

An asynchronous, iterative implementation of the original Booth multiplication algorithm

Aristides Efthymiou, Wannarat Suntiamorntut,

Jim Garside, Linda Brackenbury

University of Manchester

<http://www.cs.man.ac.uk/apt>

Motivation: data-dependent processing

- Exploited by asynchronous design
 - Fast circuits, operating in the “average case”
 - Low power
 - Low area
 - Example: carry ripple adder
- Usually associated with delay-insensitive circuits
 - Large area, large cap. → Speed, Power
 - High signal transition activity → Power
- Want to use bundled-data

Array multipliers

- Usual choice for high-speed multiplication
 - Easily pipelined to improve throughput
 - Typically use modified-Booth encoding, Wallace carry-save tree with carry-propagation adder as last stage
- But offer limited opportunities for data-dependency
 - Bypassing a carry-save adder does not save much time
 - If pipelined, some stages will be at their worst case, slowing down those in “typical” case
- Thus, we investigate iterative multipliers

The original Booth algorithm

$A \times 0 \underline{0} 1 1 1 \underline{1} 0 0$ 4 additions

$A \times +1 \quad \quad -1$ 2 add/sub

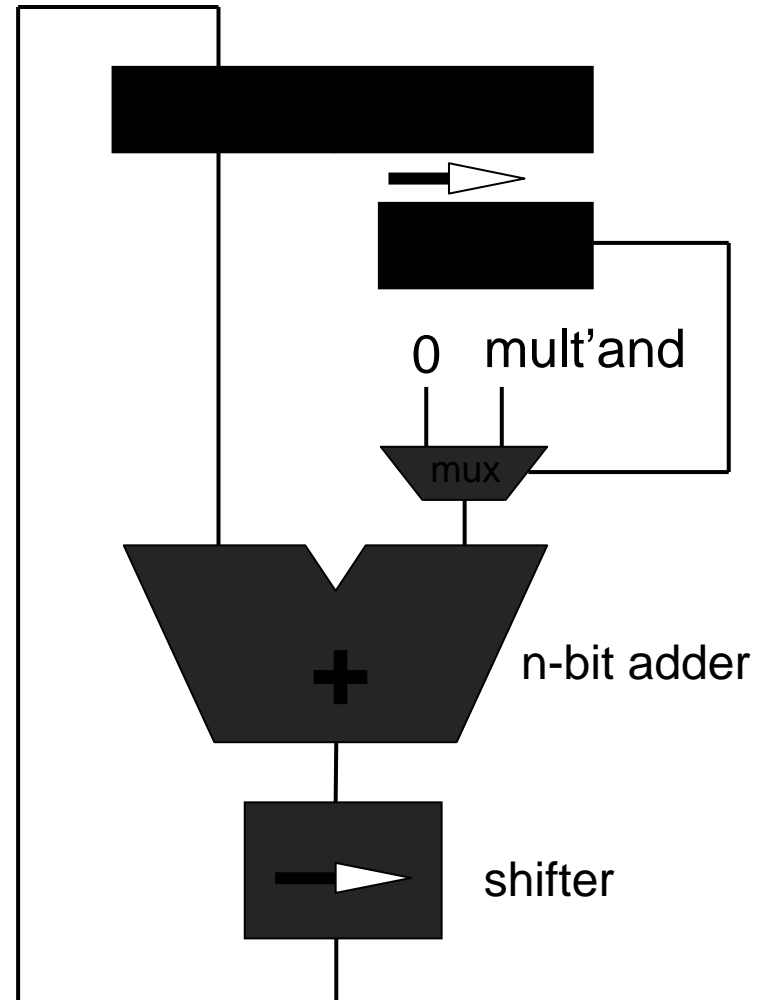
- Data-dependent number of operations
- Works with negative multiplicands
- No previous asynchronous implementation known
- But, worst case multiplicand values require more operations than needed:

