Arithmetic Circuits

Prof. Taewhan Kim

(tkim@ssl.snu.ac.kr)

Sources:

1. I. Koren, “Computer Arithmetic Algorithms,’’ Naticl, Massachusetts, 2001.
2. R. Katz, “Contemporary Logic Design,” Prentice Hall, 2004.

Contents

- Number system
- Addition/subtraction
  - Ripple-carry-adder
  - Carry-lookahead-adder
  - Carry-select-adder
- ALU
- Multiplication
  - Sequential multiplication
  - Fast multiplication
    - Partial products reduction techniques
    - Constant multiplication
- Division (not covered)
- Floating point (not covered)

Number systems

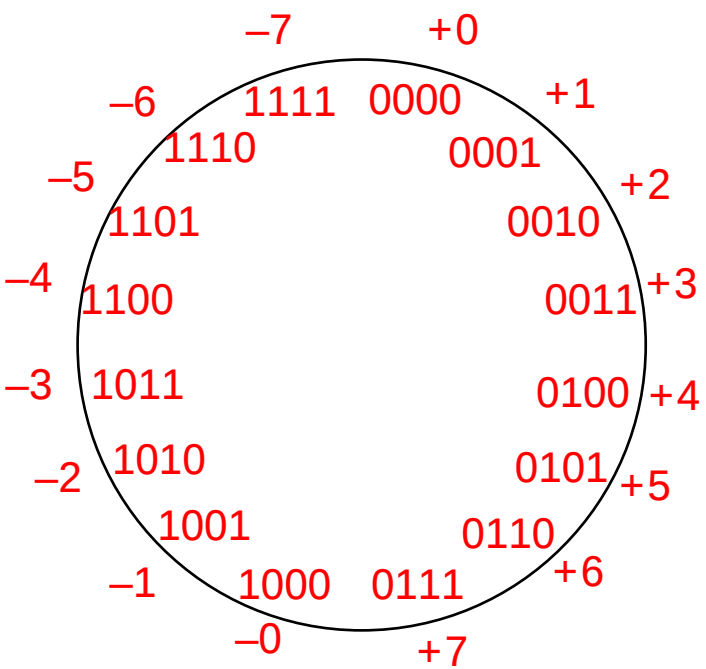
- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
  - sign and magnitude
  - 1s complement
  - 2s complement
- Assumptions
  - we'll assume a 4 bit machine word
  - 16 different values can be represented
  - roughly half are positive, half are negative

Sign and magnitude

- One bit dedicate to sign (positive or negative)
  - sign: 0 = positive (or zero), 1 = negative
- Rest represent the absolute value or magnitude
  - three low order bits: 0 (000) thru 7 (111)
- Range for n bits
  - +/– (2^{n-1} –1) (two representations for 0)
- Cumbersome addition/subtraction
  - must compare magnitudes to determine sign of result

0 100 = + 4

1 100 = – 4



1s complement

- If N is a positive number, then the negative of N (its 1s complement or N') is N' = $(2^n – 1) – N$
  - example: 1s complement of 7

$$
\begin{aligned}
2^4 &= 10000 \\
1 &= 00001 \\
\hline
2^4 – 1 &= 1111 \\
7 &= 0111 \\
\hline
&1000 = –7 \text{ in 1s complement form}
\end{aligned}
$$

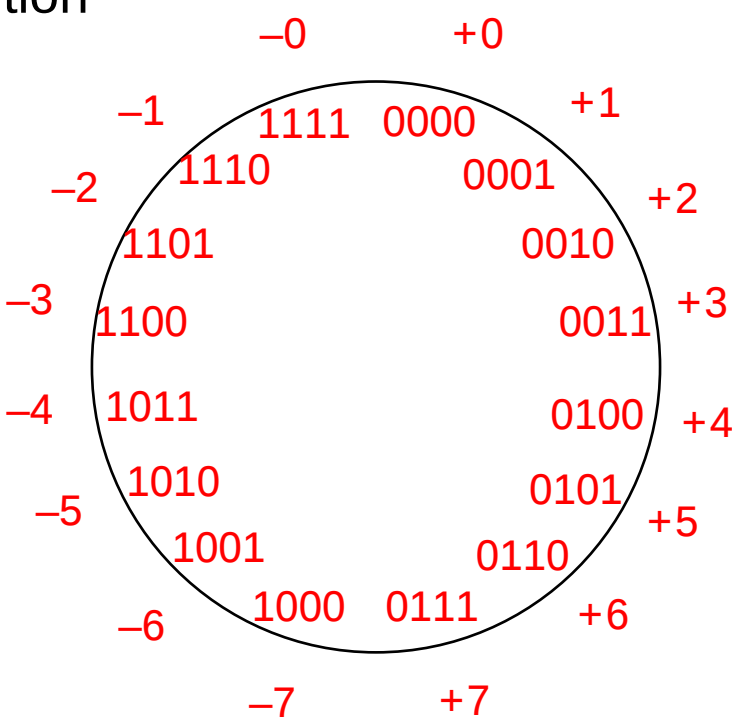
  - shortcut: simply compute bit-wise complement (0111 -> 1000)

1s complement (cont'd)

- Subtraction implemented by 1s complement and then addition
- Two representations of 0
  - causes some complexities in addition
- High-order bit can act as sign bit

0 100 = + 4

1 011 = – 4

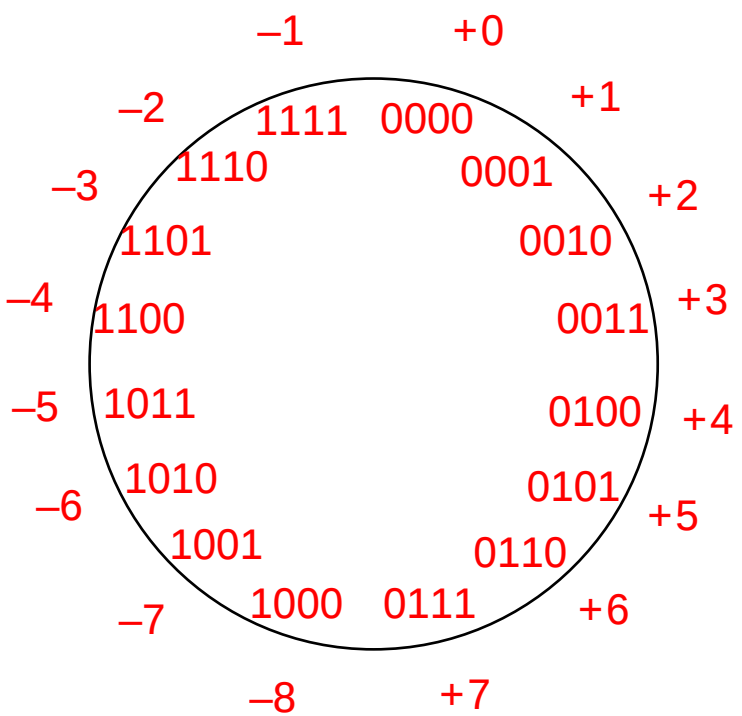


2s complement

- 1s complement with negative numbers shifted one position clockwise
  - only one representation for 0
  - one more negative number than positive numbers
  - high-order bit can act as sign bit

0 100 = + 4

1 100 = – 4



2s complement (cont'd)

- If N is a positive number, then the negative of N (its 2s complement or N*) is $N^* = 2^n - N$
  - example: 2s complement of 7

$$2^4 = 10000$$
$$\text{subtract } 7 = 0111$$
$$1001 = \text{repr. of } -7$$

  - example: 2s complement of –7

$$2^4 = 10000$$
$$\text{subtract } -7 = 1001$$
$$0111 = \text{repr. of } 7$$

  - shortcut: 2s complement = bit-wise complement + 1
    - 0111 -> 1000 + 1 -> 1001 (representation of -7)
    - 1001 -> 0110 + 1 -> 0111 (representation of 7)

2s complement addition and subtraction

- Simple addition and subtraction
  - simple scheme makes 2s complement the virtually unanimous choice for integer number systems in computers

4	0100	– 4	1100
+ 3	0011	+ (– 3)	1101
7	0111	– 7	11001

4	0100	– 4	1100
– 3	1101	+ 3	0011
1	10001	– 1	1111

Why can the carry-out be ignored?

