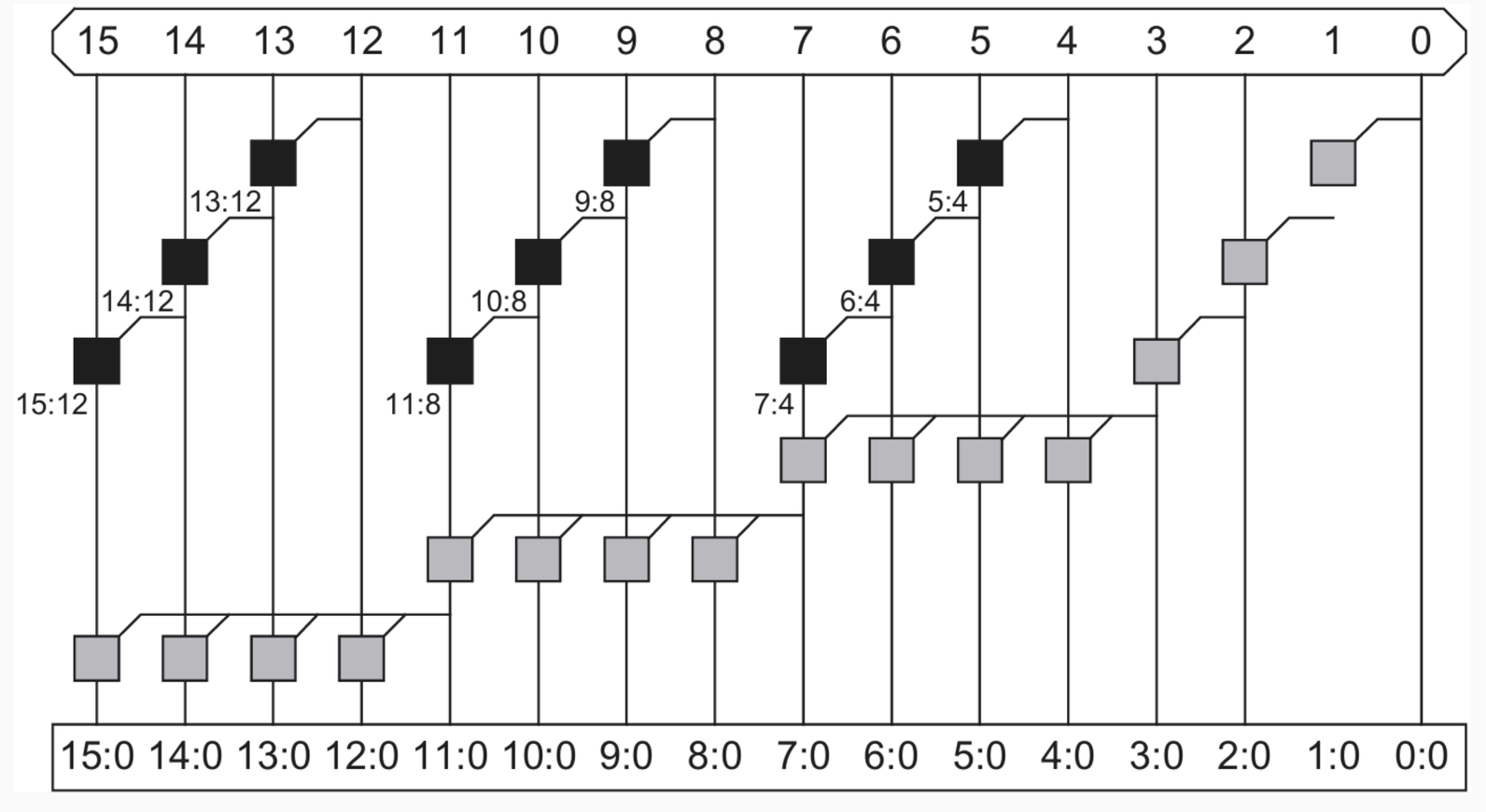
Carry Look-Ahead Adder



Carry Look-Ahead Adder

```
localparam NUM_BLOCKS = N / 4;  // Number of 4-bit CLA blocks
wire [NUM_BLOCKS:0] carry_chain;

assign carry_chain[0] = cin;  // Global carry-in

// Generate N/4 instances of 4-bit CLA
genvar i;
generate
  for (i = 0; i < NUM_BLOCKS; i = i + 1) begin : cla_block
    cla_4bit cla_inst (
      .a    (a[i*4 +: 4]),    // 4-bit slice of 'a'
      .b    (b[i*4 +: 4]),    // 4-bit slice of 'b'
      .cin  (carry_chain[i]), // Carry-in from previous block
      .sum  (sum[i*4 +: 4]),  // 4-bit slice of 'sum'
      .cout (carry_chain[i+1]) // Carry-out to next block
    );
  end
endgenerate
```

Carry Increment Adder

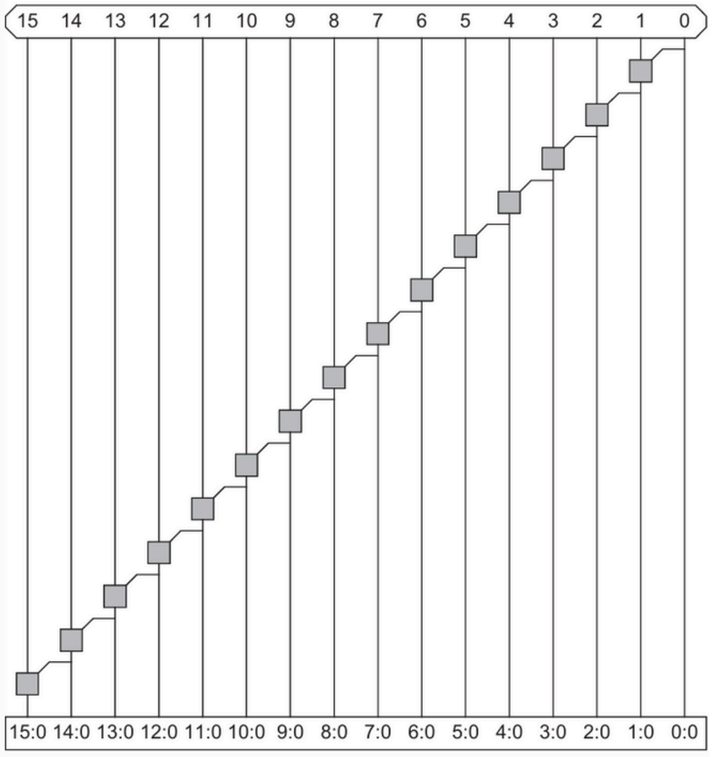


Carry Increment Adder

```
for(stage = 0; stage < 3+(N/4 -1); stage = stage + 1) begin

  g_mem[stage + 1] = g_mem[stage];
  p_mem[stage + 1] = p_mem[stage];

  if(stage >= 0 && stage <= 2) begin // upper half algorithm
    for(i = stage; i < N ; i = i + 4) begin

      g_mem[stage + 1][i+1] = g_mem[stage][i+1] | (g_mem[stage][i] & p_mem[stage][i+1]);
      p_mem[stage + 1][i+1] = p_mem[stage][i] & p_mem[stage][i+1];

    end
  end
```

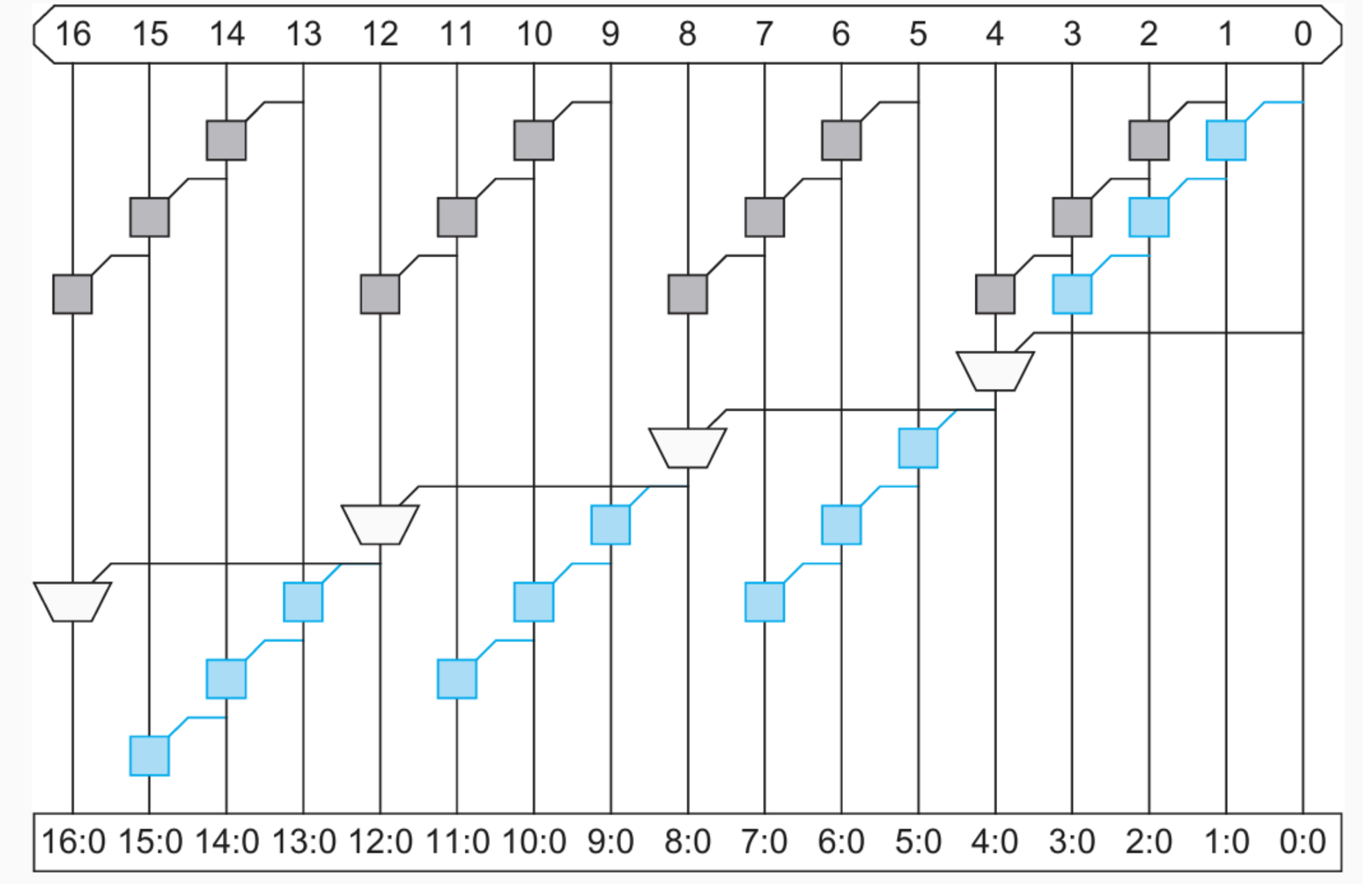
```
if(stage >= 3 && stage <= 3+(N/4 -2)) begin // lower half algorithm
  for(i = 3 + 4*(stage - 3) ; i < 4 + 4*(stage - 3) ; i = i + 1) begin
    for(j = 0 ; j < BLOCK_SIZE ; j = j + 1) begin

      g_mem[stage + 1][i+j+1] = g_mem[stage][i+j+1] | (g_mem[stage][i] & p_mem[stage][i+j+1]);
      p_mem[stage + 1][i+j+1] = p_mem[stage][i] & p_mem[stage][i+j+1];

     end
  end
end
```