$0 1 0$ 1 addition

$+1 -1$ 2 add/sub

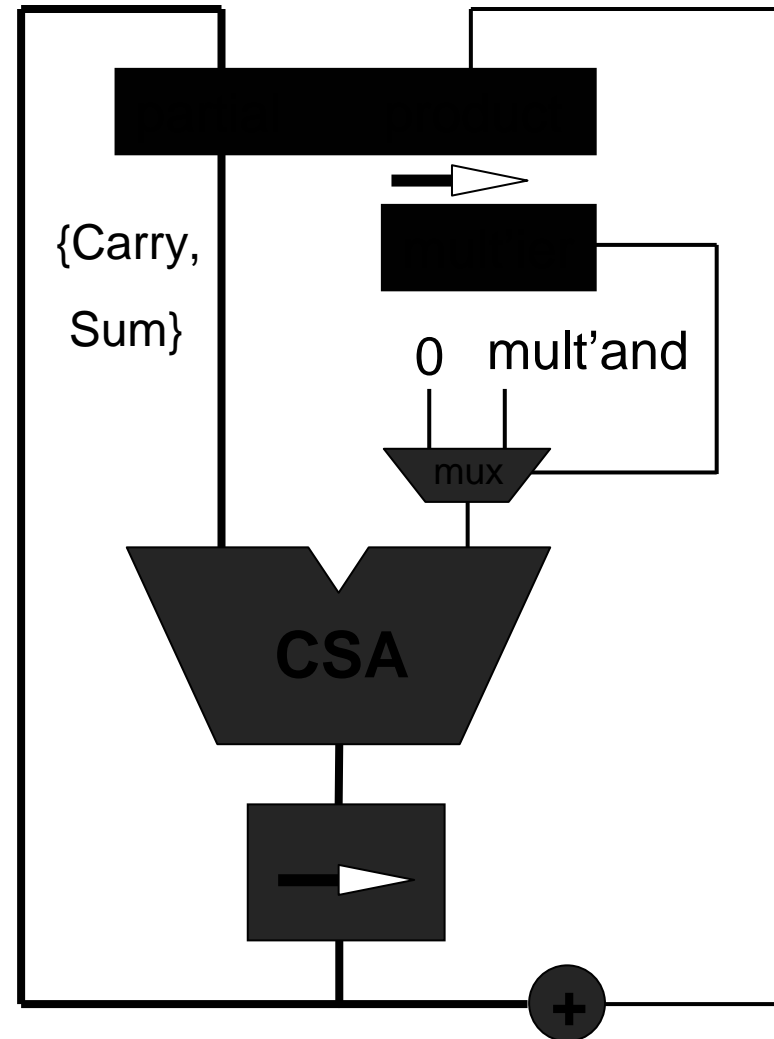
Iterative multipliers

- Relationship between shift step, number of iterations, Booth radix
- Simplest case: no Booth enc.
 - constant shift by 1
 - n iterations
- Booth-radix4
 - constant shift by 2
 - n/2 iterations
- ...



