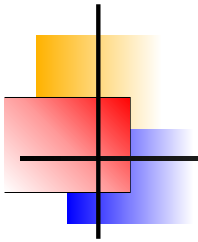
Multiplier Architecture with a Carry-Based Partial Product Encoding

Martin Langhammer, Bogdan Pasca, Igor Kucherenko

Intel Corporation

ARITH 2024
10-12 June, 2024
Malaga, Spain



Acknowledgement

Some elements of this work already exist in the following US patent:
`https://patents.google.com/patent/US10466968B1/en`

This work was conducted independently without any prior knowledge of its existence.

Why do we care about multipliers?

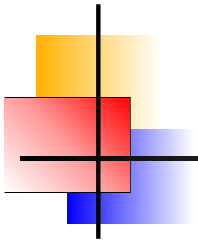
Agilex™ 7 FPGAs and SoCs F-Series Product Table

altera An Intel Company

Version 2024.01.09

Product Line		AGF 006	AGF 008	AGF 012	AGF 014	AGF 019	AGF 023	AGF 022	AGF 027
Resources	Logic elements (LEs)	573,480	764,640	1,178,525	1,437,240	1,918,975	2,308,080	2,208,075	2,692,760
	Adaptive logic modules (ALMs)	194,400	259,200	399,500	487,200	650,500	782,400	748,500	912,800
	ALM registers	777,600	1,036,800	1,598,000	1,948,800	2,602,000	3,129,600	2,994,000	3,651,200
	High-performance crypto blocks	0	0	0	0	2	2	0	0
	eSRAM memory blocks	0	0	2	2	1	1	0	0
	eSRAM memory size (Mb)	0	0	36	36	18	18	0	0
	M20K memory blocks	2,844	3,792	5,900	7,110	8,500	10,464	10,900	13,272
	M20K memory size (Mb)	56	74	115	139	166	204	212	259
	MLAB memory count	9,720	12,960	19,975	24,360	32,525	39,120	37,425	45,640
	MLAB memory size (Mb)	6	8	12	15	20	24	23	28
	Fabric PLL	6	6	8	8	5	5	12	12
	I/O PLL	12	12	16	16	10	10	16	16
	Variable-precision digital signal processing (DSP) blocks	1,640	2,296	3,743	4,510	1,354	1,640	6,250	8,528
	18 x 19 multipliers	3,280	4,592	7,486	9,020	2,708	3,280	12,500	17,056
	Single-precison or half-precision tera floating point operations per second (TFLOPS)	2.5 / 5.0	3.5 / 6.9	6.0 / 12.0	6.8 / 13.6	2.0 / 4.0	2.5 / 5.0	9.4 / 18.8	12.8 / 25.6
	Maximum EMIF x72[2]	4	4	4	4	3	3	4	4

Recent FPGAs embed thousands of DSP Blocks



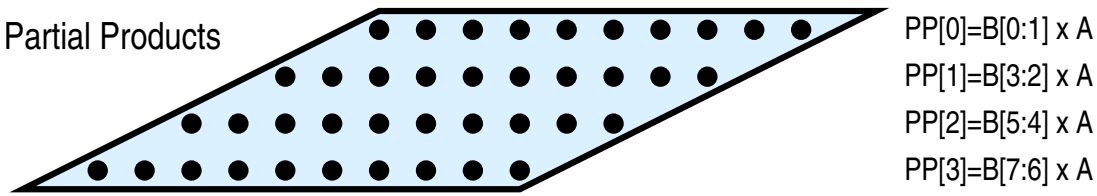
What can a recent Agilex 5 DSP Block do?

Multiplier	DSP Block Resource Usage
9x9 bits	6x per DSP block
18x19 bits	2x per DSP block
27x27 bits	1x per DSP block
half-precision	2x per DSP block
bfloat16	2x per DSP block
FP19(8,10)	2x per DSP block
single-precision	1x per DSP block
AI tensor : 2 x 10 x (8x8-bit)	1x per DSP Block

Many smaller-bitwidth multipliers used as internal building blocks

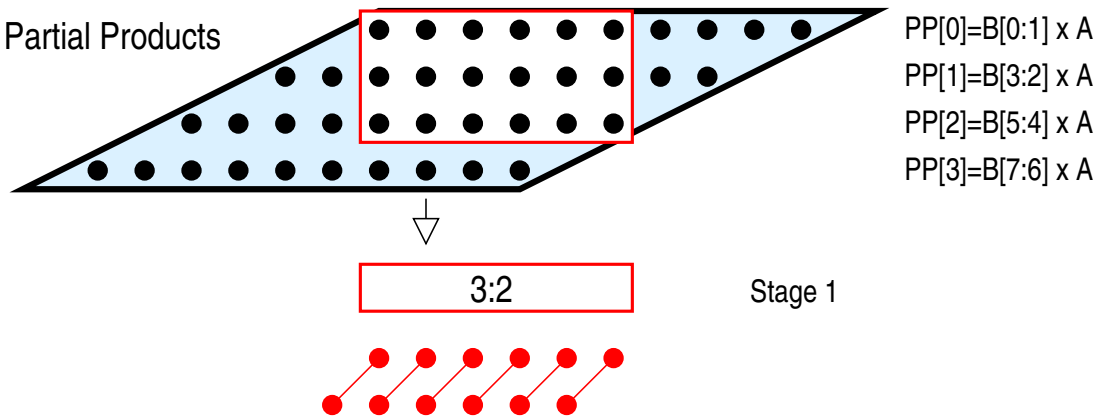
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



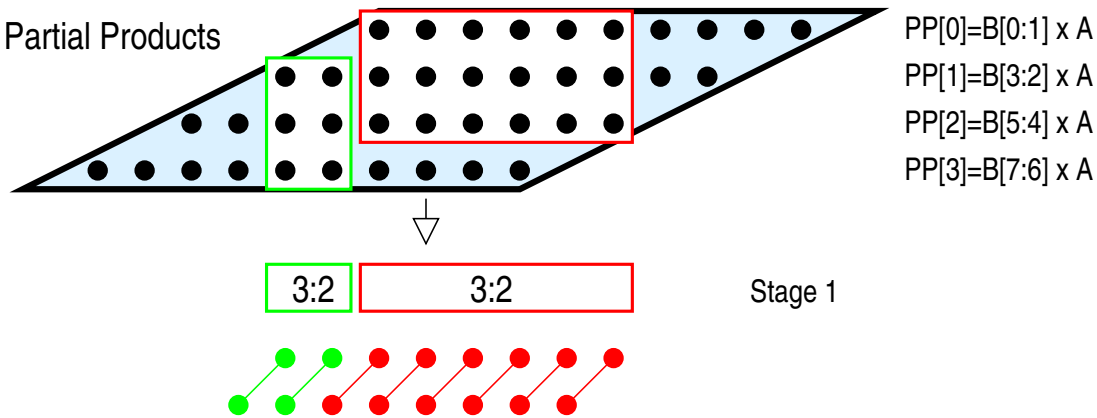
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



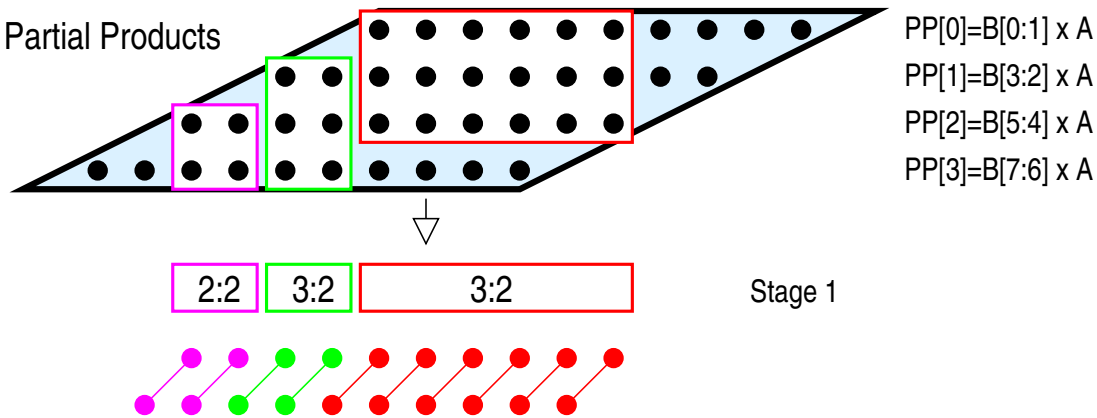
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