Carry Ripple Adder

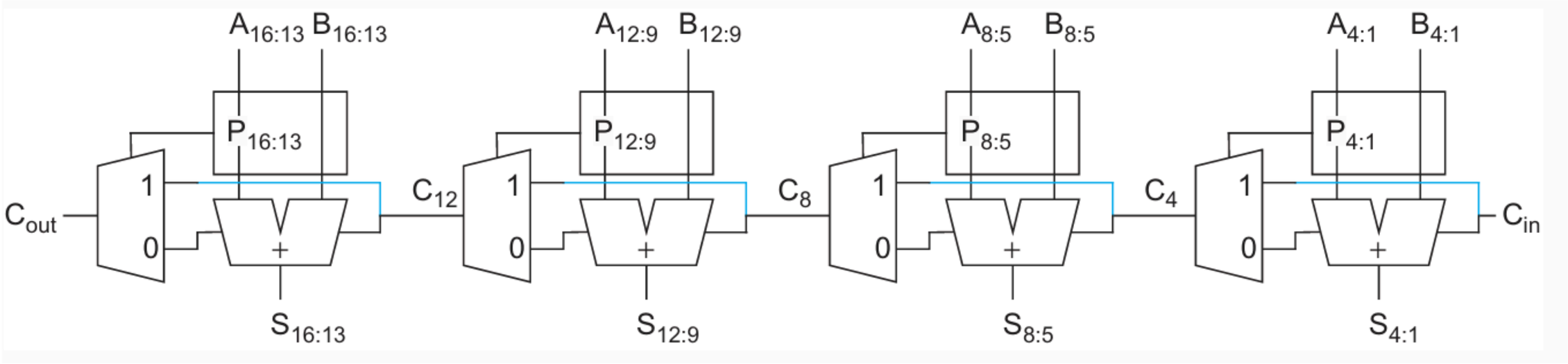



```
  for(stage = 0; stage < N-1; stage = stage + 1) begin

    g_mem[stage + 1] = g_mem[stage];
    p_mem[stage + 1] = p_mem[stage];

    g_mem[stage + 1][stage + 1] = g_mem[stage][stage + 1] | (g_mem[stage][stage] & p_mem[stage][stage + 1]);
    p_mem[stage + 1][i+(2**stage)] = p_mem[stage][i] & p_mem[stage][i+(2**stage)];

  end
end
```

Carry Skip Adder



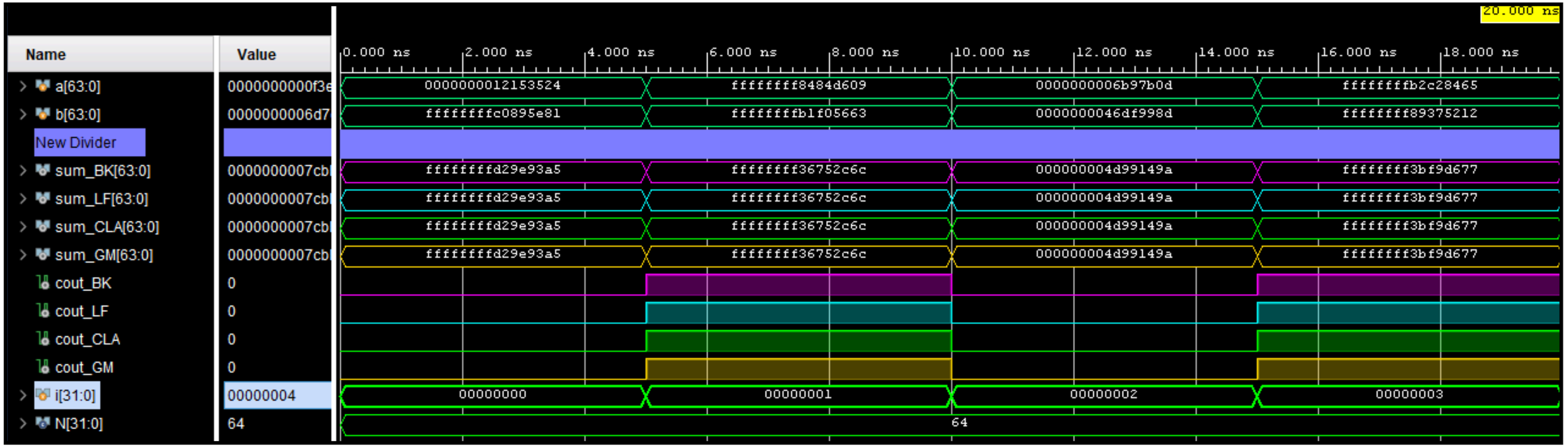
Carry Skip Adder



Carry Skip Adder

```
localparam NUM_BLOCKS = N / 4;  // Number of 4-bit blocks
wire [NUM_BLOCKS:0] carry_chain;

assign carry_chain[0] = cin;    // Global carry-in

// Instantiate N/4 carry-skip blocks
genvar i;
generate
  for (i = 0; i < NUM_BLOCKS; i = i + 1) begin : block
    carry_skip_4bit cs4 (
      .a    (a[i*4 +: 4]),      // 4-bit slice of 'a'
      .b    (b[i*4 +: 4]),      // 4-bit slice of 'b'
      .cin  (carry_chain[i]),   // Carry-in from previous block
      .sum  (sum[i*4 +: 4]),    // 4-bit slice of 'sum'
      .cout (carry_chain[i+1])  // Carry-out to next block
    );
  end
endgenerate
```

Simulation of 3 adders

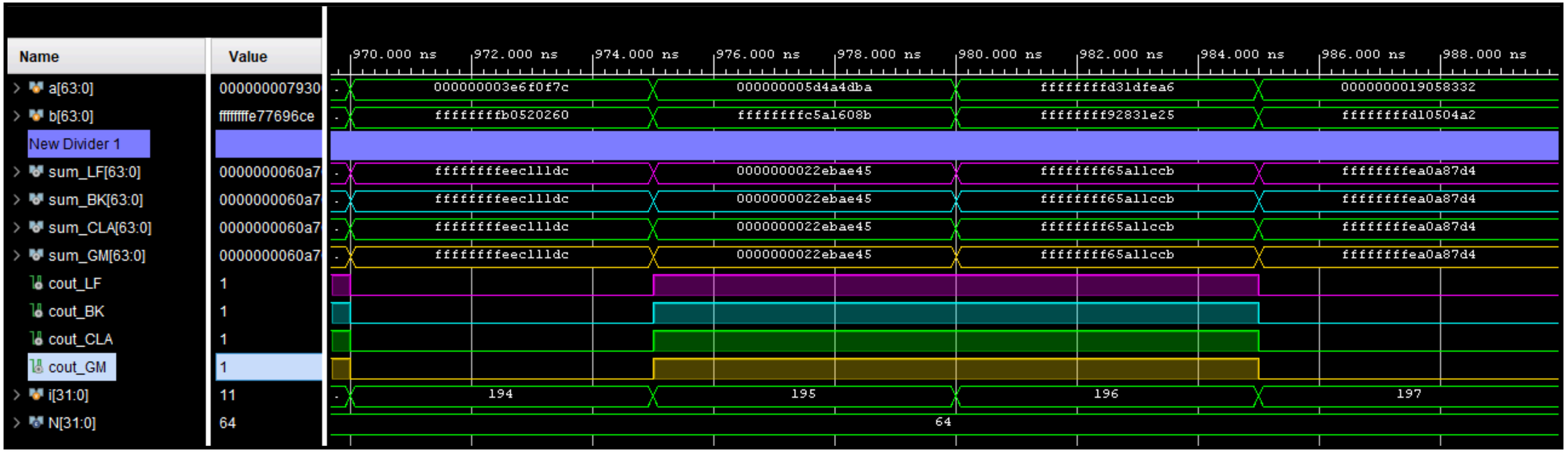


sum_BK >> Brent Kung **sum_LF >> Ladner Fischer**

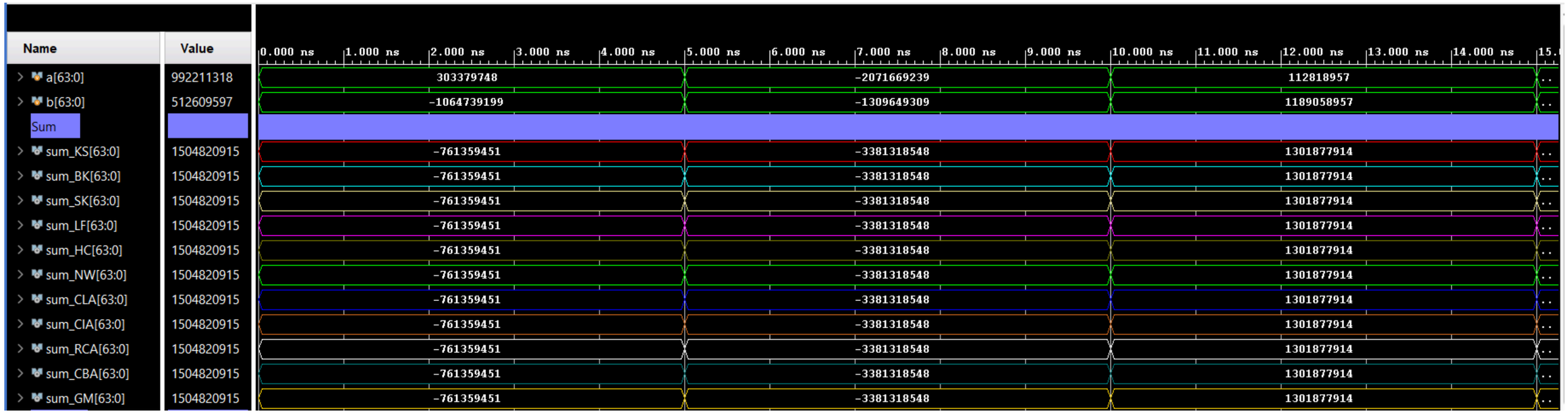
sum_CLA >> Carry look ahead

sum_GM >> Golden model (reference)