(Smarter) iterative multipliers

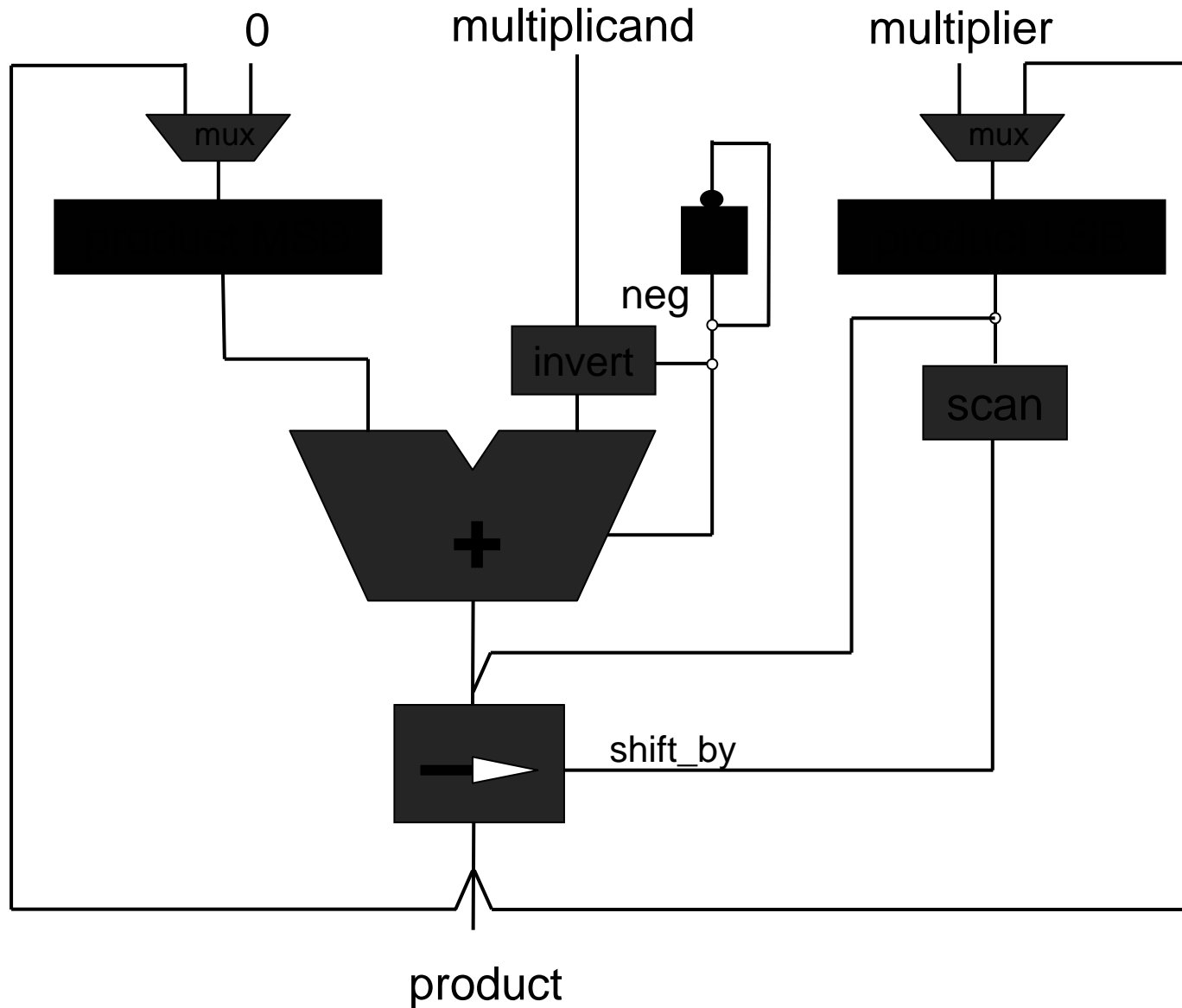
- Carry-save addition (CSA)
 - Faster, possibly lower power
 - But less potential for data-dependent delay/power
- Early-termination
 - When remaining part of multiplier all 0's or 1's terminate the process
 - Requires a shifter to align final product
 - Assumes small integers, does not work well for fractional arithmetic



Trade-offs in iterative multipliers

- Constant shifting → fixed number of iterations
 - Control overhead paid in full for each iteration
- Variable shifting per iteration
 - Variable number of iterations
 - Expectation that typical data require a few iterations
 - Need a “real” shifter: large, slow, consuming energy
- Is it better to do fewer iterations by doing more “work” per iteration?