- Can't ignore it completely
  - needed to check for overflow (see next two slides)
- When there is no overflow, carry-out may be true but can be ignored

  – M + N when N > M:

  $M^* + N = (2^n - M) + N = 2^n + (N - M)$

  ignoring carry-out is just like subtracting 2^n

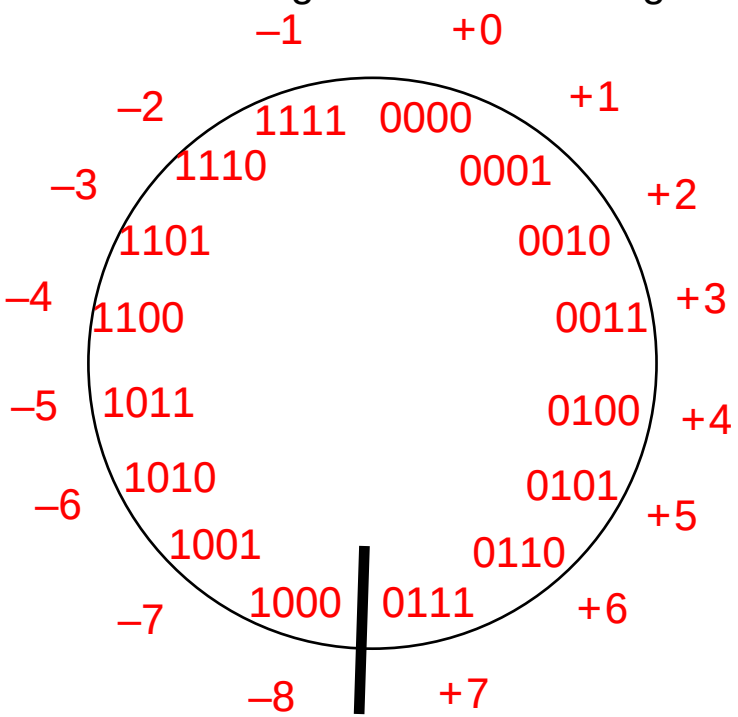
  – M + – N where $N + M \leq 2^{n-1}$

  $(-M) + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = 2^n - (M + N) + 2^n$

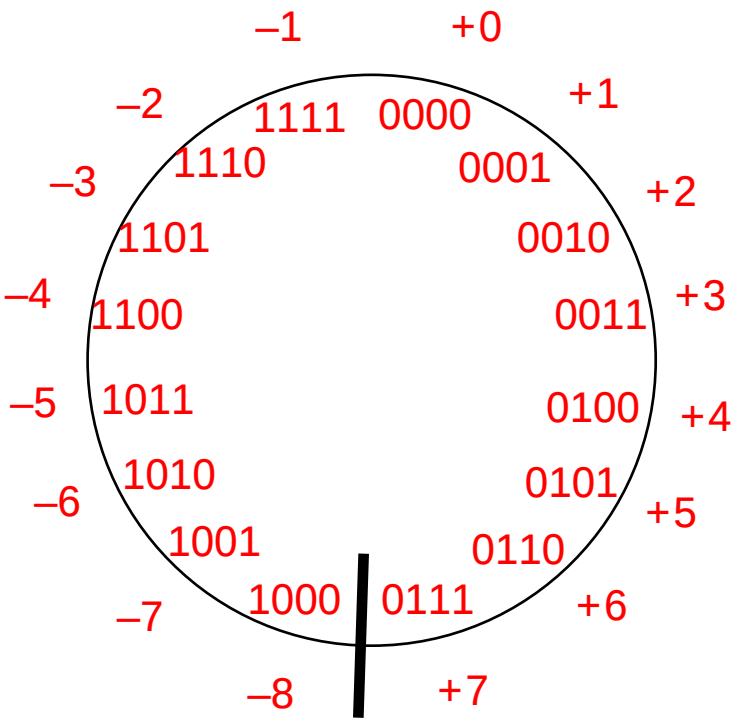
  ignoring the carry, it is just the 2s complement representation for – (M + N)

Overflow in 2s complement addition/subtraction

- Overflow conditions
  - add two positive numbers to get a negative number
  - add two negative numbers to get a positive number



5 + 3 = –8



–7 – 2 = +7

Overflow conditions

- Overflow when carry into sign bit position is not equal to carry-out

```
  0 1 1 1
    0 1 0 1        5
    0 0 1 1        3
    1 0 0 0       -8
```

overflow

```
  1 0 0 0
    1 0 0 1       -7
    1 1 1 0       -2
  1 0 1 1 1        7
```

overflow

```
  0 0 0 0
    0 1 0 1        5
    0 0 1 0        2
    0 1 1 1        7
```

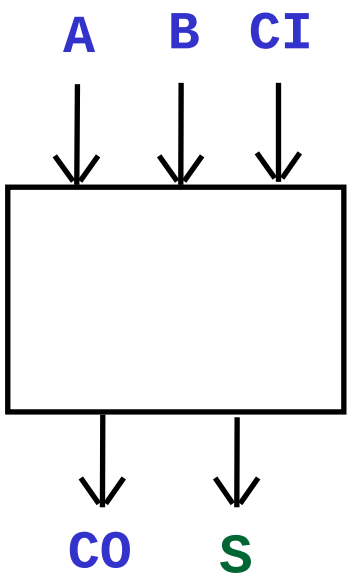
no overflow

```
  1 1 1 1
    1 1 0 1       -3
    1 0 1 1       -5
  1 1 0 0 0       -8
```

no overflow

Addition: binary addition

- This is the primitive of almost all arithmetic computation.



Truth Table:

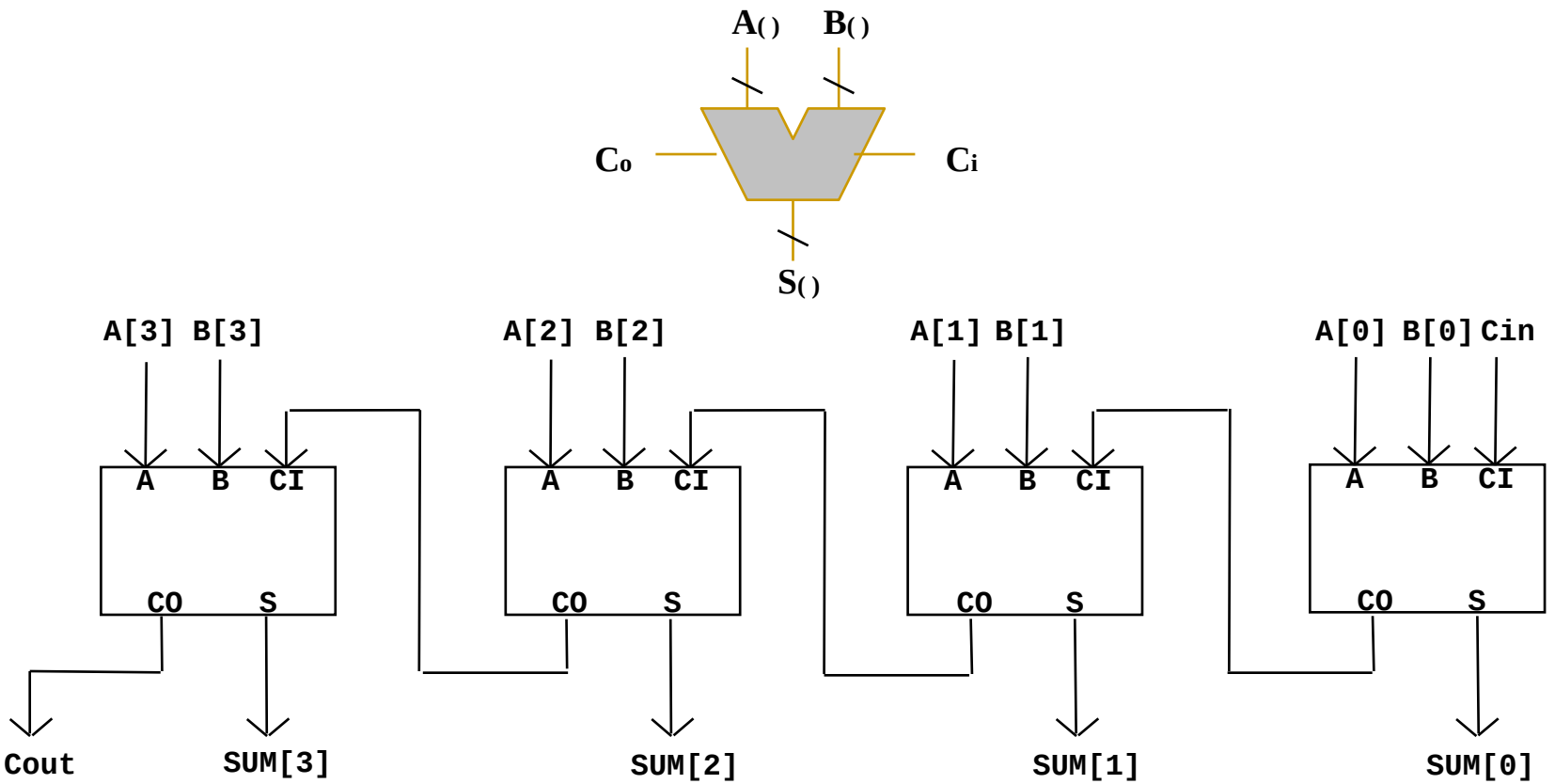
A	B	CI	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$CO = A \cdot B + B \cdot CI + CI \cdot A$$