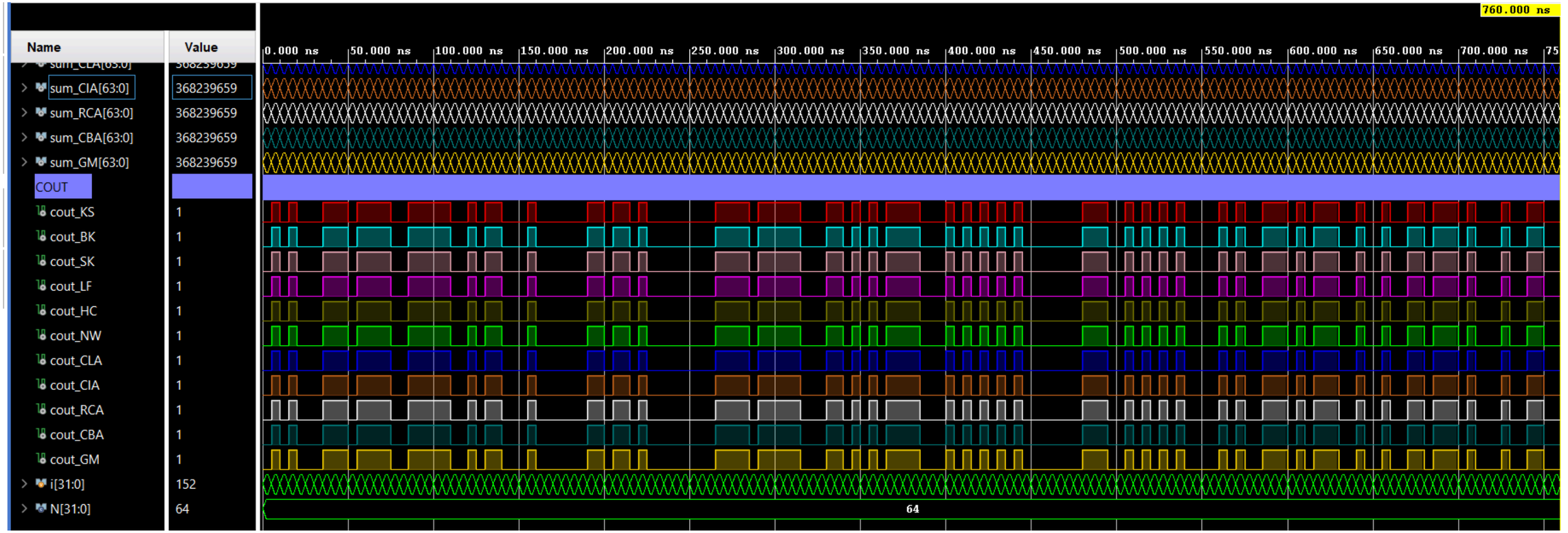
Further Testing



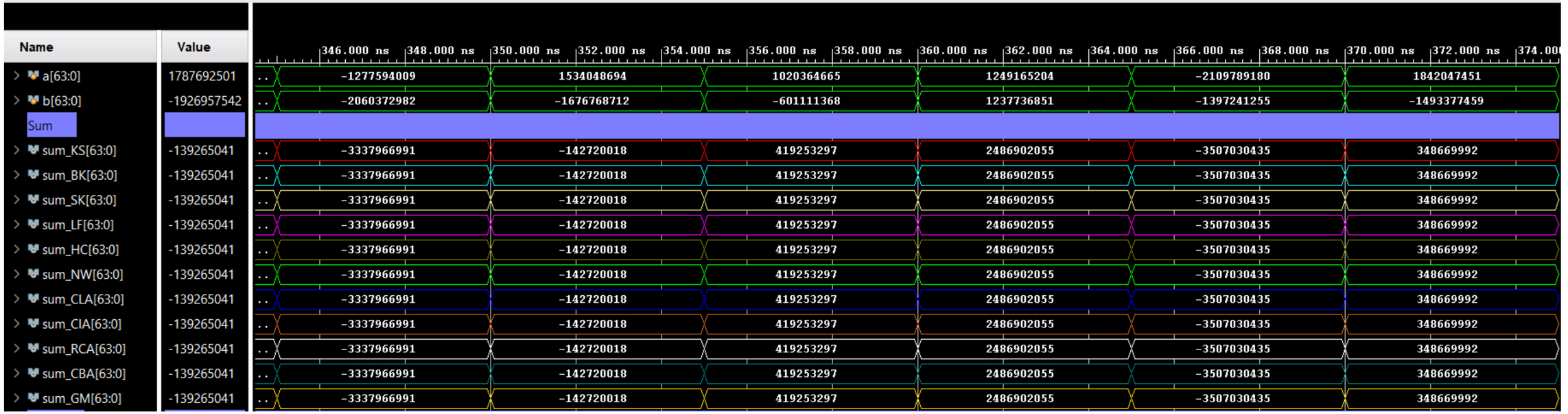
Further Testing (all adders)



Further Testing

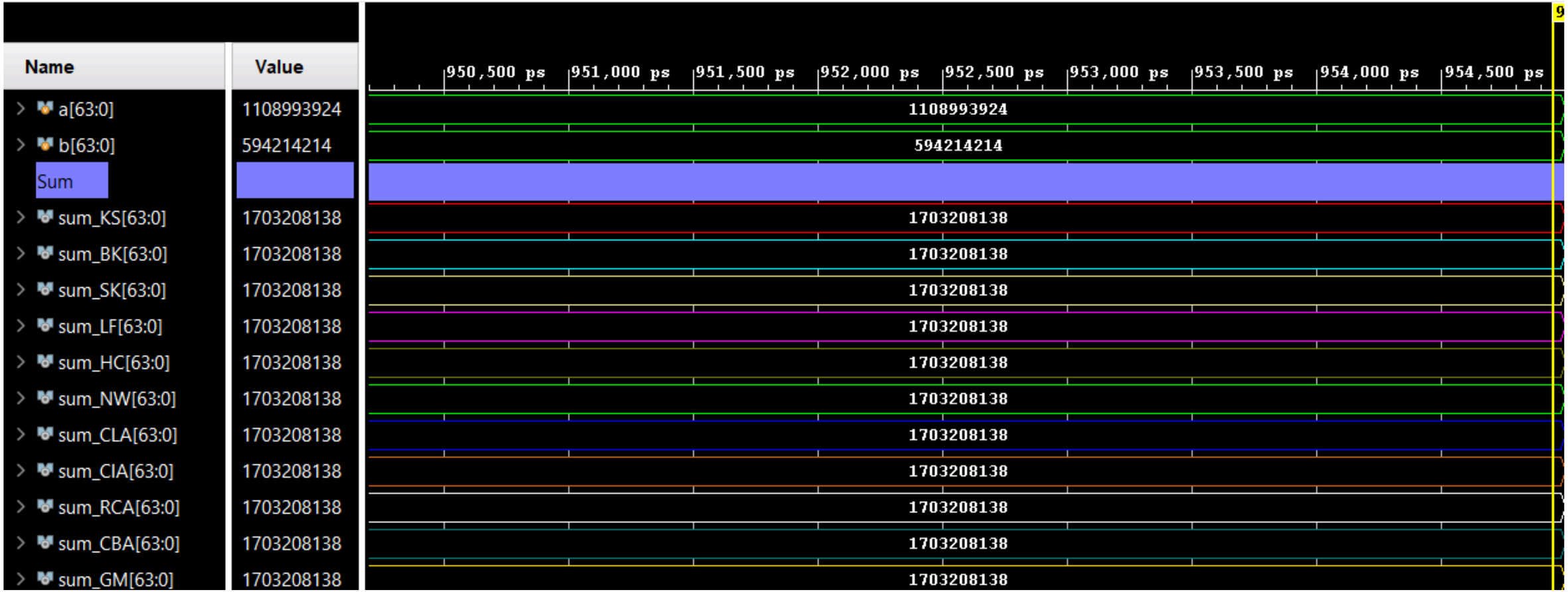


Further Testing



Name	Value	346.000 ns	348.000 ns	350.000 ns	352.000 ns	354.000 ns	356.000 ns	358.000 ns	360.000 ns	362.000 ns	364.000 ns	366.000 ns	368.000 ns	370.000 ns	372.000 ns	374.00
a[63:0]	1787692501	..	-1277594009		1534048694		1020364665		1249165204		-2109789180		1842047451			
b[63:0]	-1926957542	..	-2060372982		-1676768712		-601111368		1237736851		-1397241255		-1493377459			
Sum																
sum_KS[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_BK[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_SK[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_LF[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_HC[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_NW[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_CLA[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_CIA[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_RCA[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_CBA[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			
sum_GM[63:0]	-139265041	..	-3337966991		-142720018		419253297		2486902055		-3507030435		348669992			

Further Testing

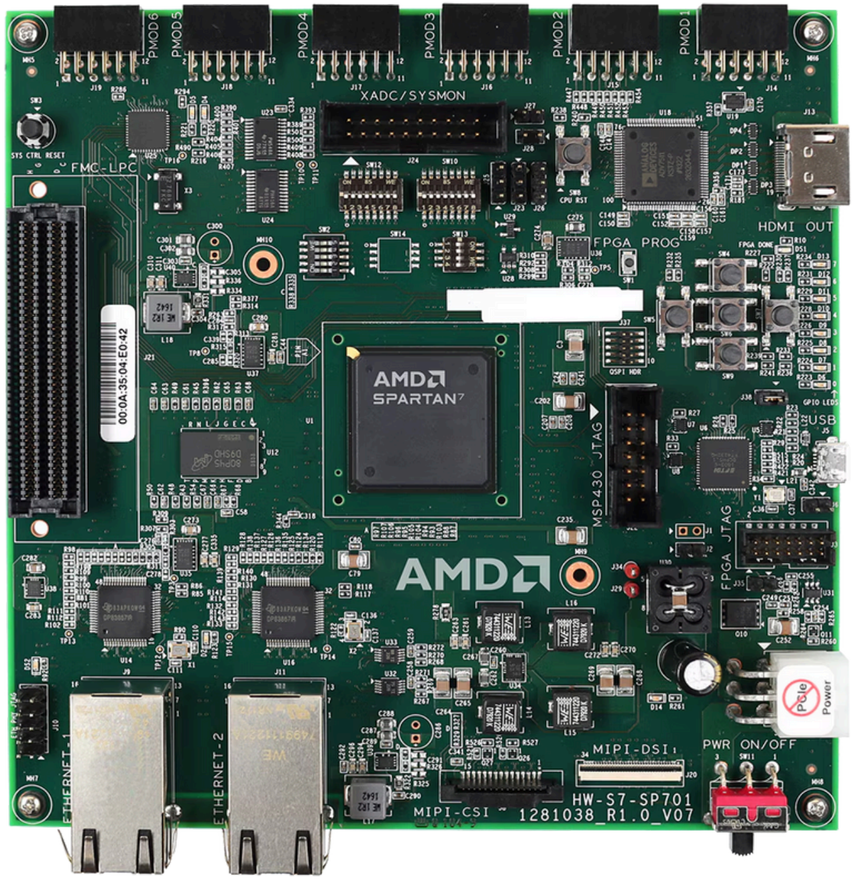


USED FPGA Device

The FPGA used for synthesis and implementation is Spartan 7 XC7S100

Device number in vivado:

xc7s100fgga676-1Q



USED ASIC TECH.

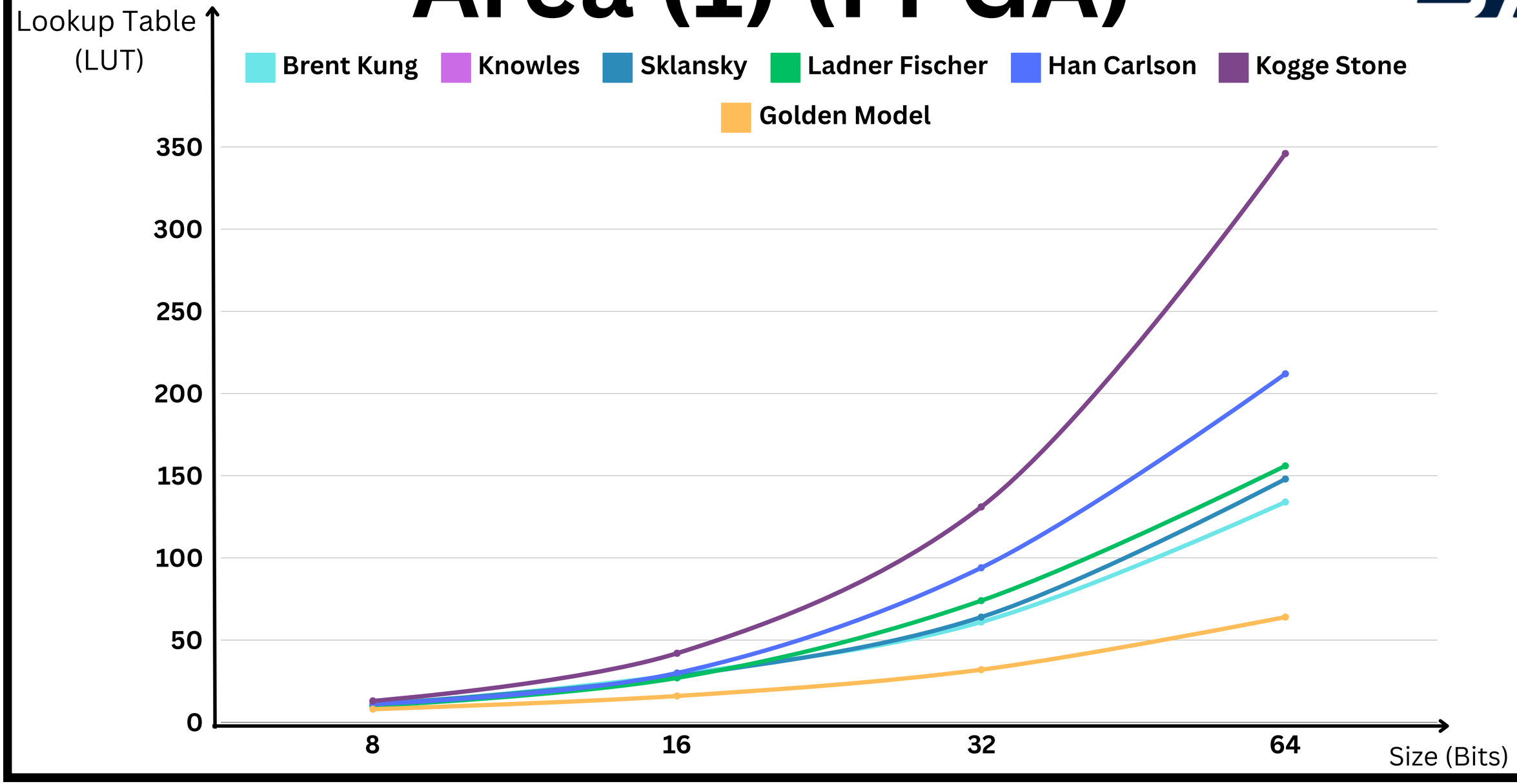
The technology node used for the ASIC implementation and comparison of adders is SKY130A using OpenLane Flow