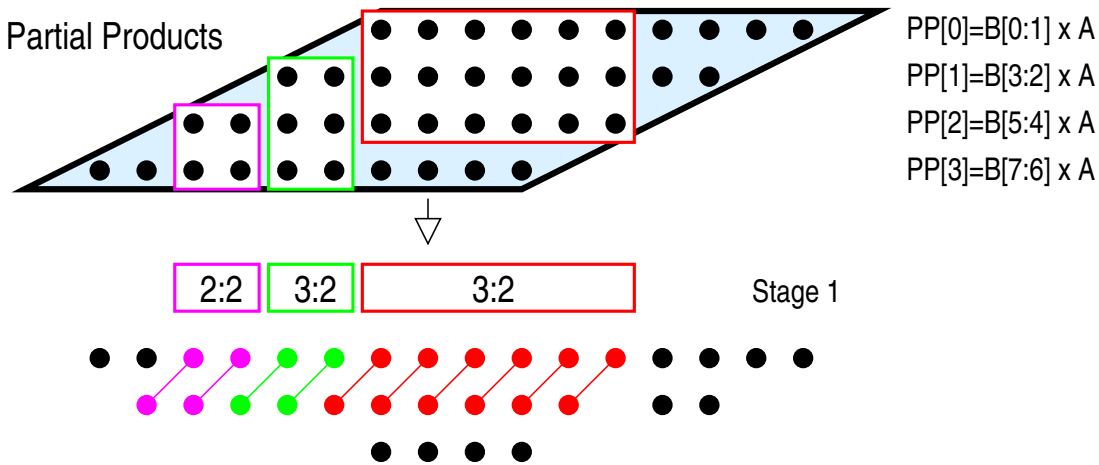
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



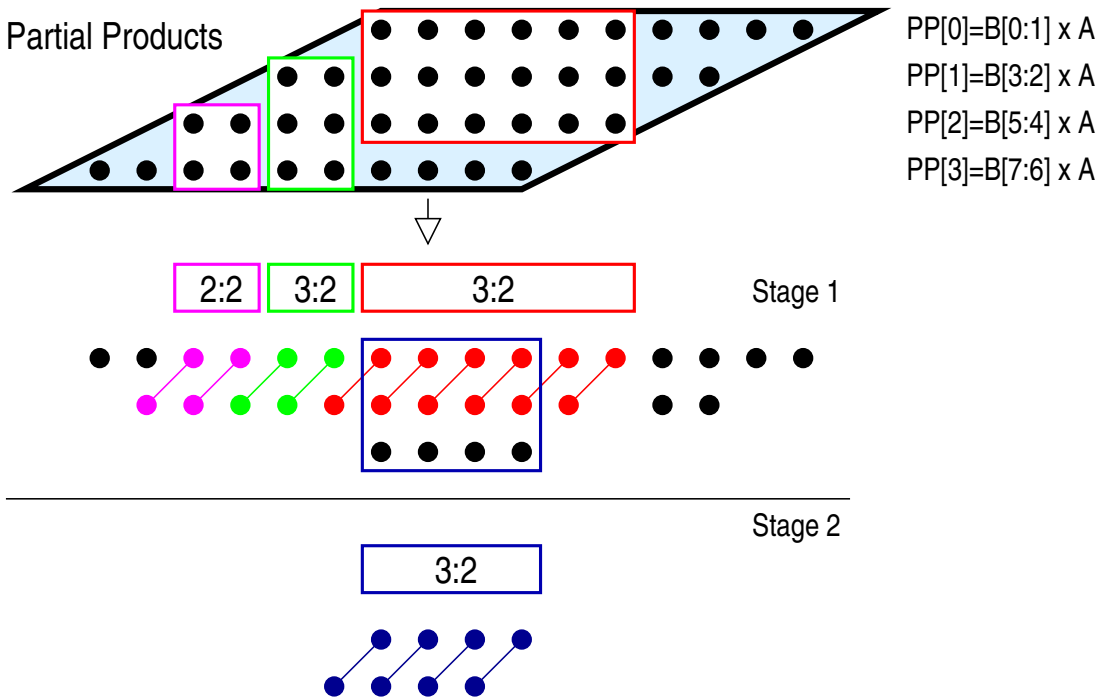
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



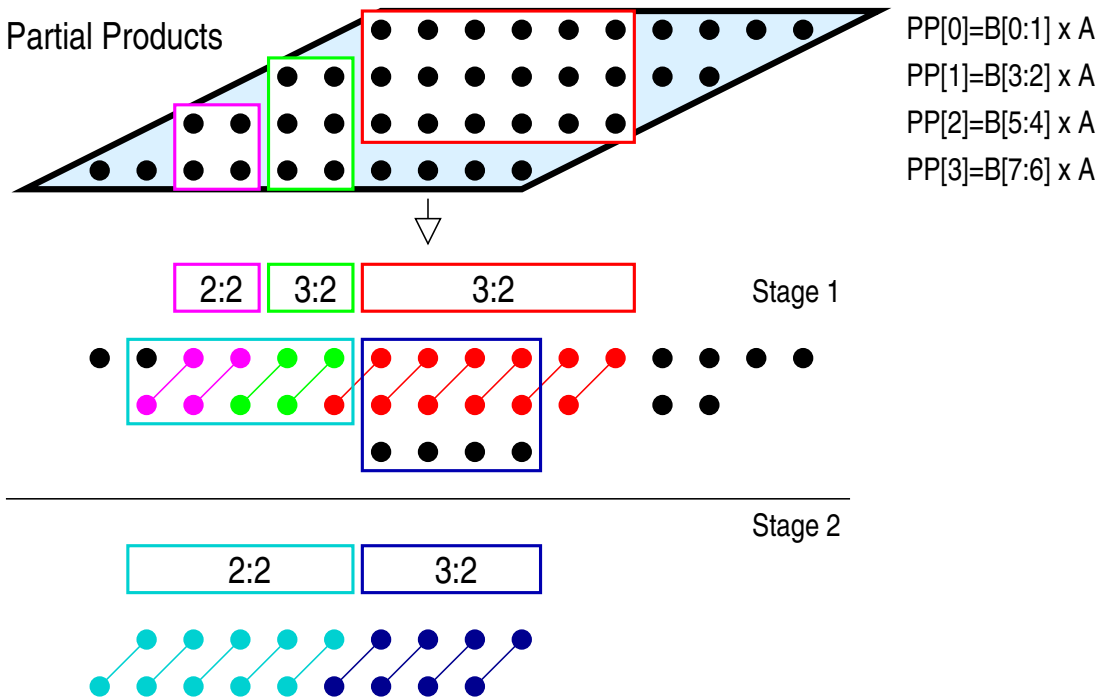
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