$$S = A \oplus B \oplus CI$$

A 4-Bit Ripple-Carry Adder (RCA)

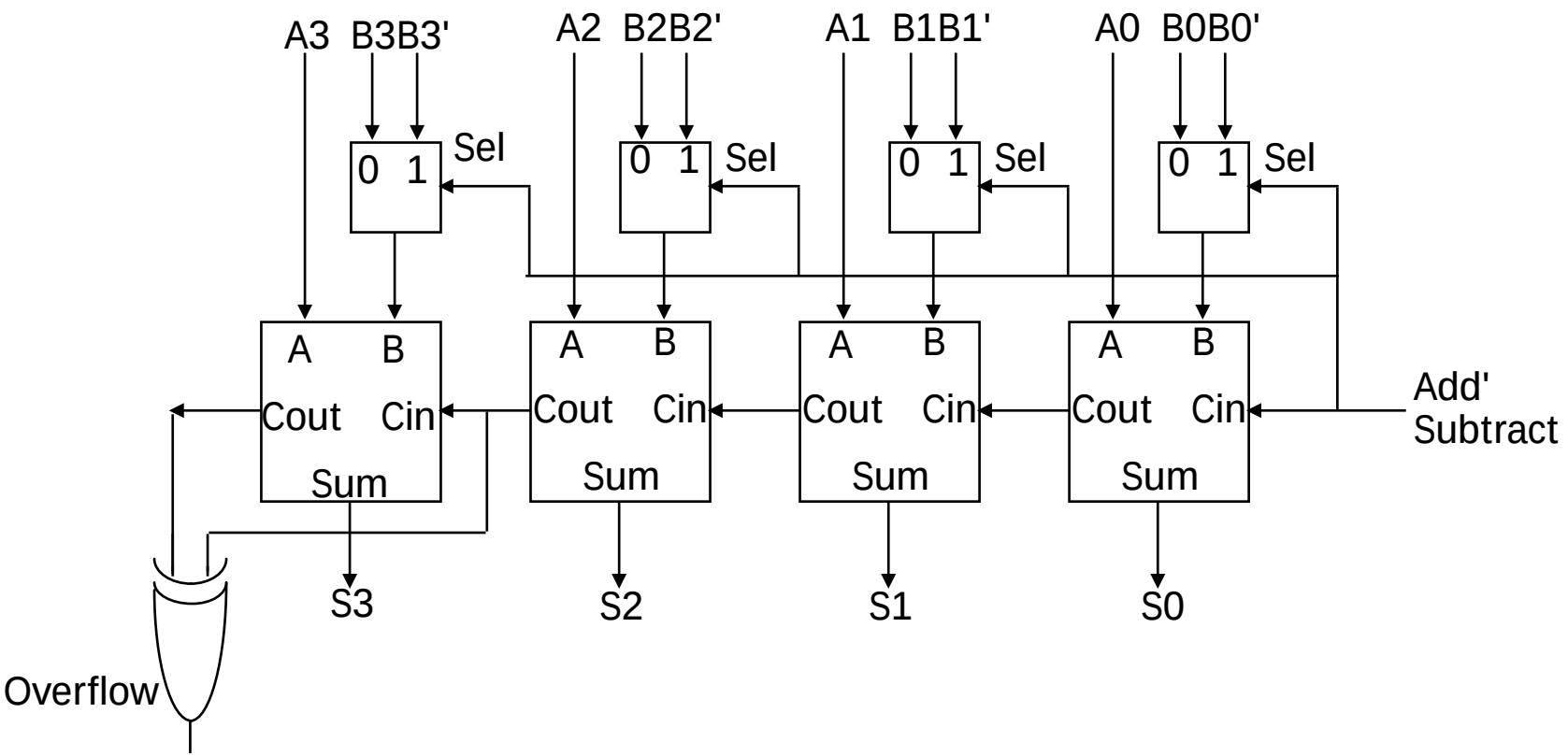
- Note: The carry chain ripples from the least to the most significant bit (LSB to MSB).



Area = (bit-width) * (area of a one-bit full-adder cell)
Delay = (bit-width) * (delay of a one-bit full-adder cell)

Adder/subtractor

- Use an adder to do subtraction thanks to 2s complement representation
  - A – B = A + (– B) = A + B' + 1
  - control signal selects B or 2s complement of B



Carry-lookahead logic

- Carry generate: Gi = Ai Bi
  - must generate carry when A = B = 1
- Carry propagate: Pi = Ai xor Bi
  - carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai + Bi)
    = Ai Bi + Ci (Ai xor Bi)
    = Gi + Ci Pi

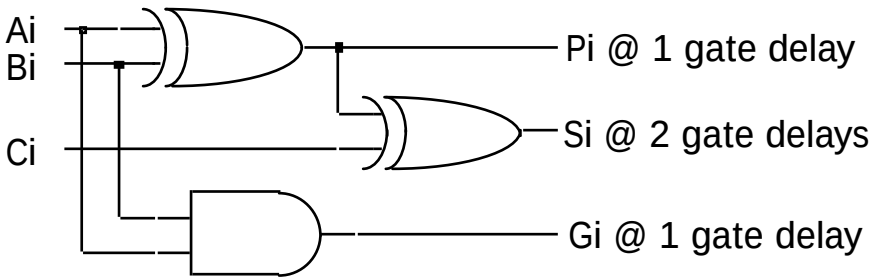
Carry-lookahead logic (cont'd)

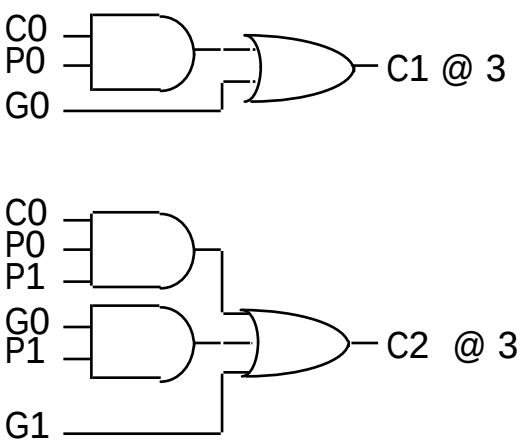
- Re-express the carry logic as follows:
  - C1 = G0 + P0 C0
  - C2 = G1 + P1 C1 = G1 + P1 G0 + P1 P0 C0
  - C3 = G2 + P2 C2 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 C0
  - C4 = G3 + P3 C3 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0 + P3 P2 P1 P0 C0

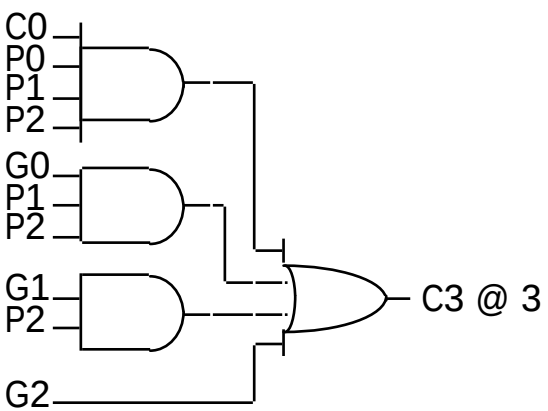
- Each of the carry equations can be implemented with two-level logic
  - all inputs are now directly derived from data inputs and not from intermediate carries
  - this allows computation of all sum outputs to proceed in parallel