skywater
TECHNOLOGY

Area (1) (FPGA)

Adder / Size in bits	8	16	32	64
Brent Kung	10	29	61	134
Kogge Stone	13	42	131	346
Sklansky	11	28	64	148
Ladner Fischer	10	27	74	156
Han Carlson	11	30	94	212
Knowles	13	42	131	346
Golden Model	8	16	32	64

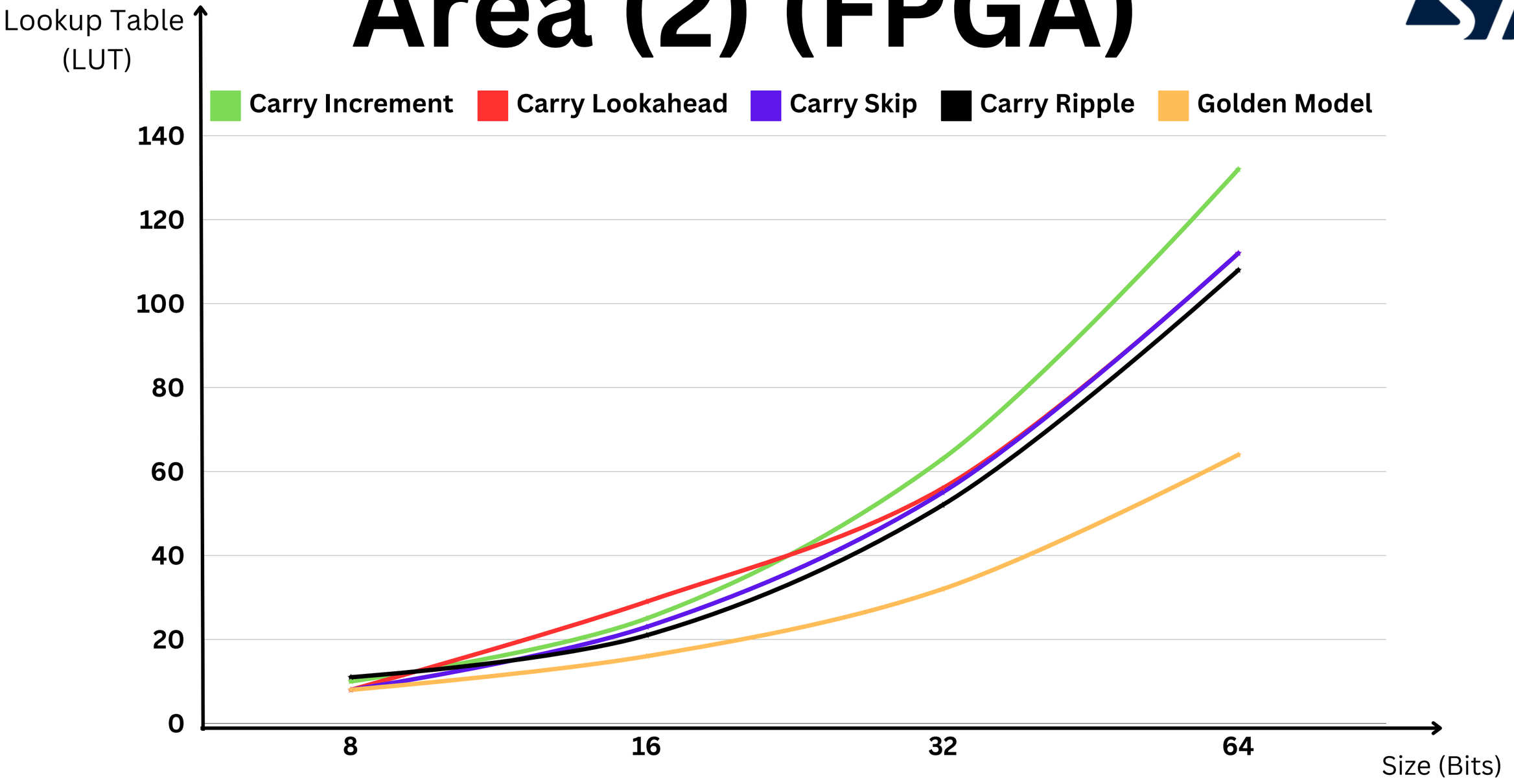
Area (1) (FPGA)


Lookup Table (LUT)
Brent Kung
Knowles
Sklansky
Ladner Fischer
Han Carlson
Kogge Stone
Golden Model
350
300
250
200
150
100
50
0
8
16
32
64
Size (Bits)

Area (2) (FPGA)

Adder / Size in bits	8	16	32	64
Carry Increment	10	25	63	132
Carry Lookahead	8	29	56	112
Carry Skip	8	23	55	112
Carry Ripple	11	21	52	108
Golden Model	8	16	32	64

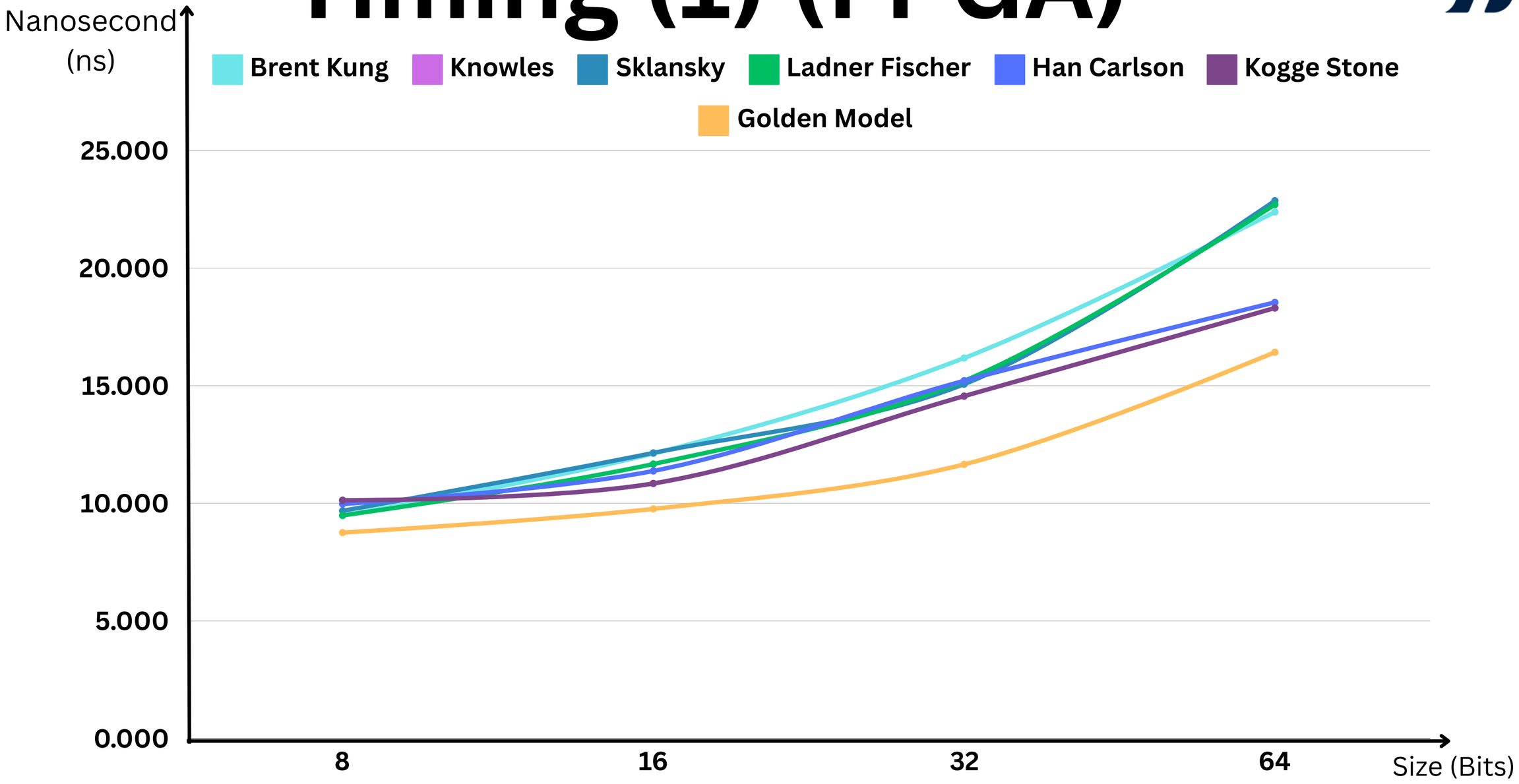
Area (2) (FPGA)


Lookup Table (LUT)
Carry Increment
Carry Lookahead
Carry Skip
Carry Ripple
Golden Model
140
120
100
80
60
40
20
0
8
16
32
64
Size (Bits)

Timing (1) (FPGA)

Adder / Size in bits	8	16	32	64
Brent Kung	9.485	12.118	16.178	22.389
Kogge Stone	10.130	10.845	14.559	18.306
Sklansky	9.682	12.144	15.066	22.867
Ladner Fischer	9.485	11.671	15.211	22.704
Han Carlson	9.972	11.377	15.211	18.545
Knowles	10.130	10.845	14.559	18.306
Golden Model	8.756	9.764	11.656	16.425