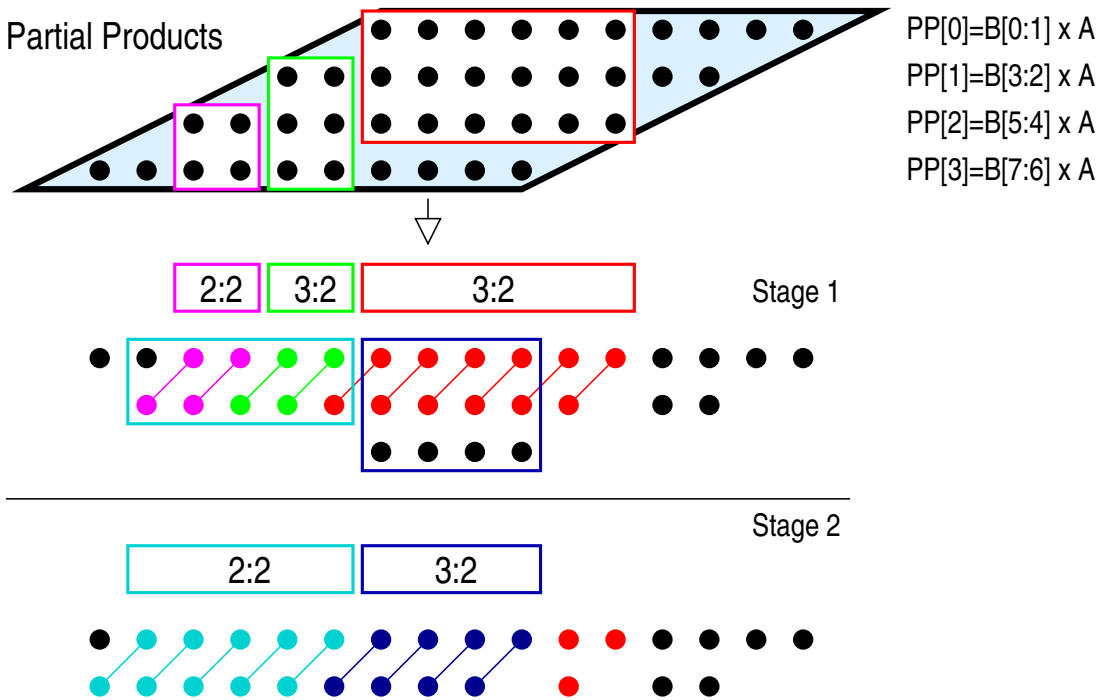
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



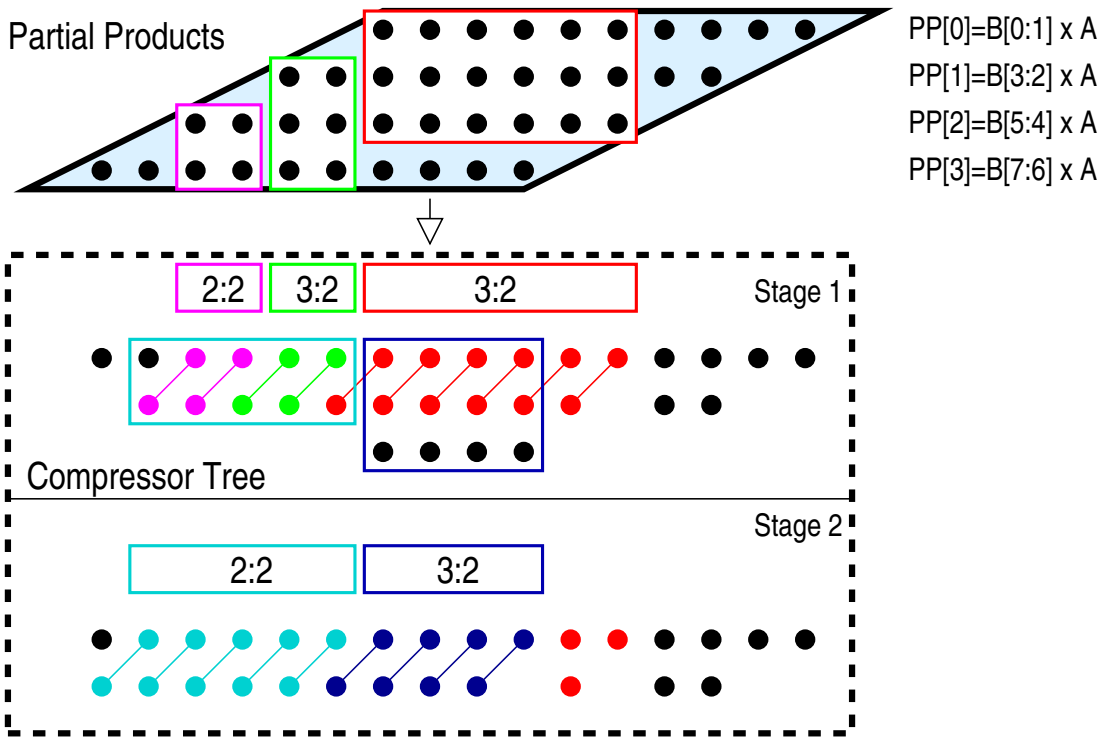
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



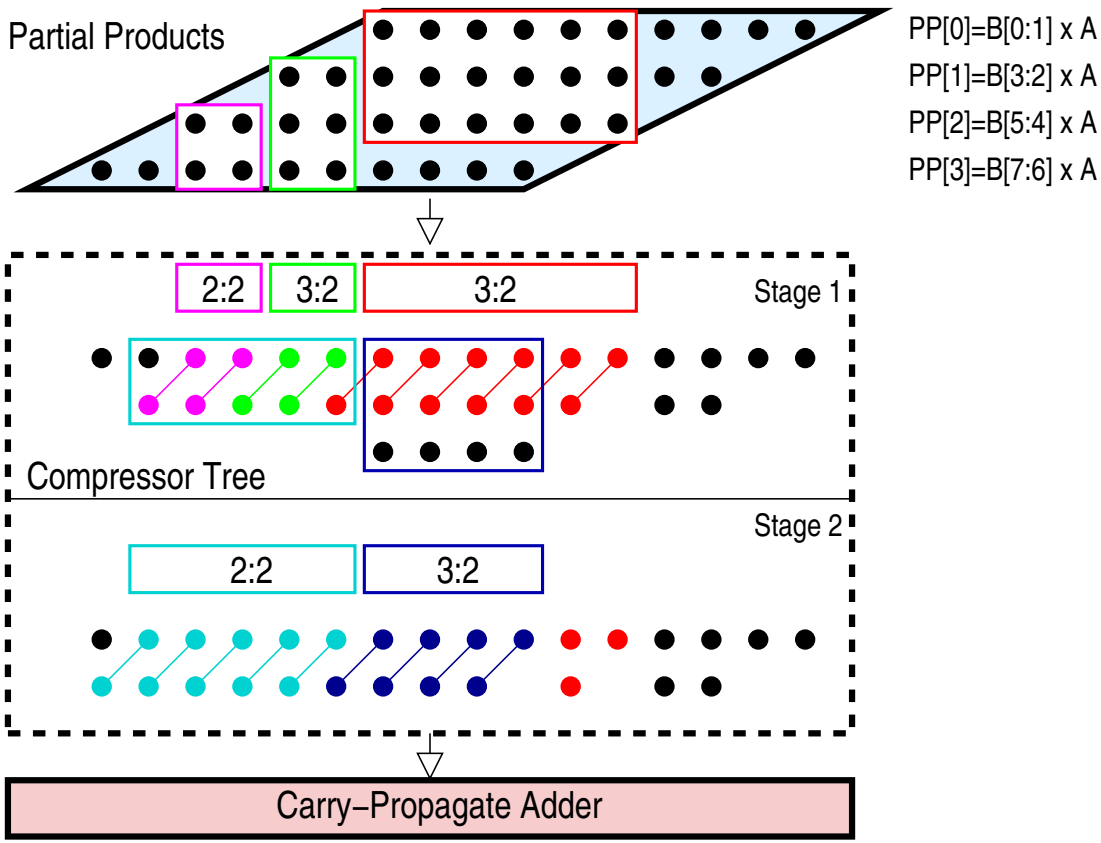
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