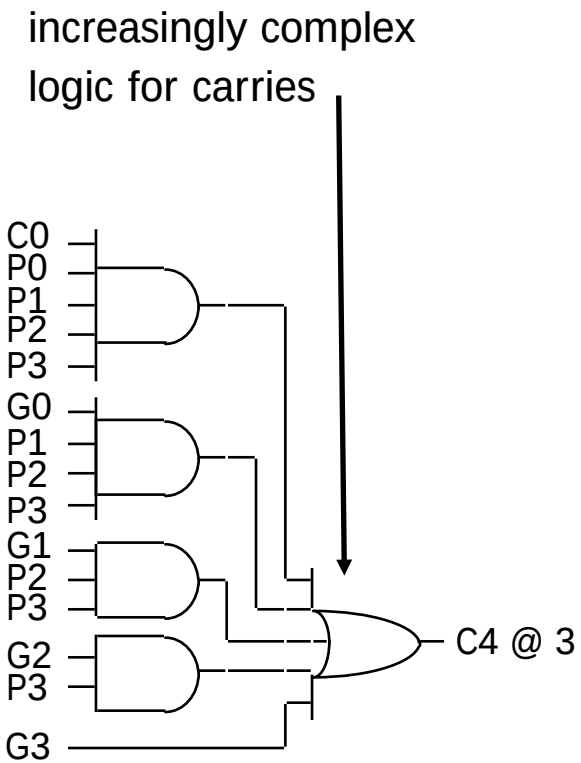
Carry-lookahead implementation

- Adder with propagate and generate outputs

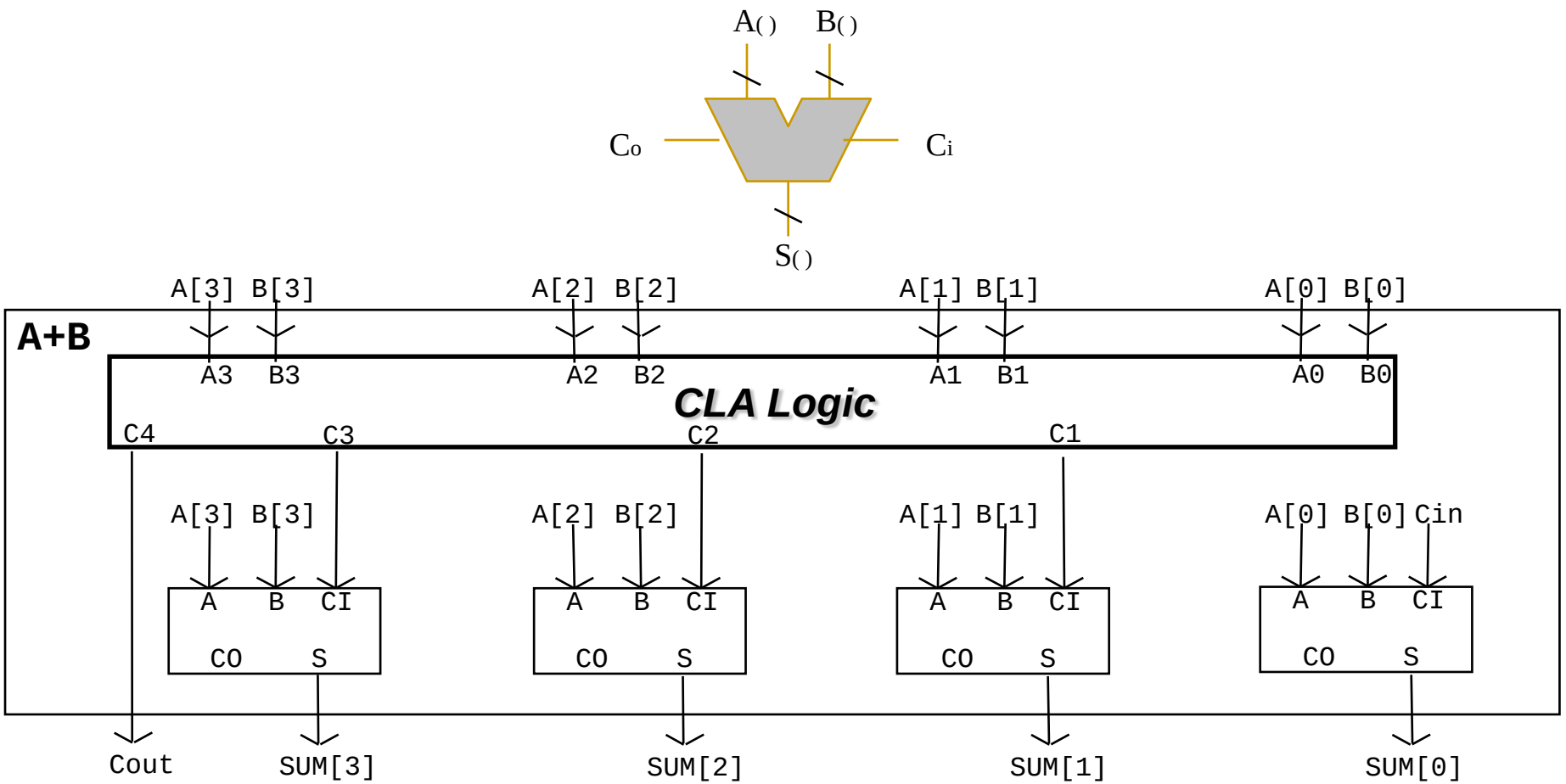








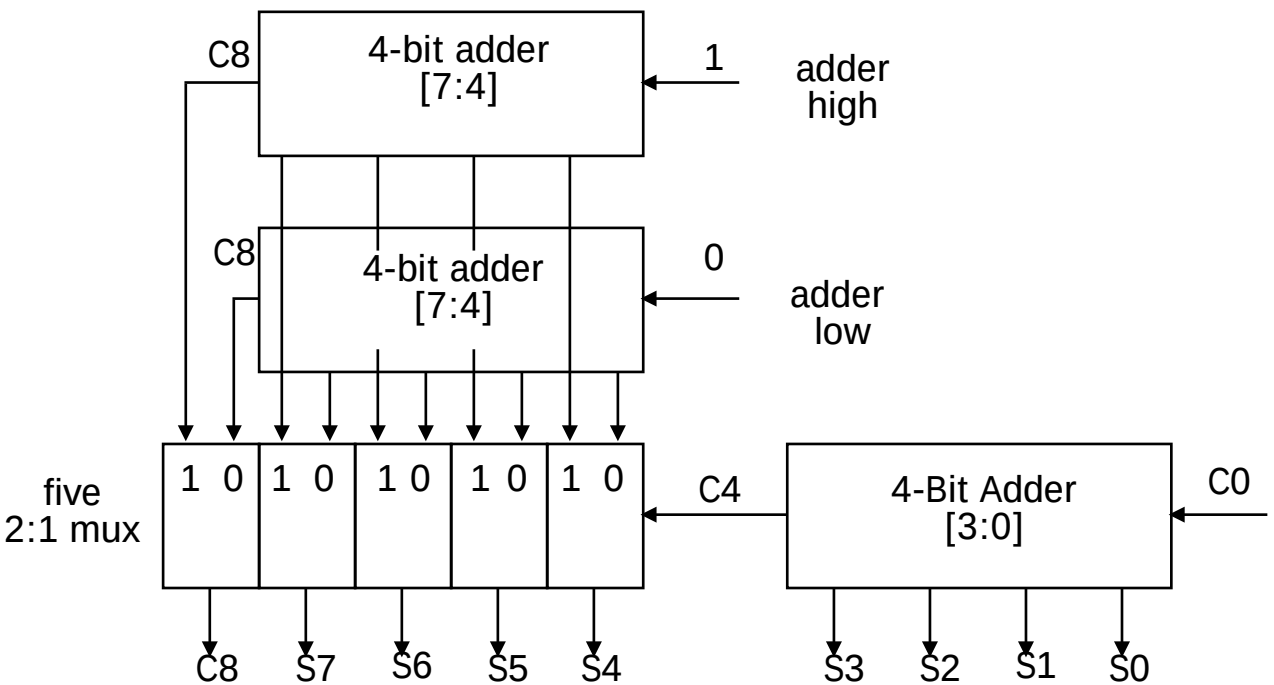
4-Bit Carry Look Ahead (CLA) Adder



Area = (CLA Logic Area) + ((bit-width) * (area of a one-bit full-adder cell))
= about(1 + Log_2(bit-width)) * (area of ripple adder)
Delay = about(1 + Log_2(bit-width)) * (delay of a one-bit full-adder cell)