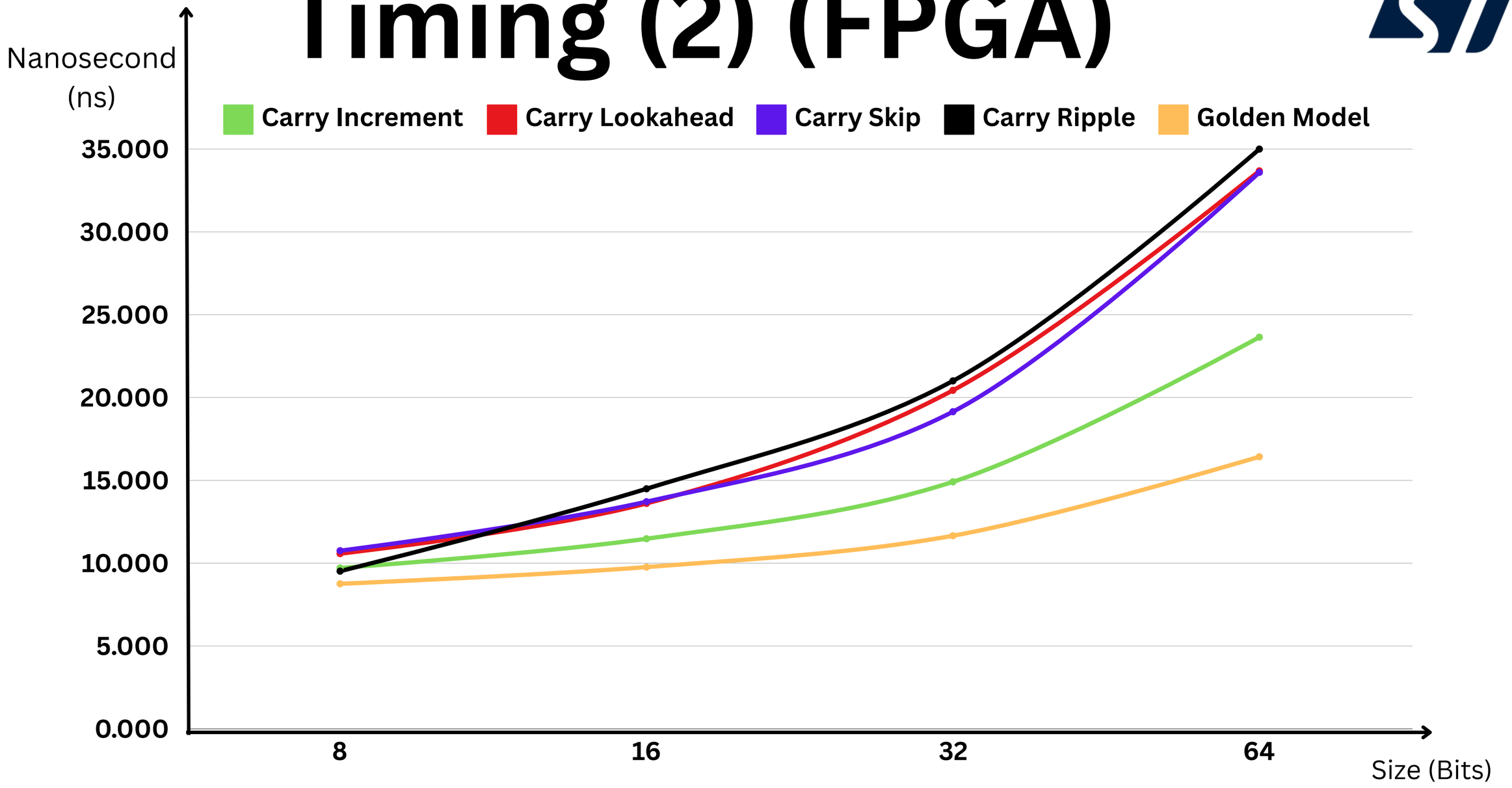
Timing (1) (FPGA)



Timing (2) (FPGA)

Adder / Size in bits	8	16	32	64
Carry Increment	9.709	11.475	14.911	23.643
Carry Lookahead	10.576	13.609	20.435	33.685
Carry Skip	10.752	13.711	19.145	33.593
Carry Ripple	9.519	14.492	21.004	35
Golden Model	8.756	9.764	11.656	16.425

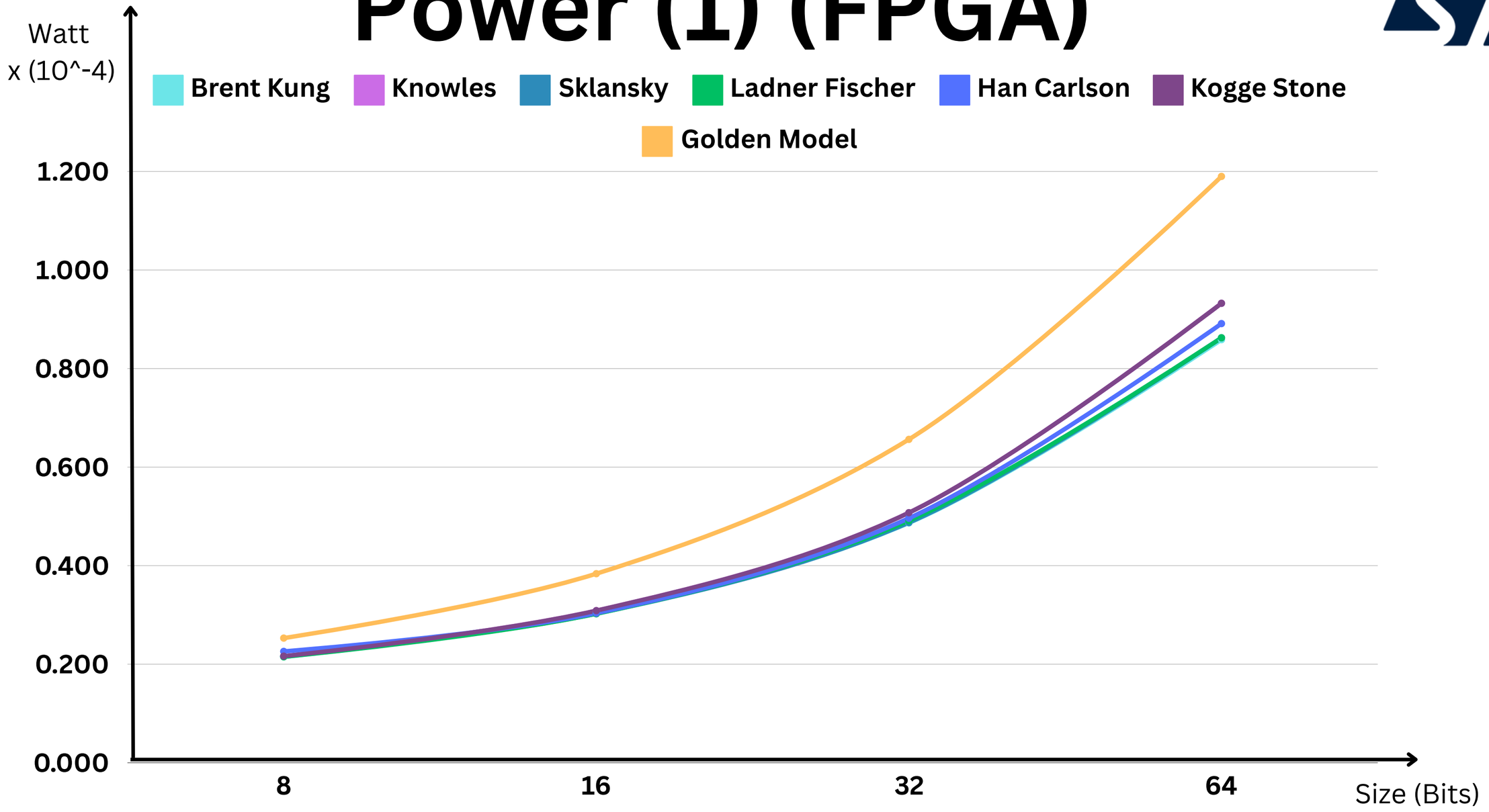
Timing (2) (FPGA)



Power (1) (FPGA)

Adder / Size in bits	8	16	32	64
Brent Kung	2.15 E-4	3.037 E-4	4.887 E-4	8.587 E-4
Kogge Stone	2.163 E-4	3.087 E-4	5.075 E-4	9.325 E-4
Sklansky	2.163 E-4	3.025 E-4	4.875 E-4	8.625 E-4
Ladner Fischer	2.15 E-4	3.025 E-4	4.912 E-4	8.625 E-4
Han Carlson	2.26 E-4	3.037 E-4	4.963 E-4	8.913 E-4
Knowles	2.163 E-4	3.087 E-4	5.075 E-4	9.325 E-4
Golden Model	2.53 E-4	3.837 E-4	6.563 E-4	1.19 E-3

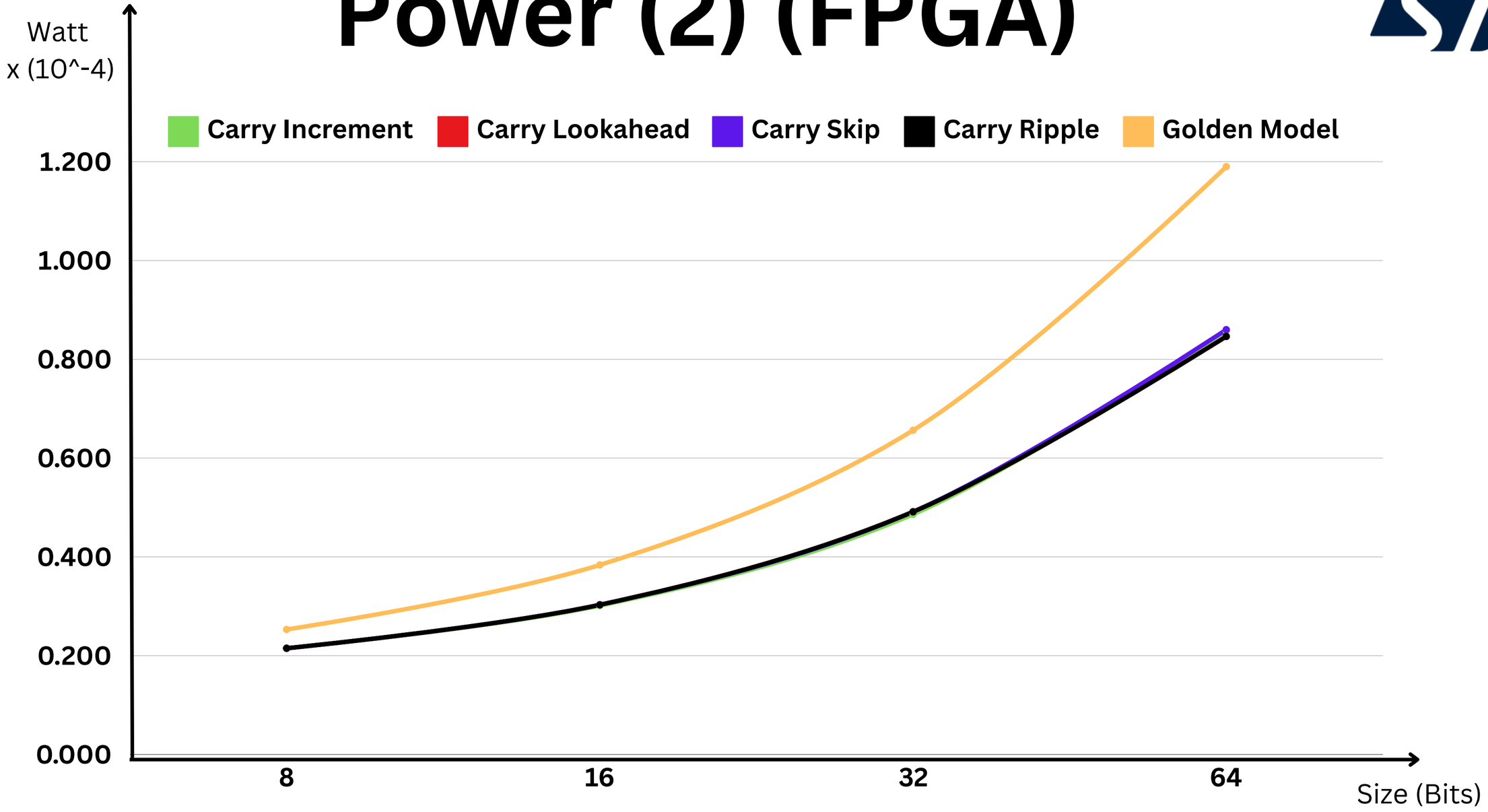
Power (1) (FPGA)