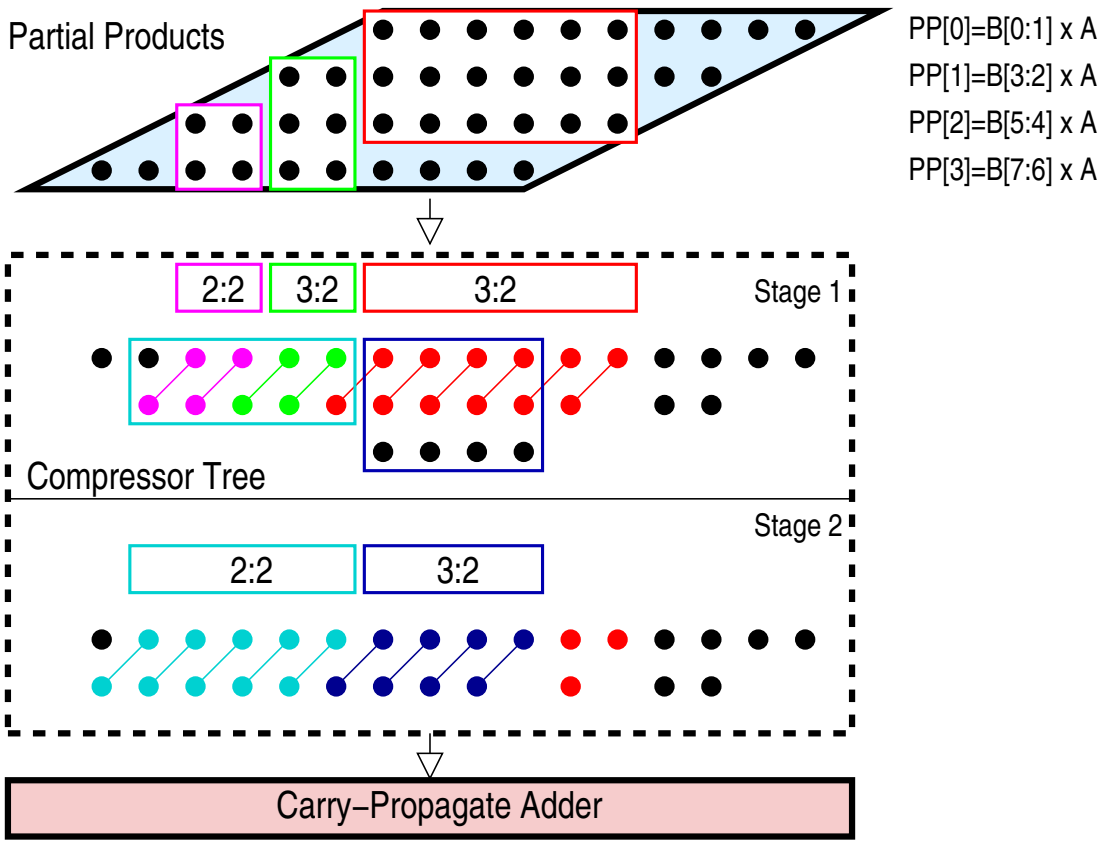
Steps for implementing a multiplier

Example: Radix 4, 8 x 8-bit unsigned



Steps for implementing a multiplier

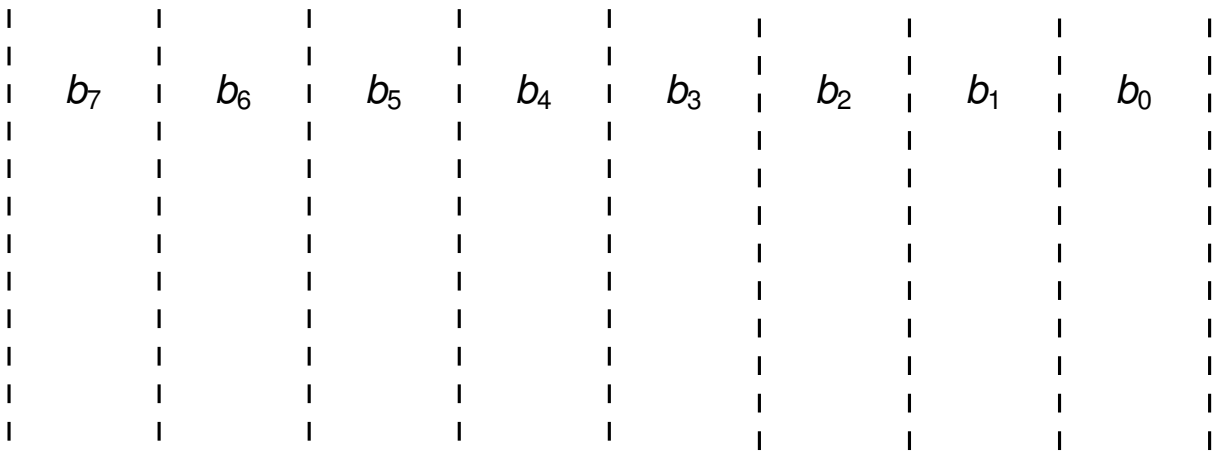
Example: Radix 4, 8 x 8-bit unsigned



In this work we focus on the partial-product generation

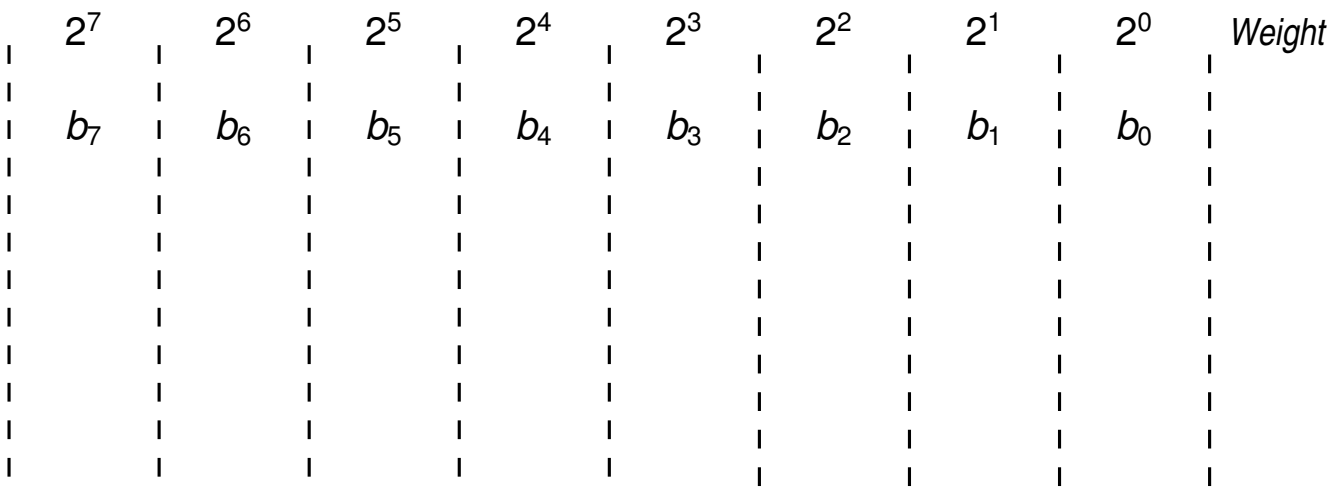
Partial Product Generation: Multiplier Encoding

Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



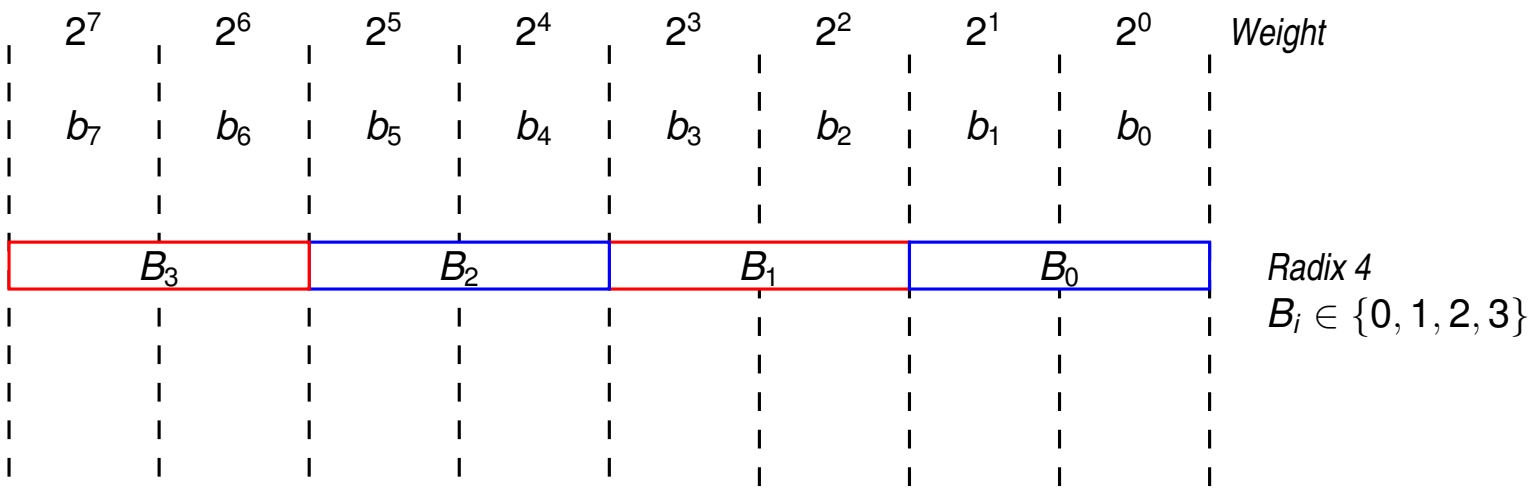
Partial Product Generation: Multiplier Encoding

Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



Partial Product Generation: Multiplier Encoding

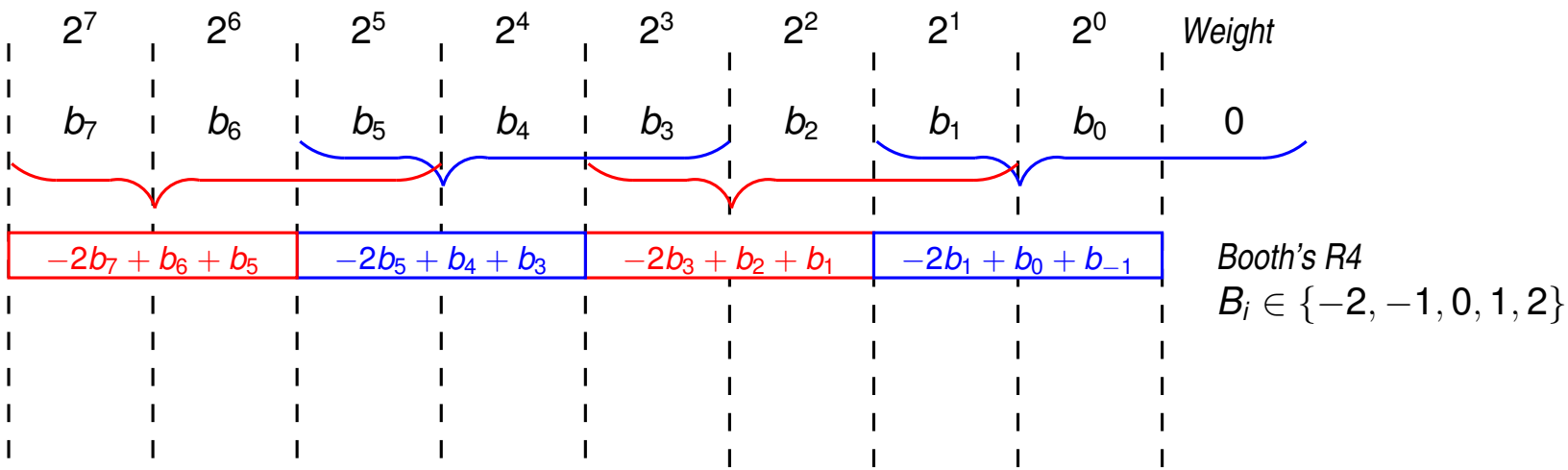
Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



Radix 4: half the PP of Radix 2, but more complex 3A multiple required

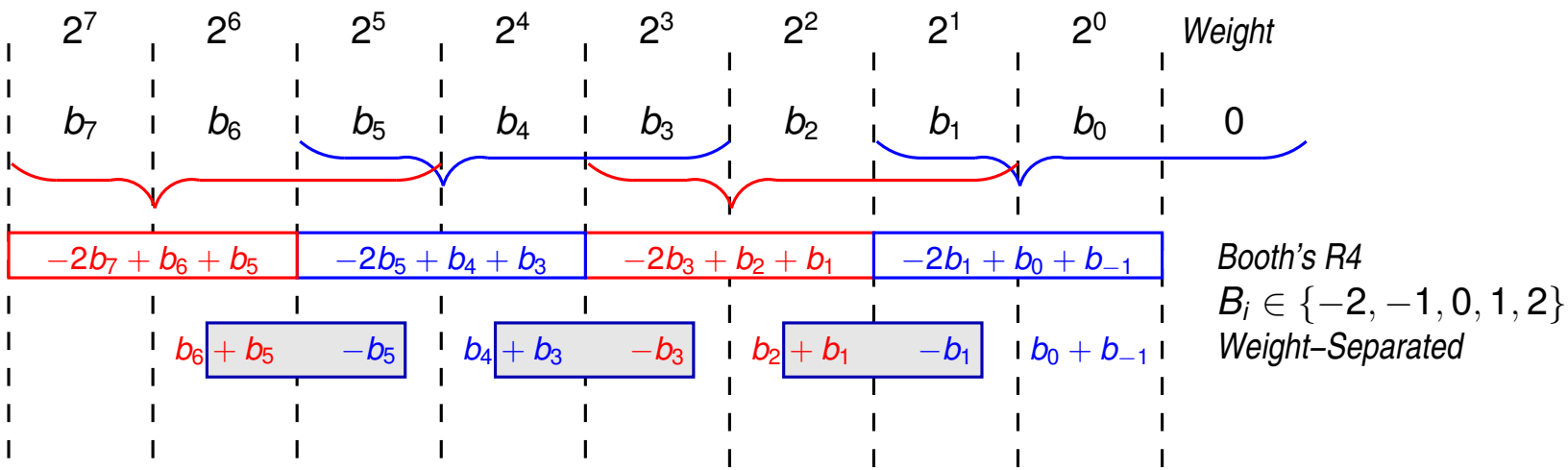
Partial Product Generation: Multiplier Encoding

Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



Partial Product Generation: Multiplier Encoding

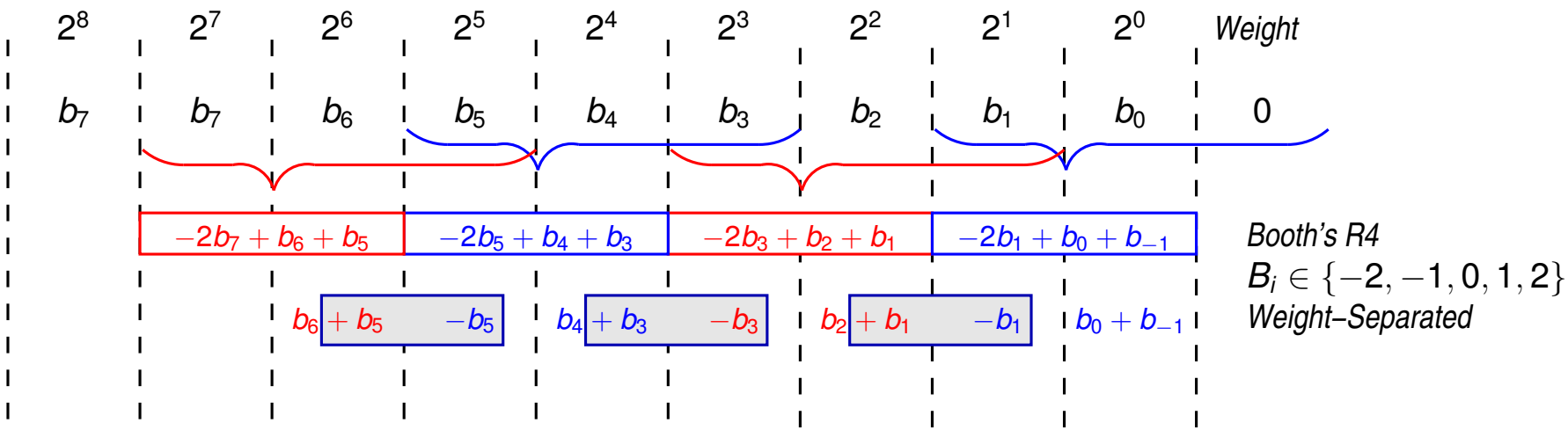
Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