Carry-select adder

- Redundant hardware to make carry calculation go faster
  - compute two high-order sums in parallel while waiting for carry-in
  - one assuming carry-in is 0 and another assuming carry-in is 1
  - select correct result once carry-in is finally computed



Arithmetic logic unit (ALU) design specification

M = 0, logical bitwise operations

S1	S0	Function	Comment
0	0	Fi = Ai	input Ai transferred to output
0	1	Fi = not Ai	complement of Ai transferred to output
1	0	Fi = Ai xor Bi	compute XOR of Ai, Bi
1	1	Fi = Ai xnor Bi	compute XNOR of Ai, Bi

M = 1, C0 = 0, arithmetic operations

0	0	F = A	input A passed to output
0	1	F = not A	complement of A passed to output
1	0	F = A plus B	sum of A and B
1	1	F = (not A) plus B	sum of B and complement of A

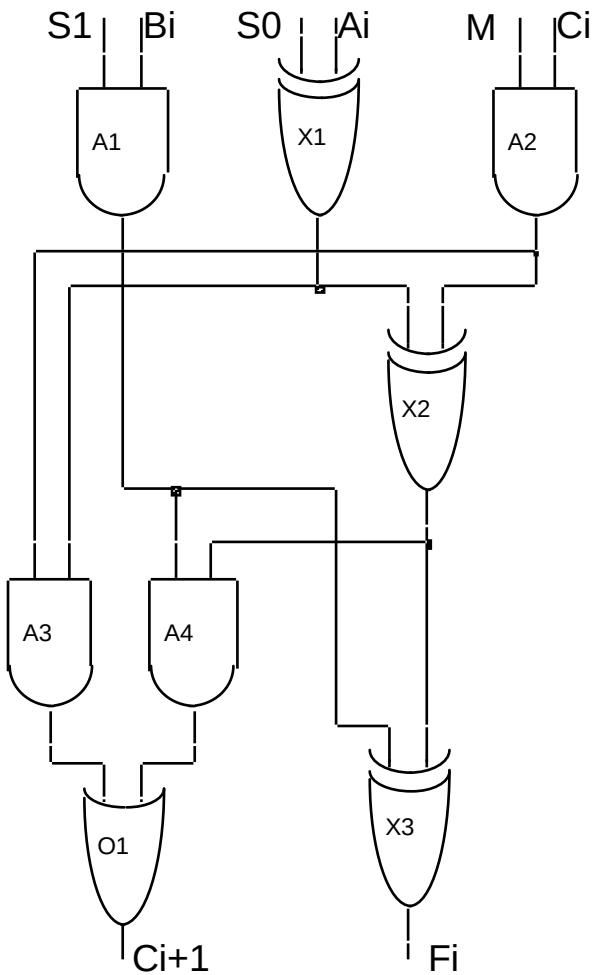
M = 1, C0 = 1, arithmetic operations

0	0	F = A plus 1	increment A
0	1	F = (not A) plus 1	twos complement of A
1	0	F = A plus B plus 1	increment sum of A and B
1	1	F = (not A) plus B plus 1	B minus A

logical and arithmetic operations
not all operations appear useful, but "fall out" of internal logic

ALU design

- Sample ALU –multi-level implementation

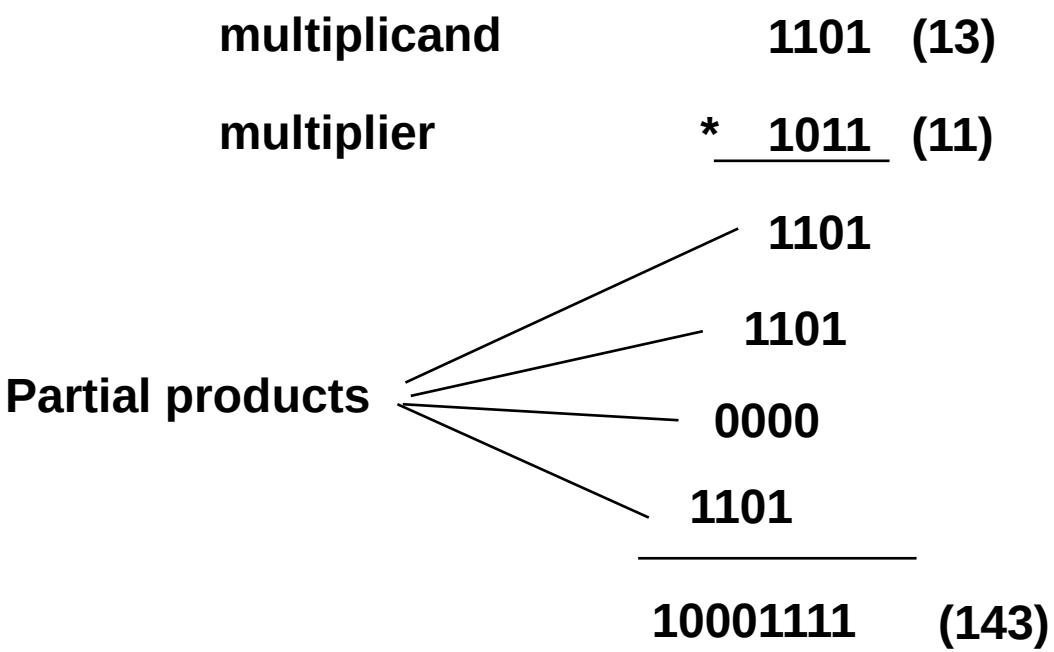


first-level gates
  use S0 to complement Ai
    S0 = 0 causes gate X1 to pass Ai
    S0 = 1 causes gate X1 to pass Ai'
  use S1 to block Bi
    S1 = 0 causes gate A1 to make Bi go forward as 0
            (don't want Bi for operations with just A)
    S1 = 1 causes gate A1 to pass Bi
  use M to block Ci
    M = 0 causes gate A2 to make Ci go forward as 0
           (don't want Ci for logical operations)
    M = 1 causes gate A2 to pass Ci

other gates
  for M=0 (logical operations, Ci is ignored)
    Fi = S1 Bi xor (S0 xor Ai)
       = S1'S0' (Ai) + S1'S0 (Ai') +
          S1 S0' (Ai Bi' + Ai' Bi) + S1 S0 (Ai' Bi' + Ai Bi)
  for M=1 (arithmetic operations)
    Fi = S1 Bi xor ((S0 xor Ai) xor Ci) =
    Ci+1 = Ci (S0 xor Ai) + S1 Bi ((S0 xor Ai) xor Ci) =