Watt
x (10^-4)
Brent Kung
Knowles
Sklansky
Ladner Fischer
Han Carlson
Kogge Stone
Golden Model
1.200
1.000
0.800
0.600
0.400
0.200
0.000
8
16
32
64
Size (Bits)

Power (2) (FPGA)

Adder / Size in bits	8	16	32	64
Carry Increment	2.15 E-4	3.013 E-4	4.8625 E-4	8.575 E-4
Carry Lookahead	2.15 E-4	3.03 E-4	4.913 E-4	8.463 E-4
Carry Skip	2.15 E-4	3.03 E-4	4.913 E-4	8.6 E-4
Carry Ripple	2.15 E-4	3.03 E-4	4.913 E-4	8.463 E-4
Golden Model	2.53 E-4	3.837 E-4	6.563 E-4	1.19 E-3

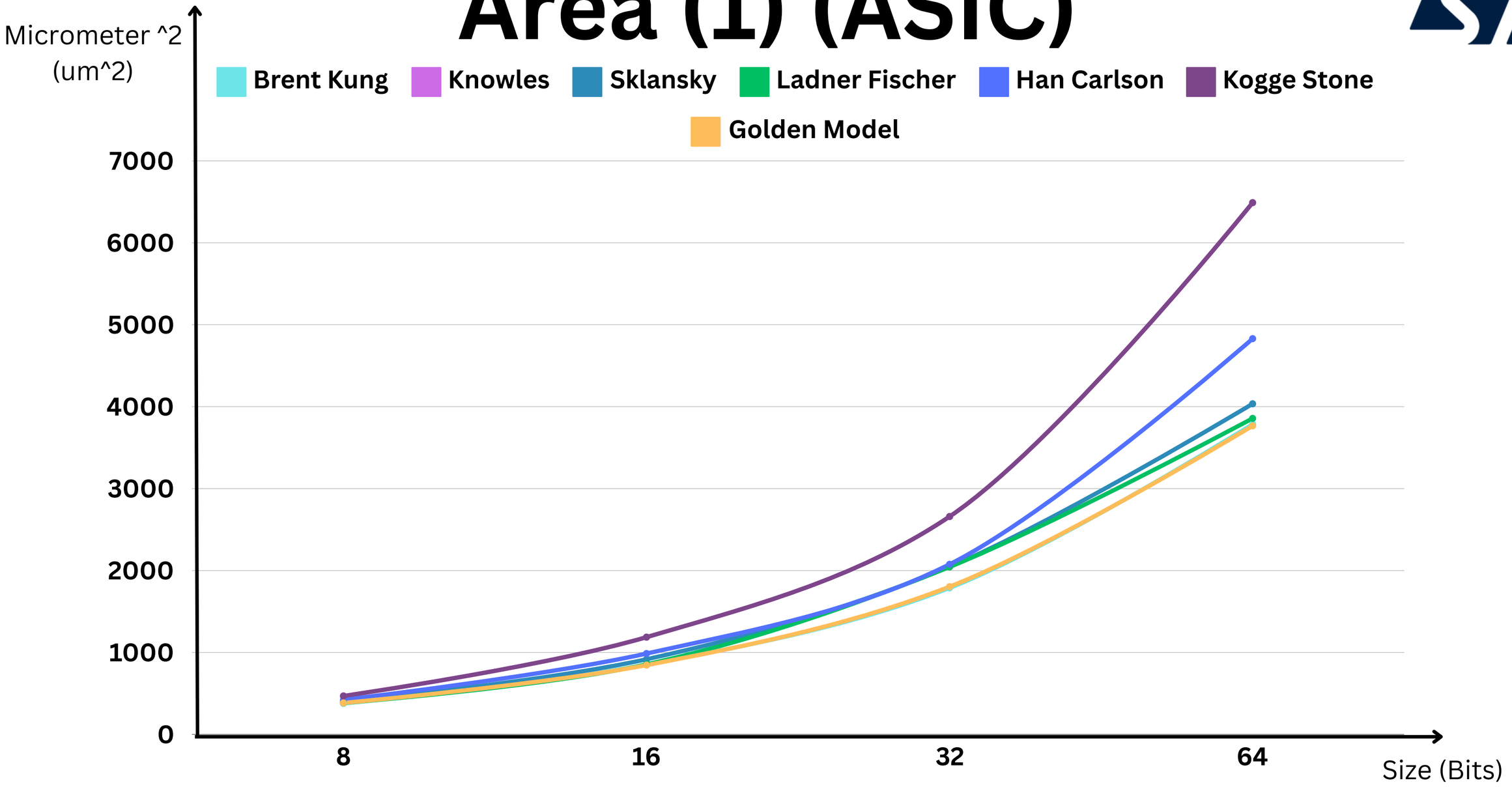
Power (2) (FPGA)


Watt
x (10^-4)
Carry Increment
Carry Lookahead
Carry Skip
Carry Ripple
Golden Model
1.200
1.000
0.800
0.600
0.400
0.200
0.000
8
16
32
64
Size (Bits)

Area (1) (ASIC)

Adder / Size in bits	8	16	32	64
Brent Kung	378	851	1788	3785
Kogge Stone	469	1187	2659	6490
Sklansky	397	917	2046	4035
Ladner Fischer	384	855	2044	3857
Han Carlson	423	987	2076	4830
Knowles	469	1187	2659	6490
Golden Model	384	846	1803	3766

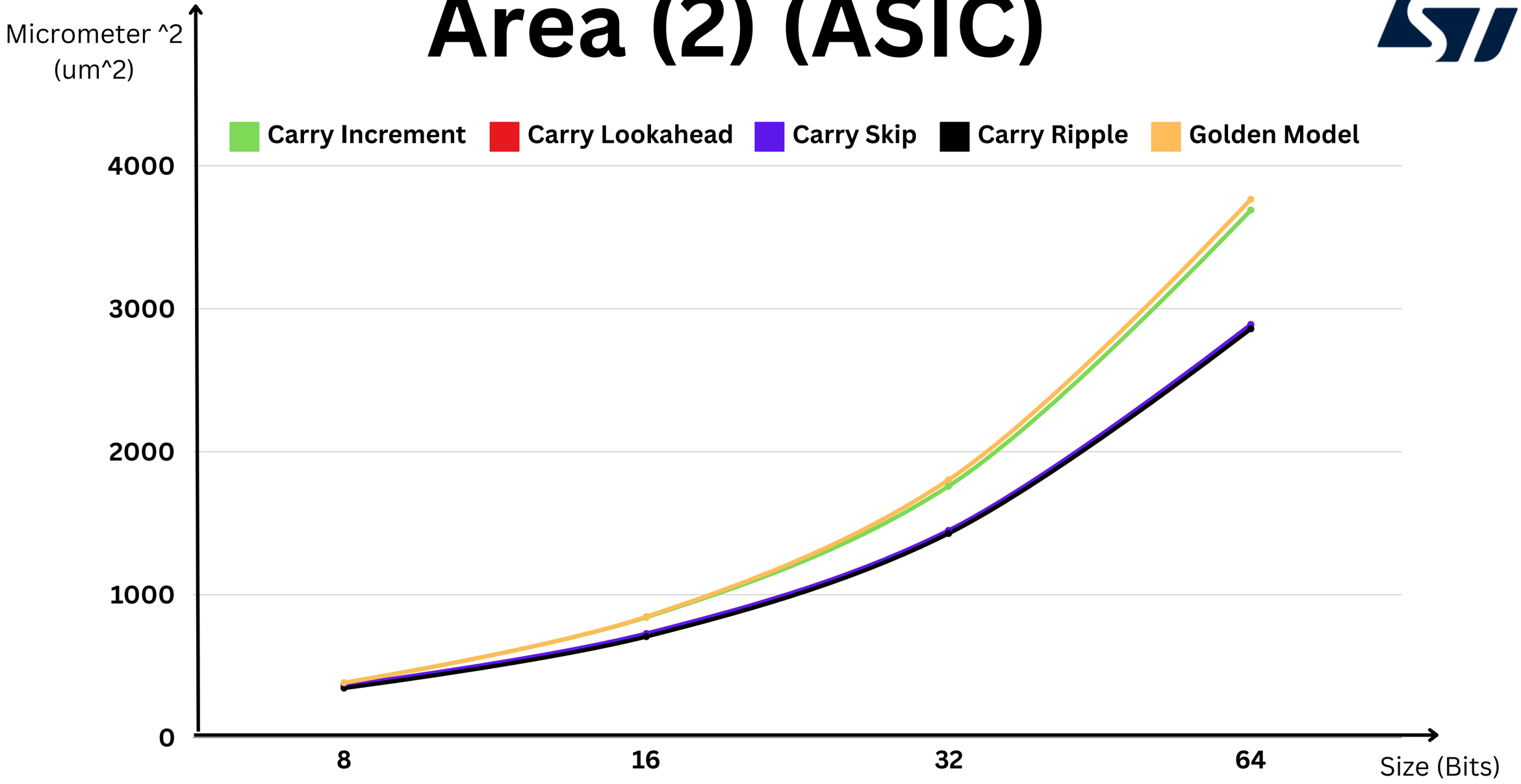
Area (1) (ASIC)


Micrometer ^2
(um^2)
Brent Kung
Knowles
Sklansky
Ladner Fischer
Han Carlson
Kogge Stone
Golden Model
7000
6000
5000
4000
3000
2000
1000
0
8
16
32
64
Size (Bits)

Area (2) (ASIC)

Adder / Size in bits	8	16	32	64
Carry Increment	384	842	1759	3690
Carry Lookahead	364	724	1445	2887
Carry Skip	364	728	1449	2890
Carry Ripple	347	707	1428	2860
Golden Model	384	846	1803	3766