Booth's Radix 4: half the PP of Radix 2*, simple multiples required

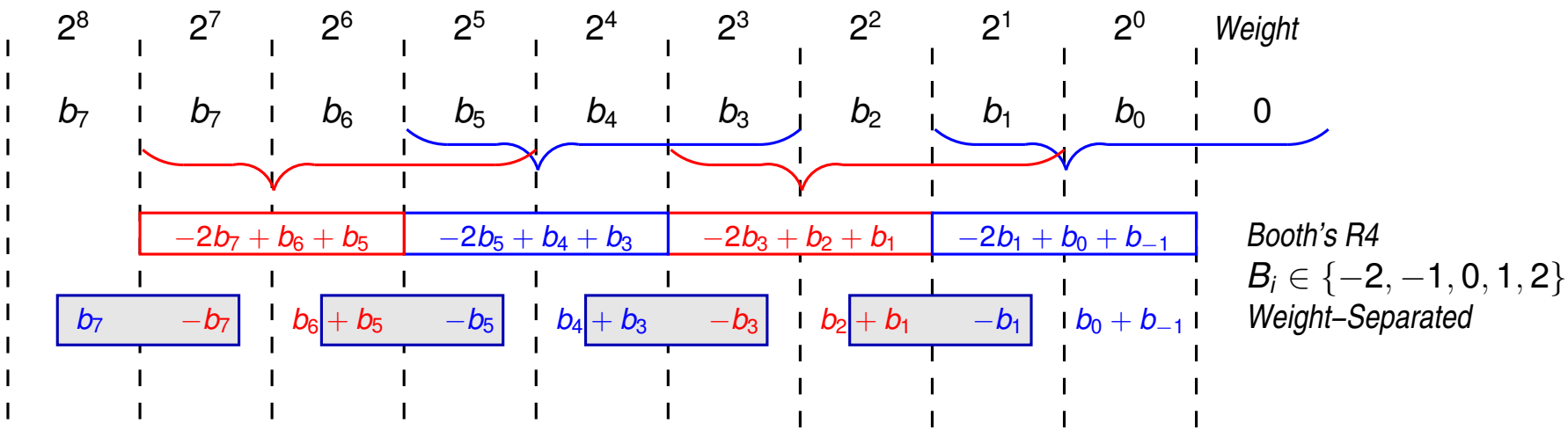
Partial Product Generation: Multiplier Encoding

Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



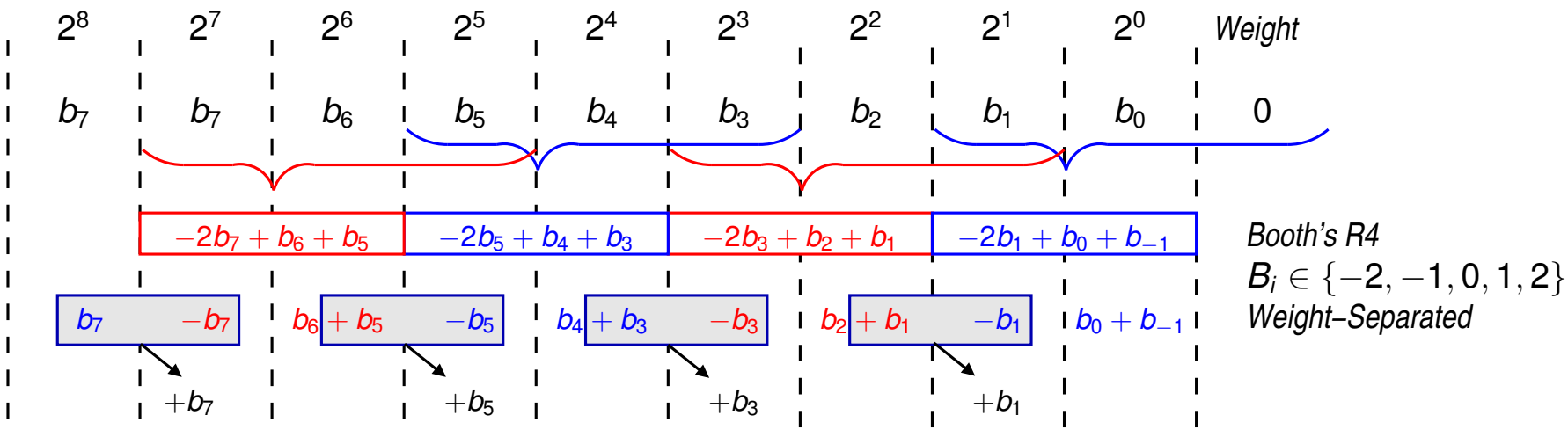
Partial Product Generation: Multiplier Encoding

Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



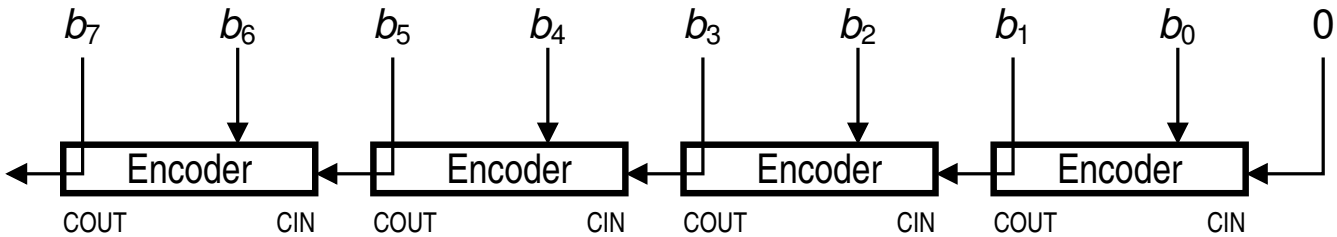
Partial Product Generation: Multiplier Encoding

Example: Radix 4 vs Modified Booth's Radix 4 for 8-bit unsigned B



Partial Product Generation: Multiplier Encoding (2)

Dibit encoding can be seen as a carry chain



b_{2j+1} **CO**	b_{2j}	b_{2j-1} **CI**	**B** $b_{2j+1:2j}$	**M**	$B + b_{2j-1}$	**CO**
0	0	0	0	0	0	0
0	0	1	0	1	1	0
0	1	0	1	1	1	0
0	1	1	1	2	2	0
1	0	0	2	-2	2	1
1	0	1	2	-1	3	1
1	1	0	3	-1	3	1
1	1	1	3	0	4	1

Alternate Multiplier Encoding

- counter-intuitive approach: use an alternate encoding
- goal: reduce input count to the partial product multiplexer

CI	B	M	CO
0	0	0	0
0	1	1	0
0	2	**-2**	**1**
0	3	-1	1
1	0	1	0
1	1	+2	0
1	2	-1	1
1	3	0	1

Alternate Multiplier Encoding

- counter-intuitive approach: use an alternate encoding
- goal: reduce input count to the partial product multiplexer
- replace "-2" with carry-out of 1 $\rightarrow$ "+2" with carry-out of 0.
- this creates a dependency between the carries

CI	B	M	CO
0	0	0	0
0	1	1	0
0	2	**-2**	**1**
0	3	-1	1
1	0	1	0
1	1	+2	0
1	2	-1	1
1	3	0	1

$\Rightarrow$

CI	B	M	CO
0	0	0	0
0	1	1	0
0	2	**+2**	**0**
0	3	-1	1
1	0	1	0
1	1	+2	0
1	2	-1	1
1	3	0	1

Handling the carry dependencies

CI	B	M	CO
0	0	0	0
0	1	1	0
0	2	**+2**	**0**
0	3	-1	1
1	0	1	0
1	1	+2	0
1	2	-1	1
1	3	0	1

- use the concept of prefix computations for computing carries
- define the generate and propagate across dibits

Handling the carry dependencies

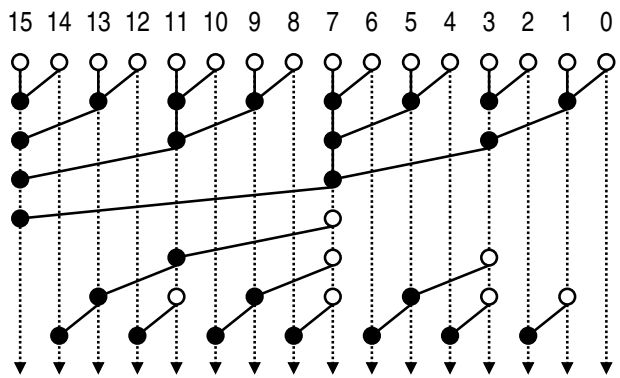
CI	B	M	CO
0	0	0	0
0	1	1	0
0	2	**+2**	**0**
0	3	-1	1
1	0	1	0
1	1	+2	0
1	2	-1	1
1	3	0	1

- use the concept of prefix computations for computing carries
- define the generate and propagate across dibits
- **generate** when b_{2j+1}, b_{2j} are 1.
- **propagate** when b_{2j+1} is 1.