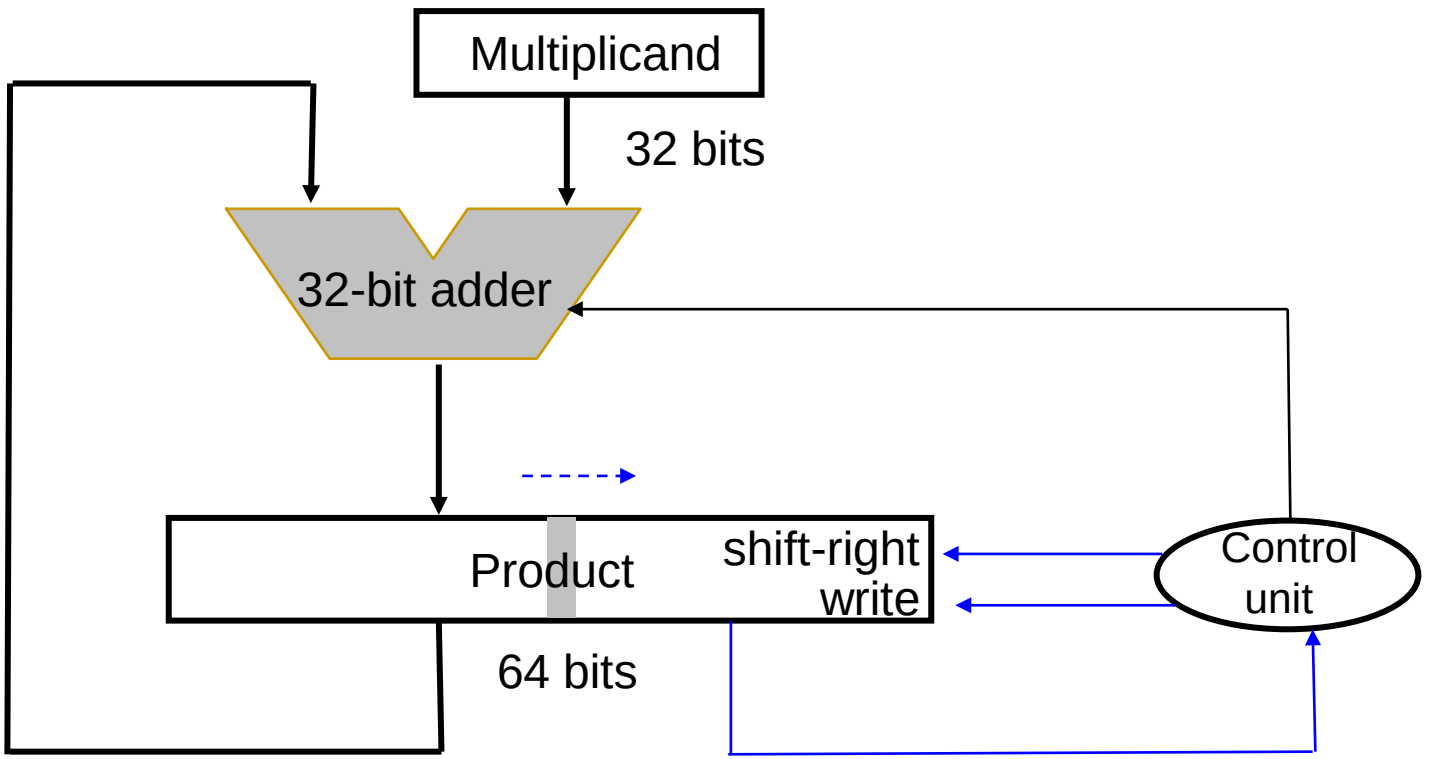
    just a full adder with inputs S0 xor Ai, S1 Bi, and Ci

Multiplication



product of 2 4-bit numbers is an 8-bit number

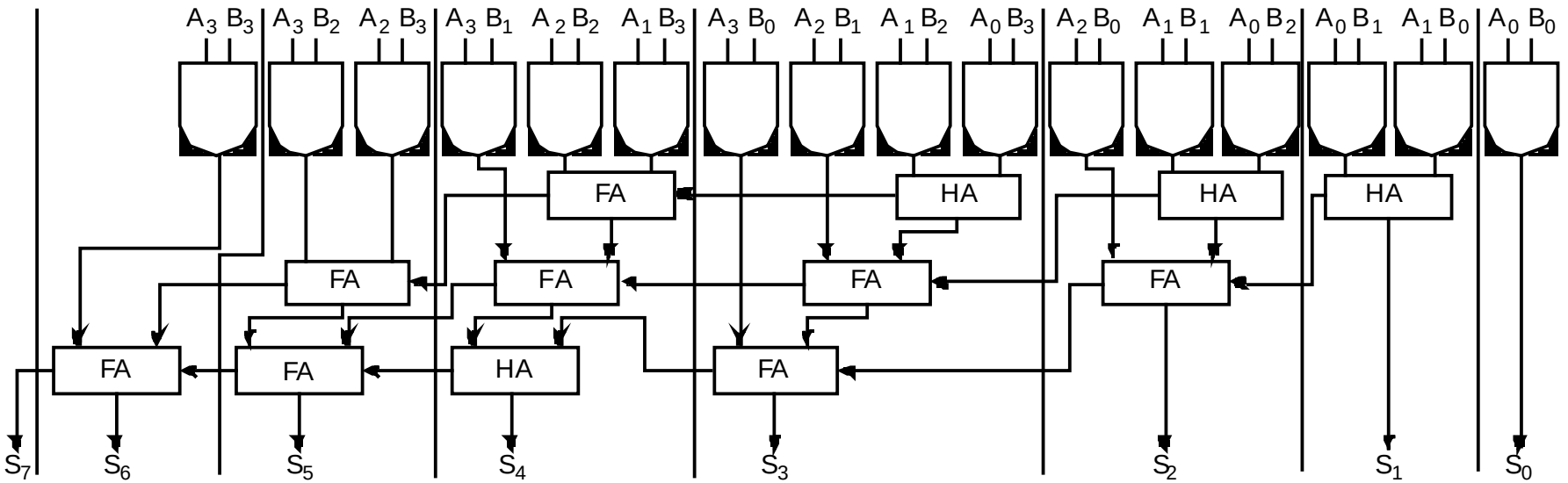
Sequential Multiplier



Partition Products

Partial Product Accumulation				A3	A2	A1	A0
				B3	B2	B1	B0
				A2 B0	A2 B0	A1 B0	A0 B0
			A3 B1	A2 B1	A1 B1	A0 B1	
		A3 B2	A2 B2	A1 B2	A0 B2		
	A3 B3	A2 B3	A1 B3	A0 B3			
S7	S6	S5	S4	S3	S2	S1	S0

Reduction Example



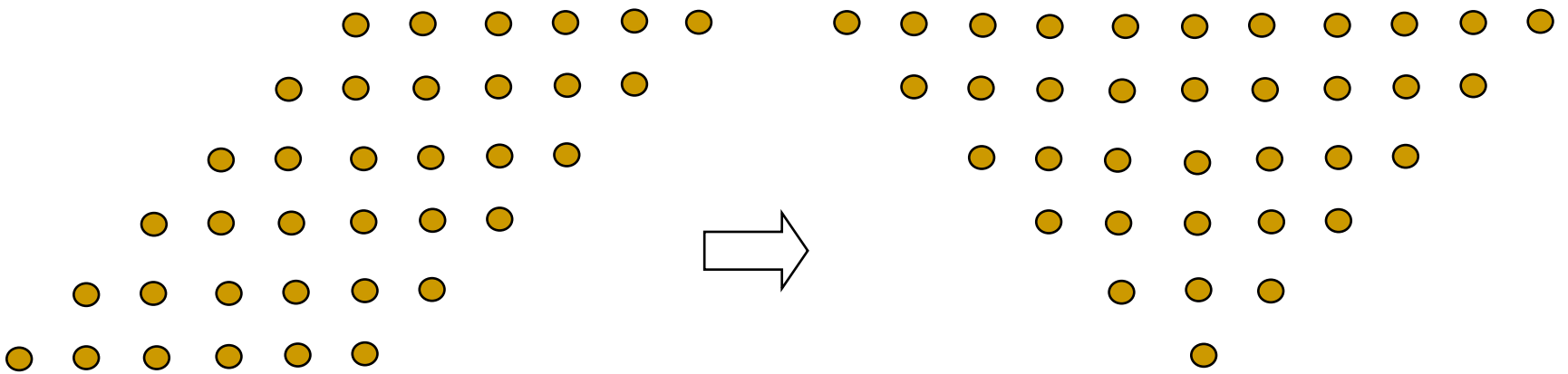
Note use of parallel carry-outs to form higher order sums

12 Adders, if full adders, this is 6 gates each = 72 gates

16 gates form the partial products

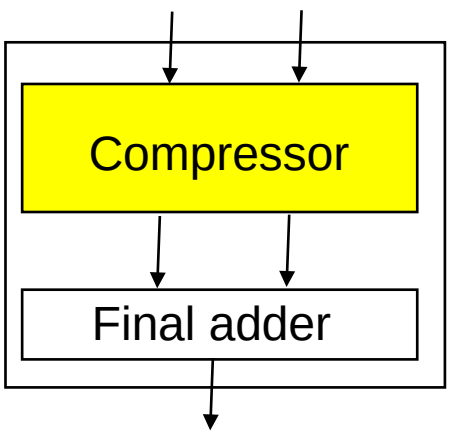
total = 88 gates!