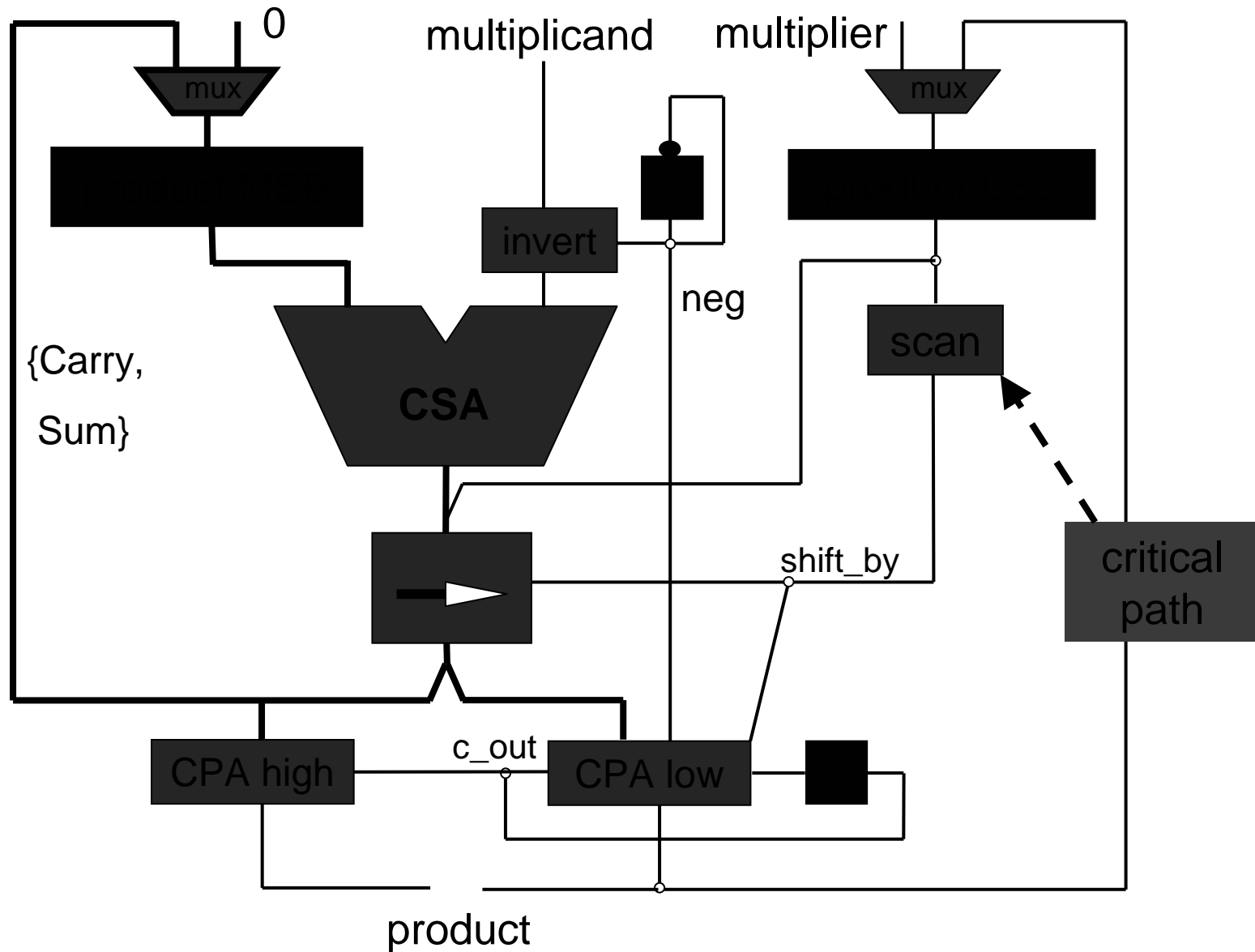
Simple Booth multiplier – CPA



Reducing the worst case # operations

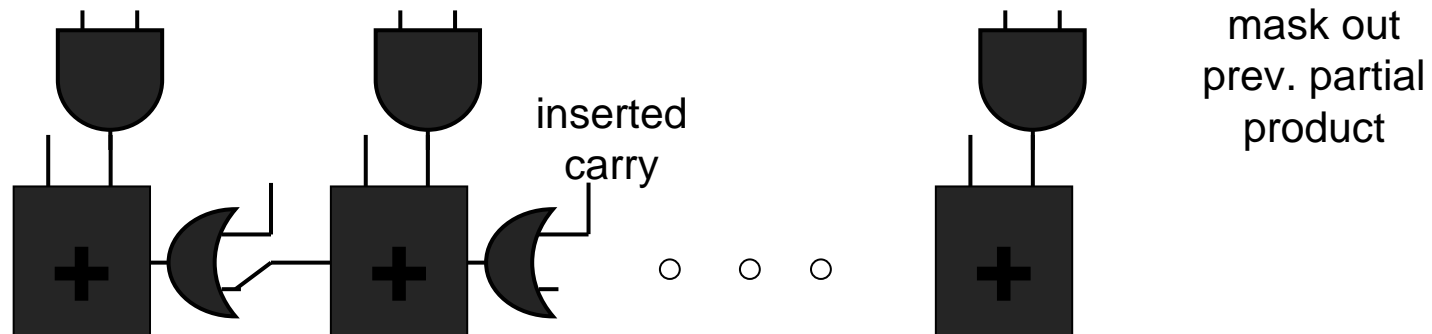
- Worst case multiplier value: 01...01 or 10...10
 - Number of operations equal to bit-width, n
 - Booth radix-4 requires $n/2$ operations (always)
- Solution
 - Modified "scan" circuit, looks one bit ahead of current discontinuity
 - If the look-ahead bit is different from last bit
 - Next discontinuity (at next bit position) ignored
 - Next operation (add/sub) is inverse of what it should be
 - Logic depth of "scan" circuit unchanged
- Guarantees at most $n/2$ operations to complete the multiplication