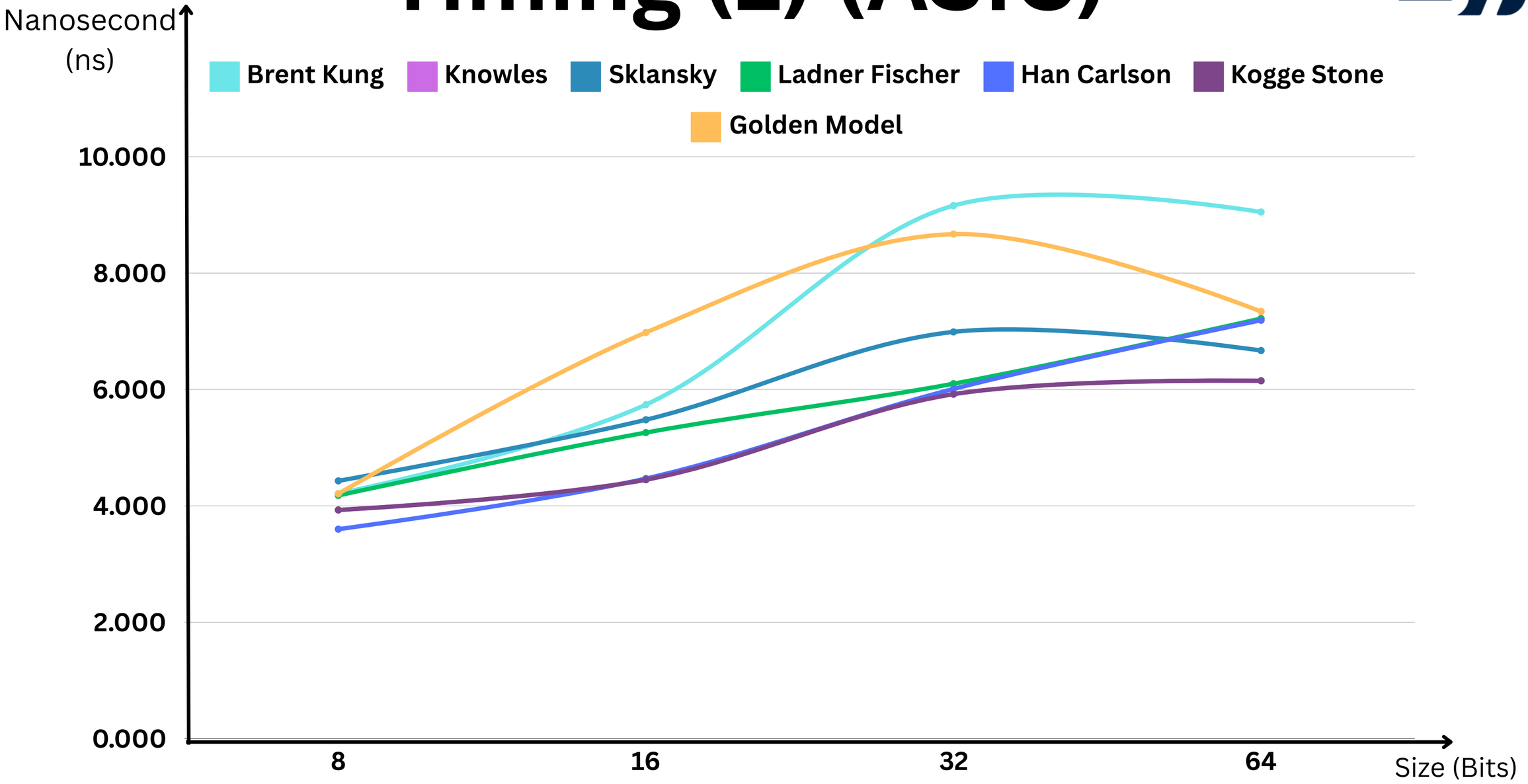
Area (2) (ASIC)


Micrometer ^2
(um^2)
Carry Increment
Carry Lookahead
Carry Skip
Carry Ripple
Golden Model
4000
3000
2000
1000
0
8
16
32
64
Size (Bits)

Timing (1) (ASIC)

Adder / Size in bits	8	16	32	64
Brent Kung	4.21	5.74	9.16	9.05
Kogge Stone	3.93	4.45	5.92	6.15
Sklansky	4.43	5.48	6.99	6.67
Ladner Fischer	4.18	5.26	6.1	7.22
Han Carlson	3.60	4.47	6.01	7.19
Knowles	3.93	4.45	5.92	6.15
Golden Model	4.21	6.98	8.67	7.34

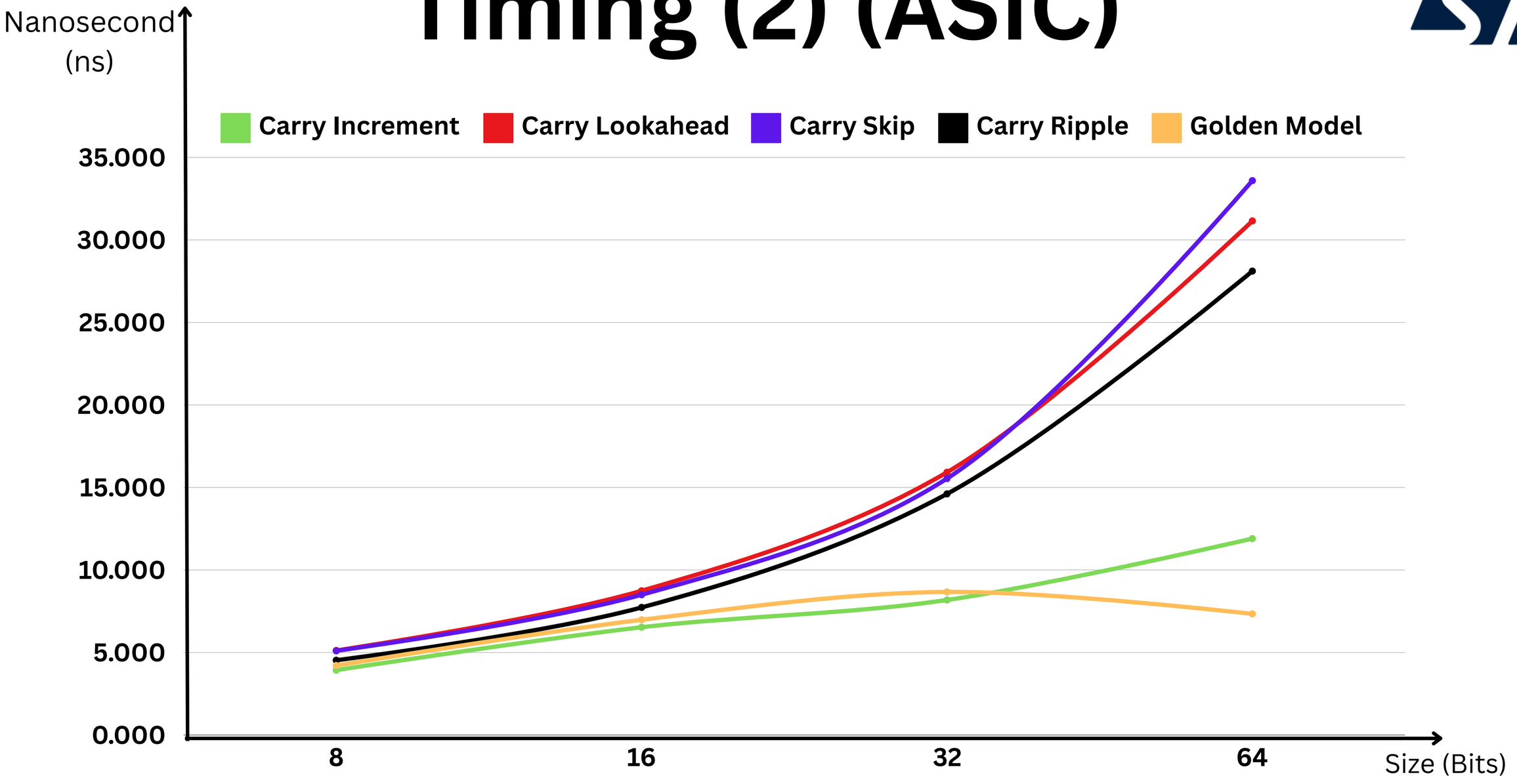
Timing (1) (ASIC)



Timing (2) (ASIC)

Adder / Size in bits	8	16	32	64
Carry Increment	3.93	6.53	8.18	11.9
Carry Lookahead	5.12	8.74	15.92	31.15
Carry Skip	5.10	8.5	15.54	33.593
Carry Ripple	4.53	7.73	14.61	28.11
Golden Model	4.21	6.98	8.67	7.34

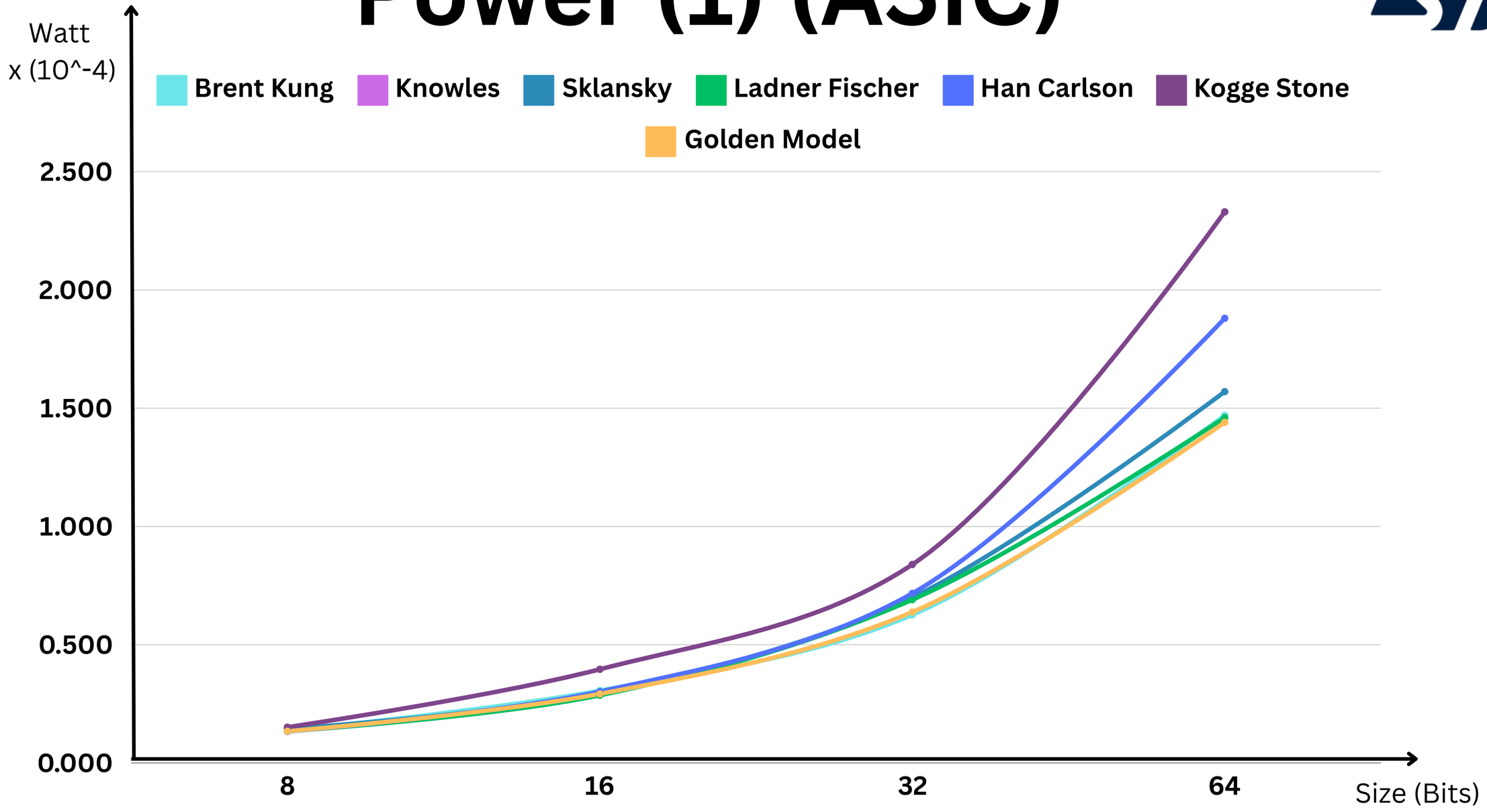
Timing (2) (ASIC)



Power (1) (ASIC)

Adder / Size in bits	8	16	32	64
Brent Kung	1.37 E-5	3.06 E-5	6.26 E-5	1.47 E-4
Kogge Stone	1.51 E-5	3.96 E-5	8.39 E-5	2.33 E-4
Sklansky	1.44 E-5	2.96 E-5	7 E-5	1.57 E-4
Ladner Fischer	1.34 E-5	2.86 E-5	6.9 E-5	1.46 E-4
Han Carlson	1.33 E-5	3.02 E-5	7.17 E-5	1.88 E-4
Knowles	1.51 E-5	3.96 E-5	8.39 E-5	2.33 E-4
Golden Model	1.34 E-5	2.92 E-5	6.38 E-5	1.44 E-4