Example: Six partial products to be added



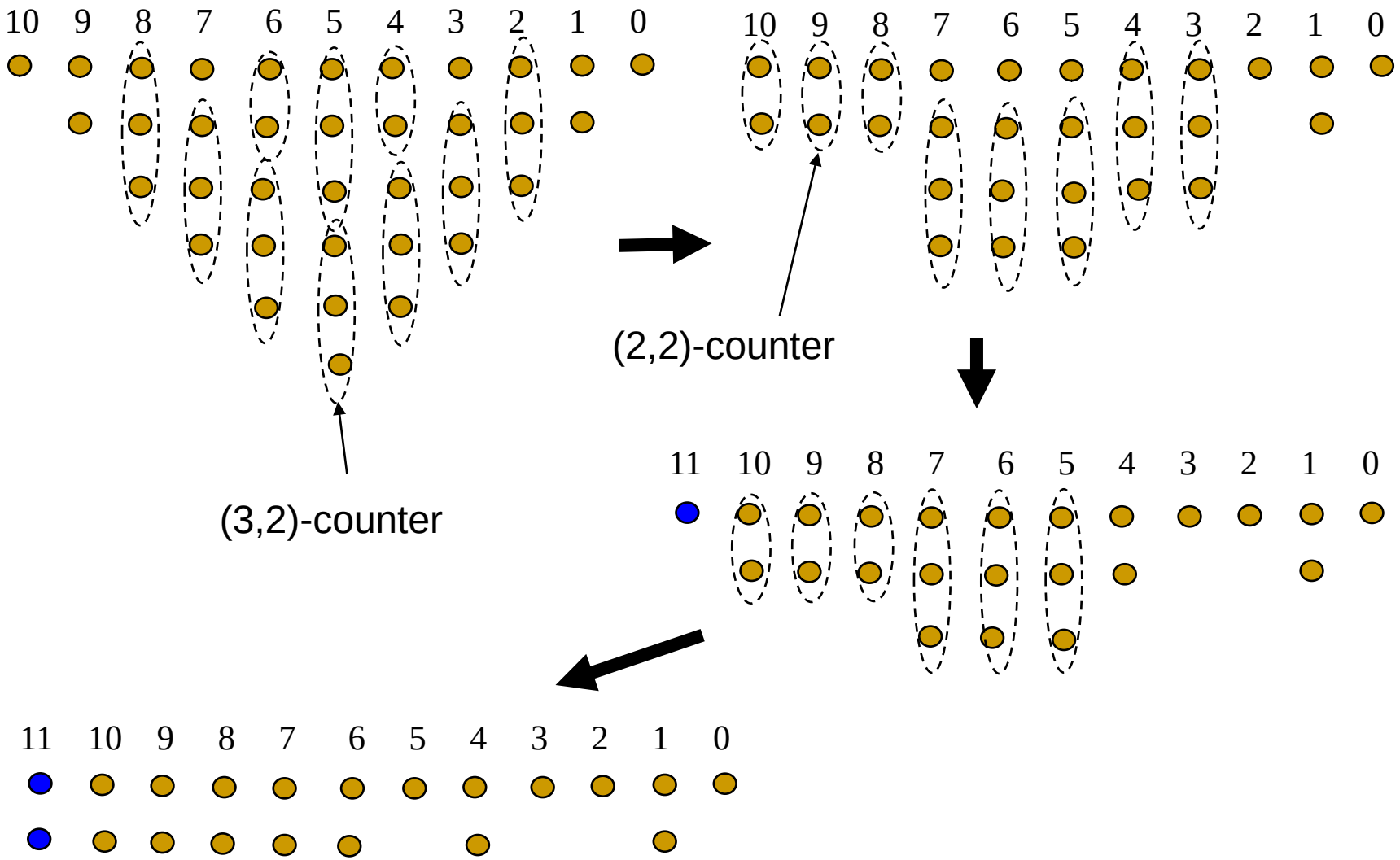
Original matrix of 36-bits

Reorganized matrix of bits

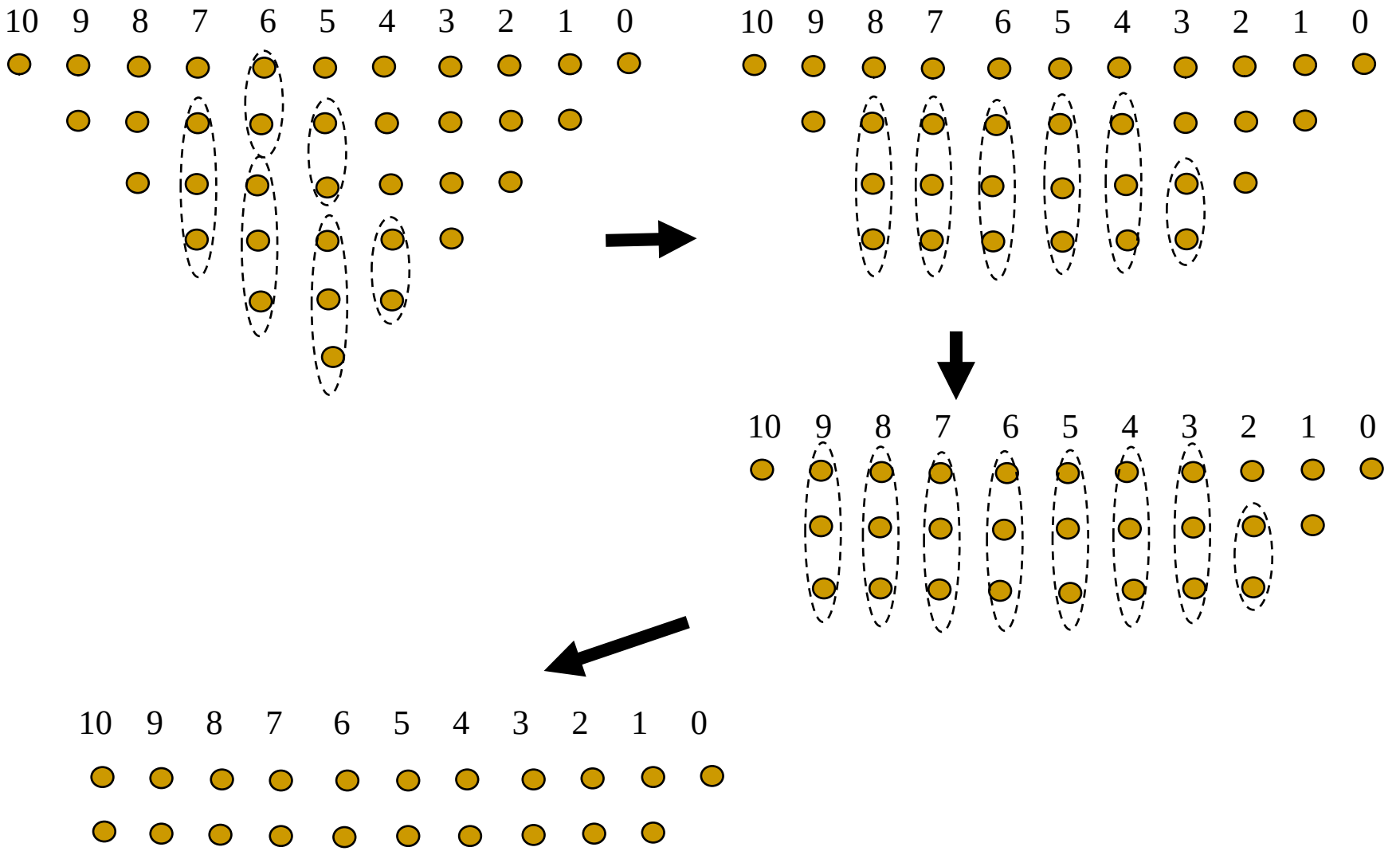


Parallel multiplier

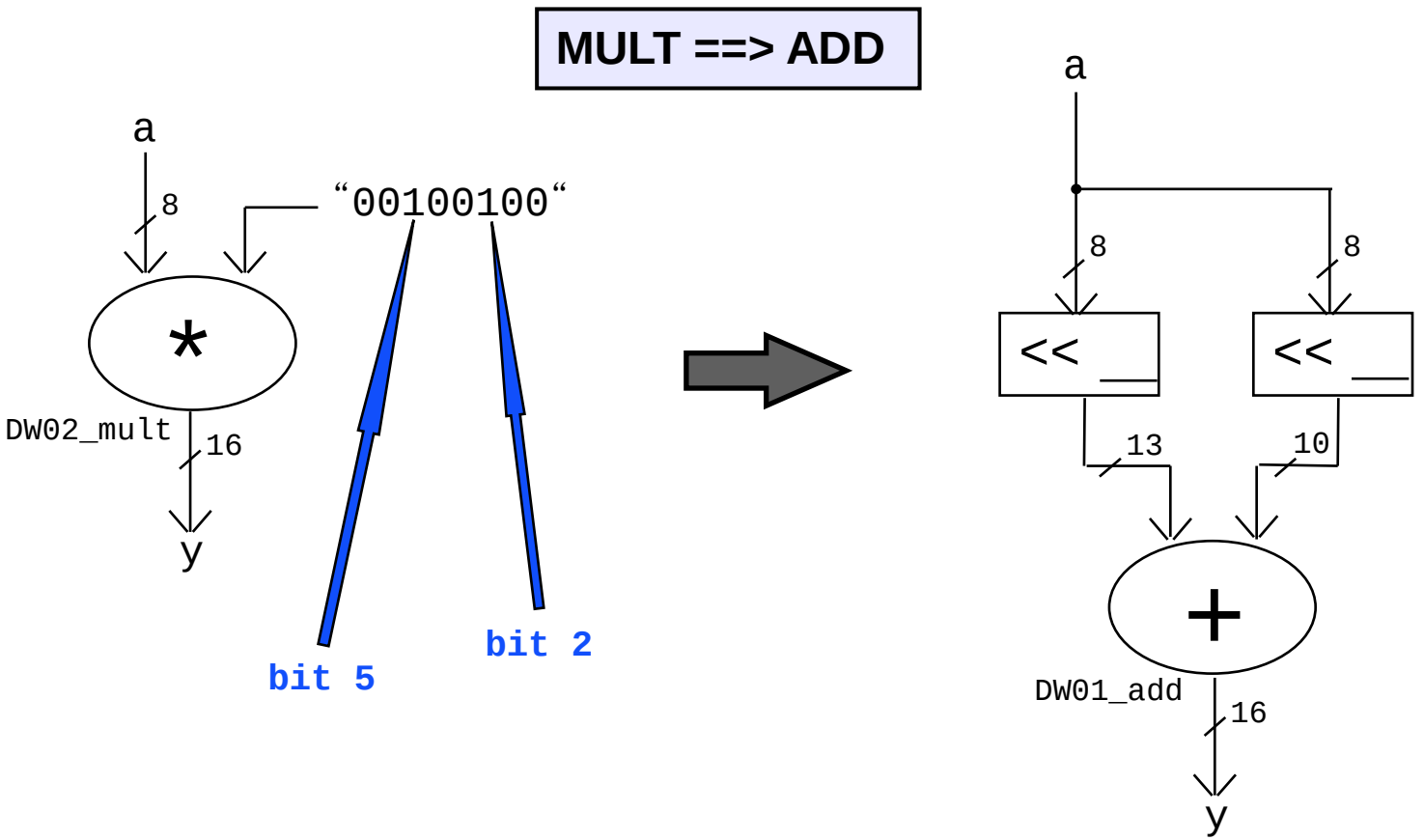
Reduction of the six partial products (Wallace)



Reduction of the six partial products (Dadda)



Constant Multiplication

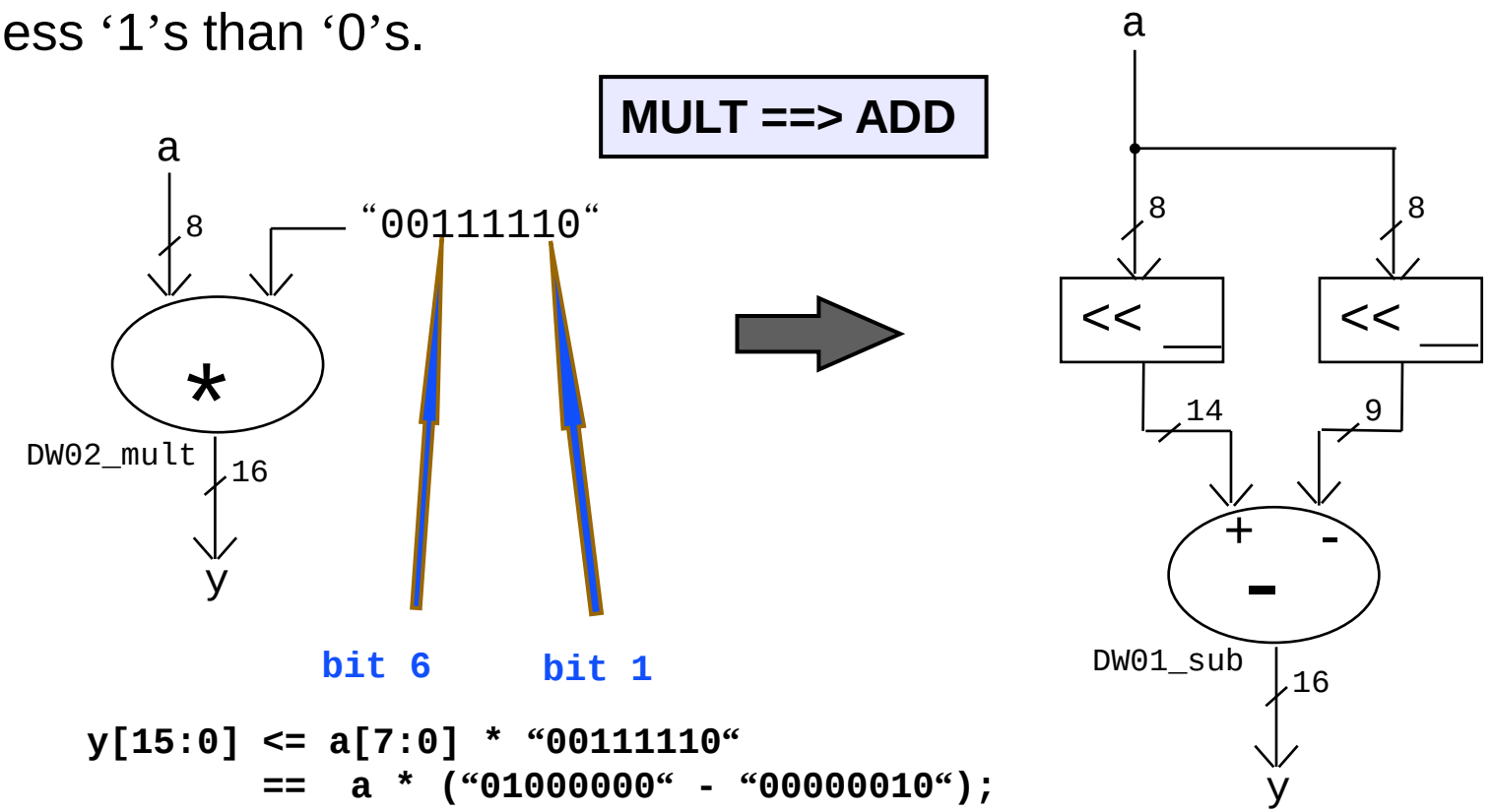



```
y[15:0] <= a[7:0] * "00100100"
        == (a[7:0] << 5) + (a[7:0] << 2)
```

Constant multiplication by shift-and-add

Canonical Encoding

Canonical encoding transforms a constant [vector] such that it contains less ‘1’s than ‘0’s.




```
y[15:0] <= a[7:0] * “00111110“
        ==  a * (“01000000“ - “00000010“);
        ==  a * “01000000” - a * “00000010“;
        == (a << 6) - (a << 1);
        == (a << 6) + ~(a << 1) + 1;
```

Canonical Encoding for constant multiplication