Handling the carry dependencies

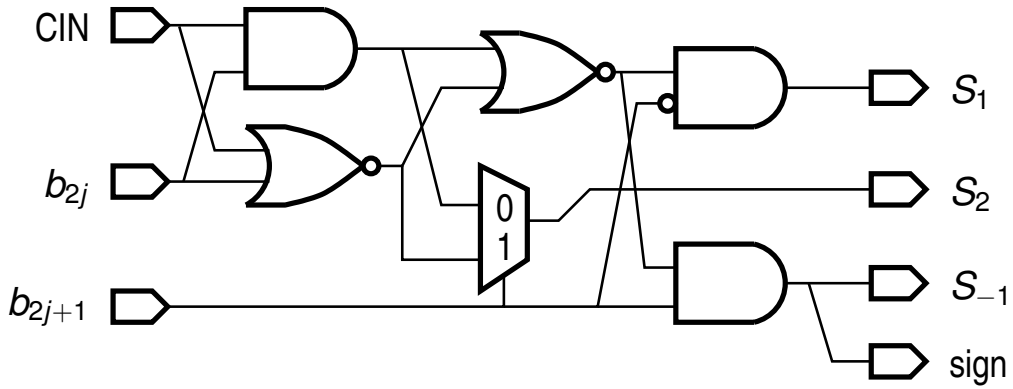
CI	B	M	CO
0	0	0	0
0	1	1	0
0	2	**+2**	**0**
0	3	-1	1
1	0	1	0
1	1	+2	0
1	2	-1	1
1	3	0	1

- use the concept of prefix computations for computing carries
- define the generate and propagate across dibits
- **generate** when b_{2j+1}, b_{2j} are 1.
- **propagate** when b_{2j+1} is 1.