Power (1) (ASIC)


Watt
x (10^-4)
Brent Kung
Knowles
Sklansky
Ladner Fischer
Han Carlson
Kogge Stone
Golden Model
2.500
2.000
1.500
1.000
0.500
0.000
8
16
32
64
Size (Bits)

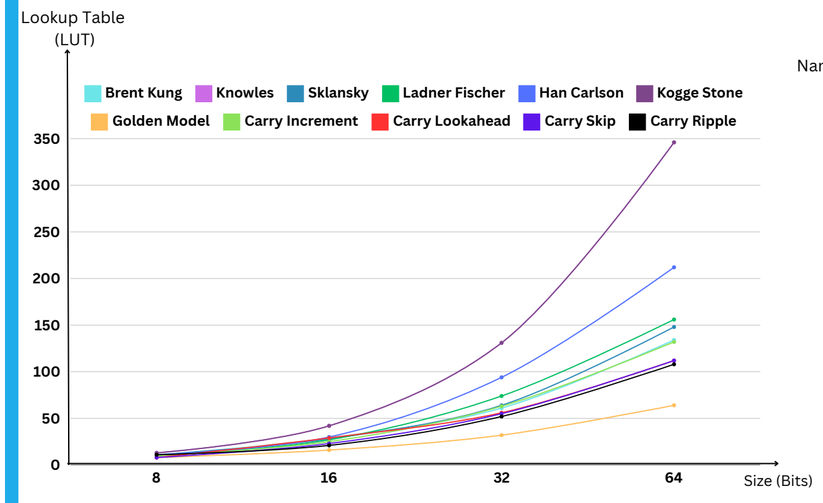
Power (2) (ASIC)

Adder / Size in bits	8	16	32	64
Carry Increment	1.34 E-5	2.92 E-5	6.44 E-5	1.46 E-4
Carry Lookahead	1.42 E-5	2.88 E-5	5.92 E-5	1.36 E-4
Carry Skip	1.42 E-5	2.85 E-5	5.89 E-5	1.27 E-4
Carry Ripple	1.33 E-5	2.73 E-5	5.84 E-5	1.27 E-4
Golden Model	1.34 E-5	2.92 E-5	6.38 E-5	1.44 E-4

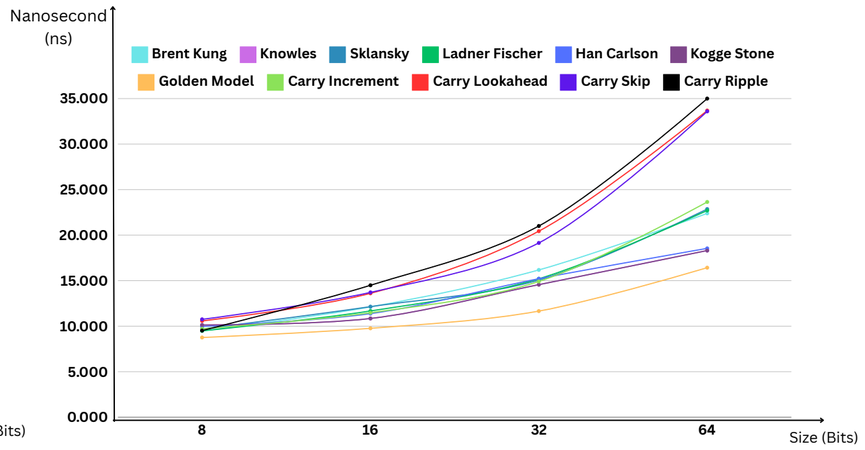
Power (2) (ASIC)


Watt
x (10^-4)
Carry Increment
Carry Lookahead
Carry Skip
Carry Ripple
Golden Model
2.000
1.500
1.000
0.500
0.000
8
16
32
64
Size (Bits)

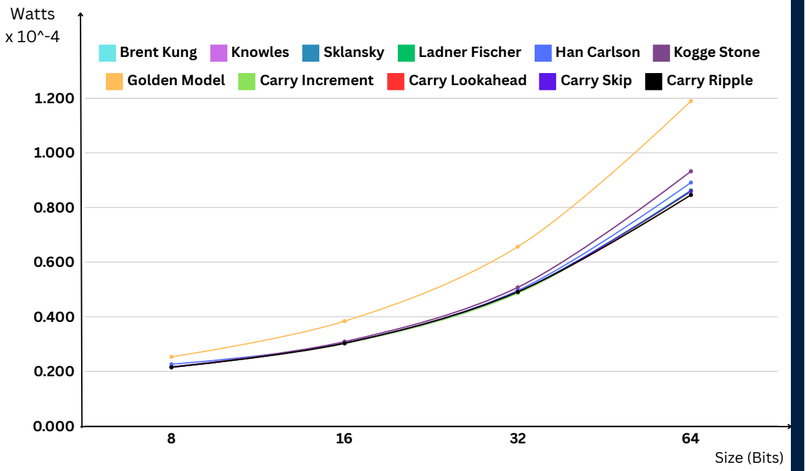
Summary & Results (FPGA)



Area



Timing



Power

Area & Timing

- Golden Model (+ operator) outperforms all adders due to FPGA's optimized carry chains in CLBs, minimizing area and routing delays
- Custom adder architectures (e.g., Kogge-Stone, Brent-Kung) are built from scratch using LUTs, resulting in higher area and timing overhead

Summary & Results (FPGA)

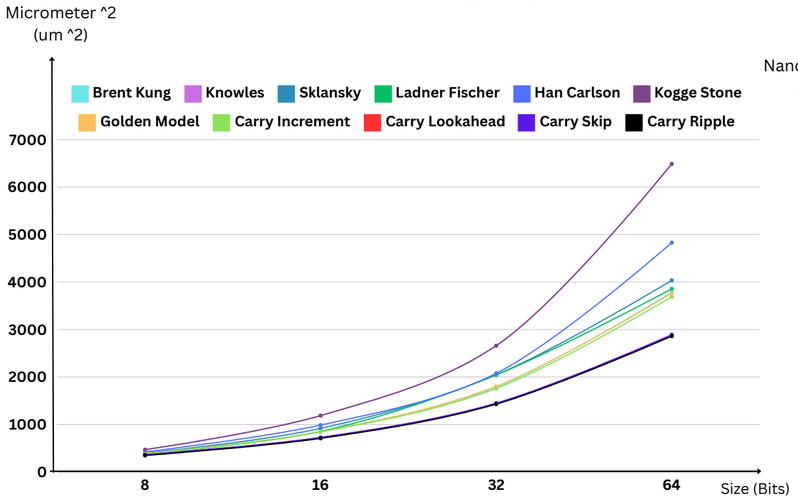
Area (Prefix Adders)

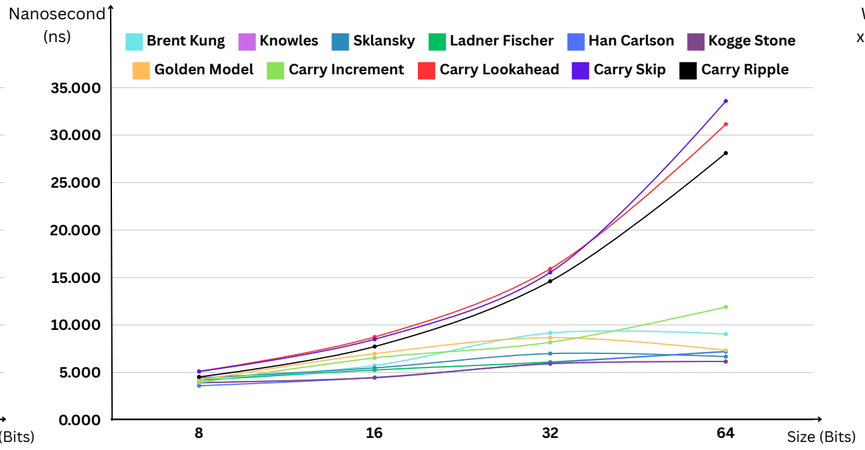
- **Kogge-Stone: Worst area among prefix adders**
- **Brent-Kung: Lowest area among prefix adders**

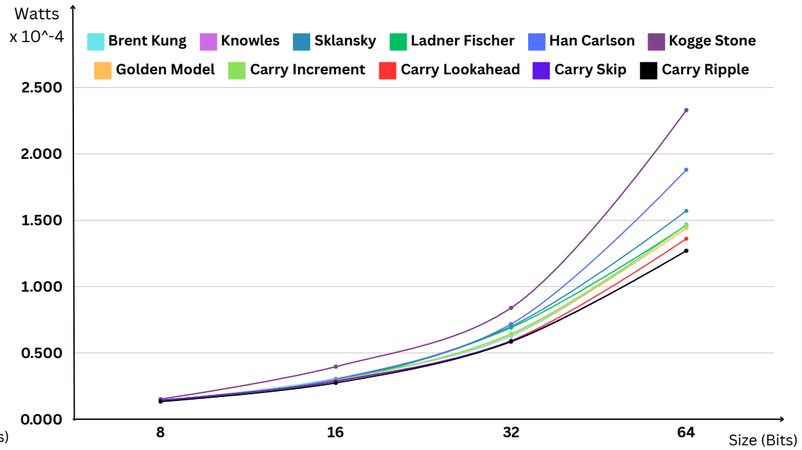
Power

- **Golden Model (+ operator) consumes more power across all bit sizes (8, 16, 32, 64 bits)**
- **Carry Ripple and CLA: Lowest power, especially at 64 bits**
- **Brent-Kung: Best power-efficiency trade-off among prefix adders for speed and low power**

Summary & Results (ASIC)







Area

Timing

Power

Overview

- (+ operator) (Golden Model) performs poorly across all metrics, as ASIC synthesis uses standard cells without optimization for specific blocks

Summary & Results (ASIC)

Area

- Kogge-Stone: Worst area among all adders
- Brent-Kung: Best area among prefix adders
- Carry Ripple: Lowest area overall, as expected
- Golden Model: Resembles Brent-Kung in area (overlaps in graphs)

Performance (Timing)

- Golden Model: Inconsistent timing (e.g., 32-bit slower than 64-bit due to inferred adder types like Brent-Kung vs. Han-Carlson)
- Kogge-Stone and Knowles: Fastest Adders especially for higher bit sizes

Power

- Kogge-Stone and Knowles: Worst power consumption
- No clear best prefix adder for power; varies by bit size
- Carry Ripple: Best absolute power efficiency, as expected

Github Link

You can access all RTL Codes on Github also: LINK

References

[1] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, 2nd ed. New York, NY, USA: Oxford University Press, 2010.

[2] N. H. E. Weste and D. Money Harris, CMOS VLSI Design: A Circuits and Systems Perspective, 4th ed. Boston, MA, USA: Pearson, 2011.

[3] H. A. Omran, "Adders," in Digital IC Design, Lecture 26, Integrated Circuits Laboratory (ICL), Electronics and Communications Eng. Dept., Faculty of Engineering, Ain Shams University, Cairo, Egypt.

[4] M. Talsania and E. John, "A Comparative Analysis of Parallel Prefix Adders," Dept. of Electrical and Computer Engineering, University of Texas at San Antonio, San Antonio, TX, 78249, USA.

[5] N. Aruna Kumari, M. Sai Srinivas, and K. Aravind, "Implementation of 64 Bit Arithmetic Adders," Int. J. Recent Technol. Eng. (IJRTE), vol. 7, no. 5S4, pp. 554–558, Feb. 2019.

Thank You