Booth multiplier – CSA



Variable-width CP adder (*CPA_{low}*)

- Resolves the LSB Carry-Sum output of the shifter
- Bit width ranges from 1 to n
 - Determined by the *shift_by* scan output signal
- Implemented as modified carry-ripple adder:



Booth-CSA features

- Data-dependent delay per iteration
 - Delay through CSA and shifter is *not* data dependent
 - But delay through CPA_{low} strongly depends on data values
- Variation of “speculative completion” [NoYB’97] for delay matching CPA_{low}
 - Maximum possible number of carry propagations equal to number of bits shifted in current iteration
- Carry-ripple organization is not bad!
 - At most n carry propagations for the whole operation
 - Whether it happens all at once, or little by little doesn’t matter much
 - But CPA_{high} should be fast: up to n carry propagations at the last iteration

Implementation

- Technology: UMC 0.18 μ , standard cells
 - No asynchronous cells, e.g. C-elements
- RTL-level Verilog
 - Synthesized using Synopsys tools
 - Placed & routed with Silicon Encounter
 - Effect of parasitics included in results
- Designs implemented (16 \times 16 bit) :
 - Standard Booth radix-4 iterative multiplier (Booth-r4)
 - Original Booth with CP addition (CPA)
 - CPA with look-ahead improvement (CPA-impr)
 - Original Booth with CS addition and look-ahead (CSA)
- Shifters built out of multiplexer trees
 - Design attempt using tri-state buffers too power hungry

Evaluation: delay

Design	cycle (ns)	last
CPA	2.0	0
CPA-impr	2.1	0
CSA	2.5-4.4	+0.6
Booth-r4	1.2	+0.5

- Booth-r4 lowest cycle time [\checkmark expected]
 - no shifter in critical path
- CSA longer cycle than CPA [\times unexpected]
 - "Scan" in critical path, delay through shifter increases
 - Variable-width CPA is added
- Booth-r4: $8 \times 1.2 + 0.5 = 10.1\text{ns}$
- CPA: $10.1 / 2.0 = 5$ iterations

CPA better when
#iterations < 5

Evaluation: power consumption

Design	Power	norm.
CPA	25.5	1.8
CPA-impr	23.6	1.6
CSA	22.0	1.5
Booth-r4	14.4	1.0

- Results from PowerCompiler
 - Based on test patterns, not typical multiplier inputs
- All original Booth designs consume more power
 - Shifters consume about 40% of the total

Evaluation: area

Design	Cells	Area	norm.
CPA	999	26.2	1.7
CPA-impr	1078	28.7	1.8
CSA	1949	52.1	3.3
Booth-r4	928	15.7	1.0

- All original Booth designs are larger [$\sqrt{\quad}$ expected]
 - Required shifters are large
- CSA is very large [\times unexpected]
 - Carry-sum representation doubles the shifters
 - Two extra 16-bit adders compared to CPA designs

Further work

- Full-custom implementation
 - The shifter design is greatly disadvantaged due to std cell based design
 - Performance and size of the arithmetic circuits could be considerably improved
- Pipelining
 - CPA_{low} can be placed in a different pipeline stage
- Scan circuit
 - Sequential logic implementations
- Evaluation
 - Gather typical, “real-world”, data from benchmarks
 - These will provide info on average number of iterations

Conclusions

- First asynchronous implementation of the original Booth algorithm
- Exploits data-dependency in delay, power consumption
 - In number of iterations required
 - Within each iteration
- Mixed results
 - Power significantly higher than standard implementation
 - Promising delay results
 - Full-custom shifters may improve the results
- Further work needed to optimize & evaluate circuits