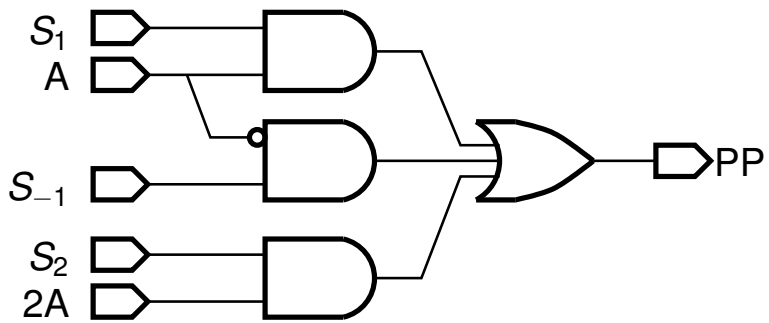


Encoder and Partial-Product Multiplexer

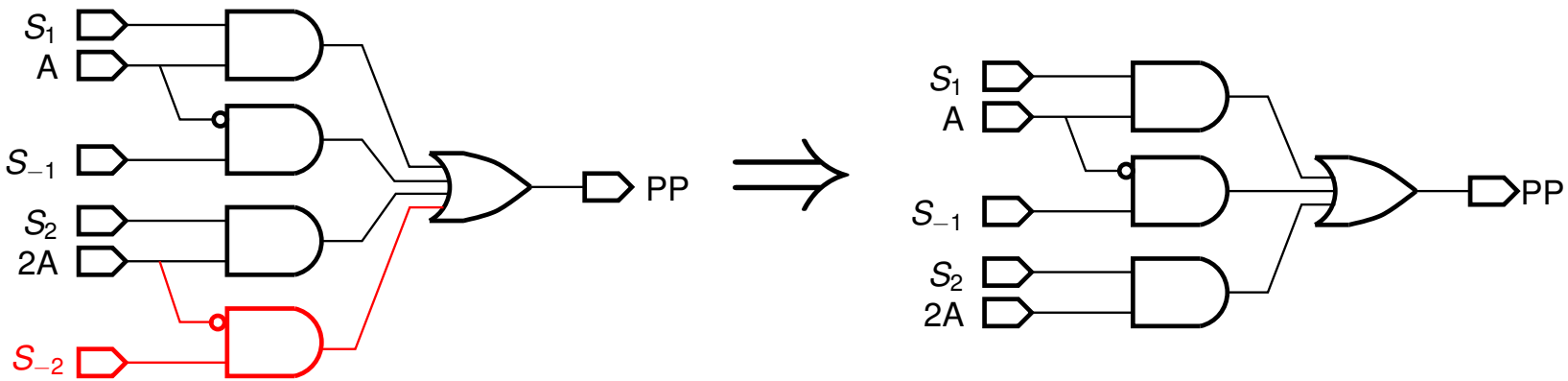
- Encoder



- Partial-Product Multiplexer

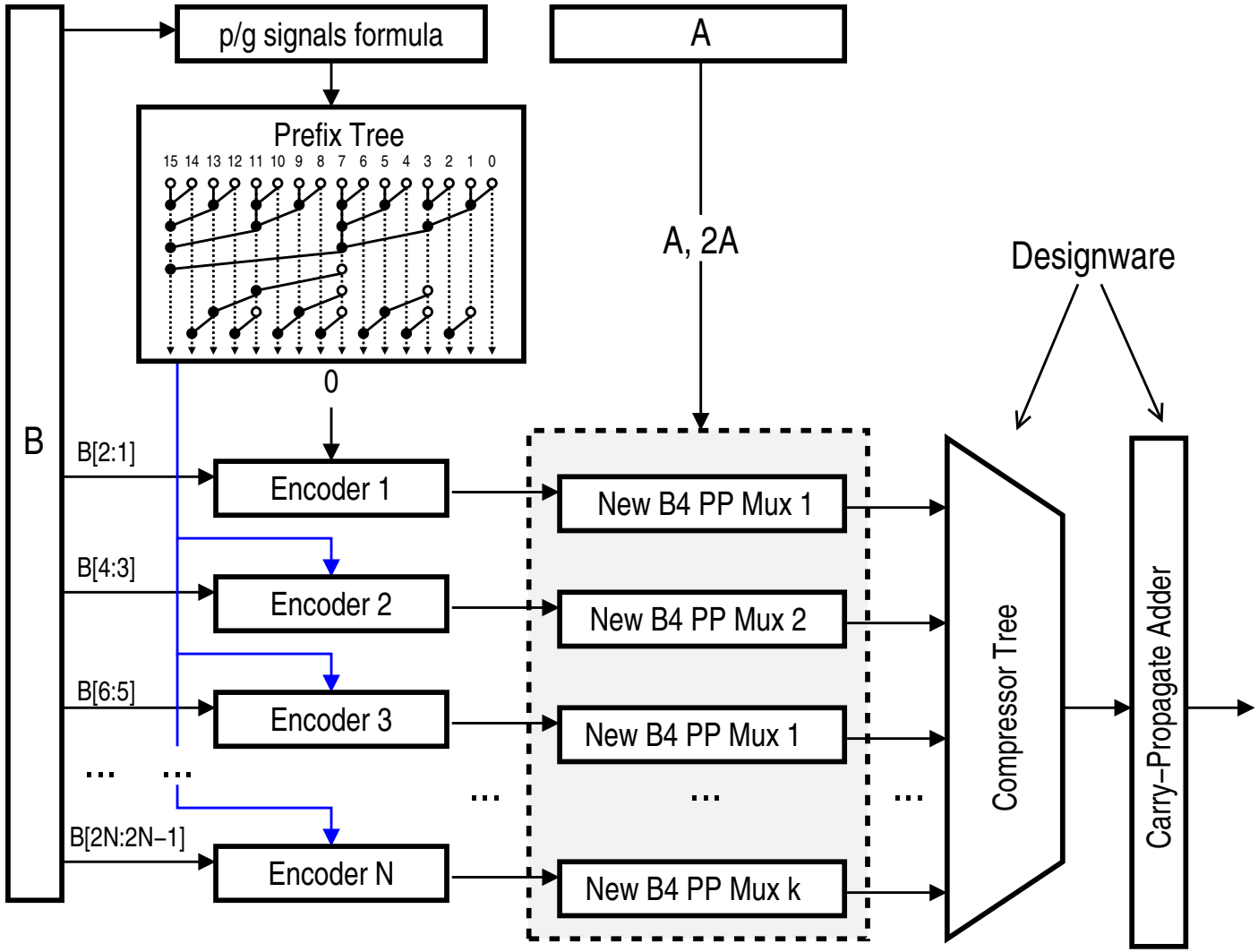


Why does this work?

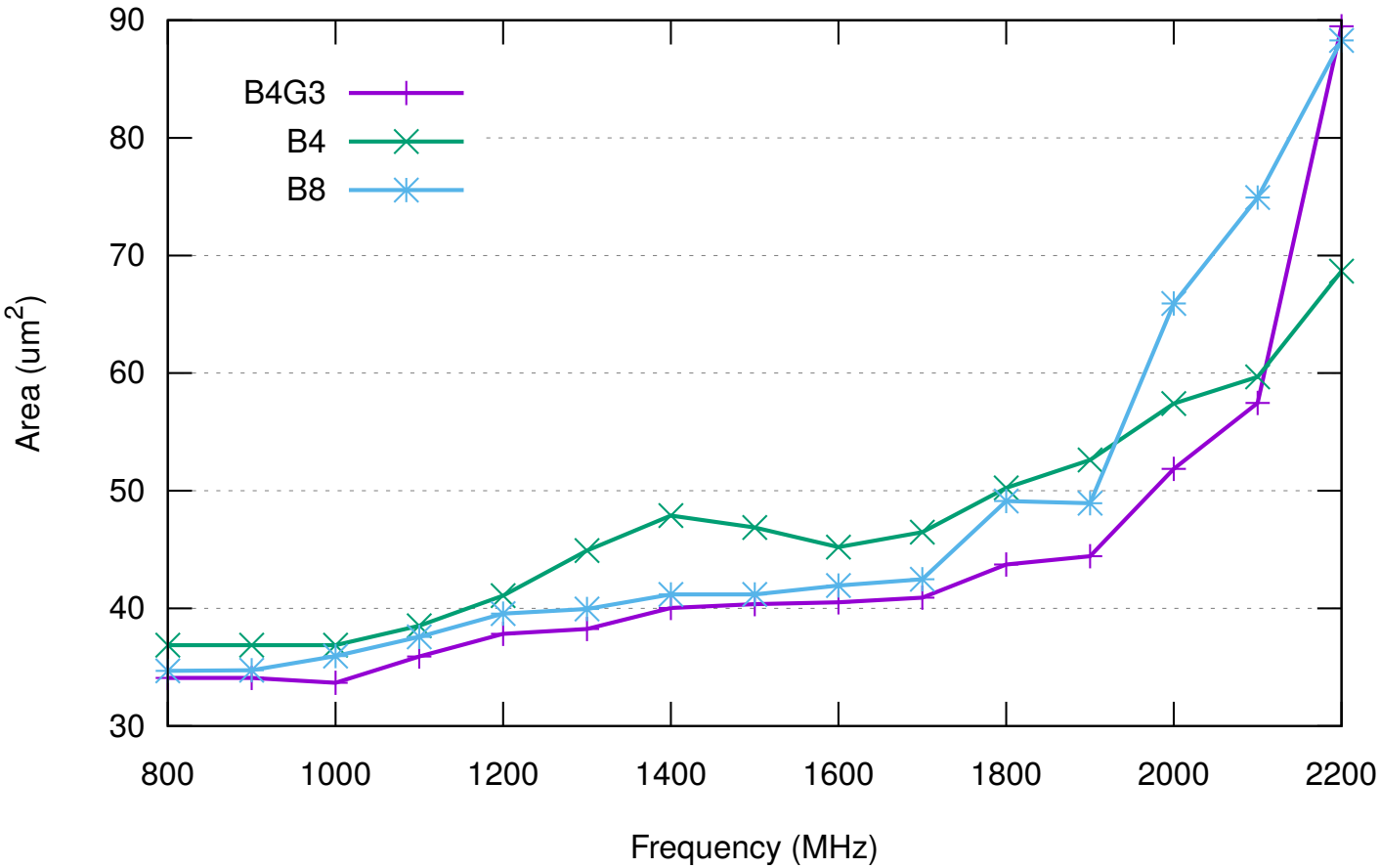


- 16-bit multiplier $\rightarrow$ 11 DOTS in Brent-Kung Tree $\approx$ 33 gates
- removing mux input: 2 gates $\times$16-bit $\times$8 PP $\approx$ 256 gates

Proposed Carry-Chain-Based Encoder - Setup

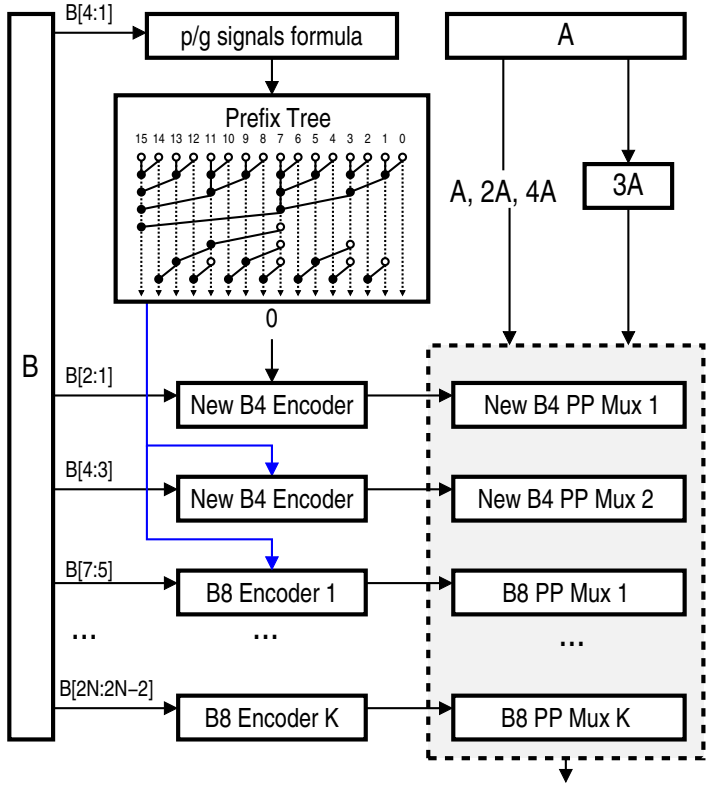


Results - 12-bit signed multiplier

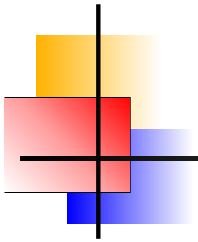


- B4 and B8: Synopsys Designware multipliers
- B4G3 yields better area for 800 MHz – 2 GHz targets

Future Work - Use in Mixed Radix



- Proposed method - additional delay in the multiplier encoding
- Higher radix (B8) - adds delay in the multiplicand (3A)
- Combined (Hybrid) approach may yield lower area



Conclusion

- Surge in AI is pushing multiplier densities on all devices.
- Efficient architectures are crucial.
- Multiplier encoding change reduces PP mux size.
- New encoder dependency solved using prefix structures.
- Synthesis results: better logic usage (800MHz - 2.1GHz)
- Use in mixed-radix multipliers shows promising results.