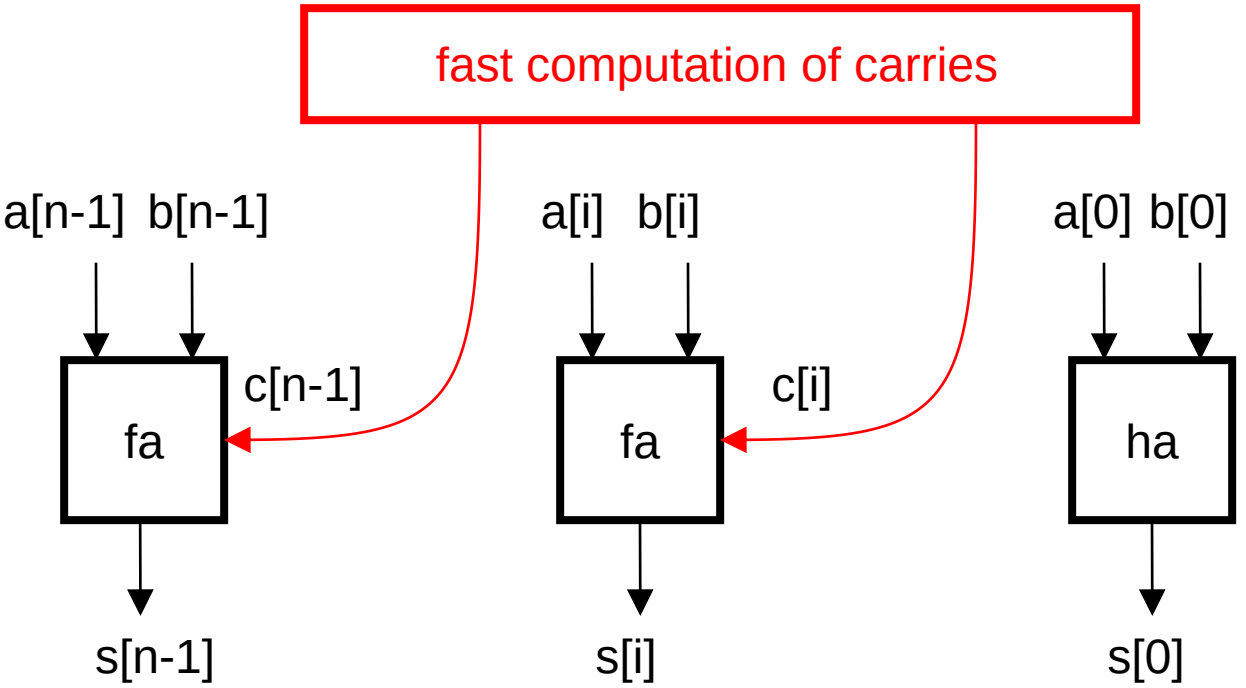
Parallel adder
Iterative structure

carry lookahead adder



- $s_i = a_i \oplus b_i \oplus c_i$

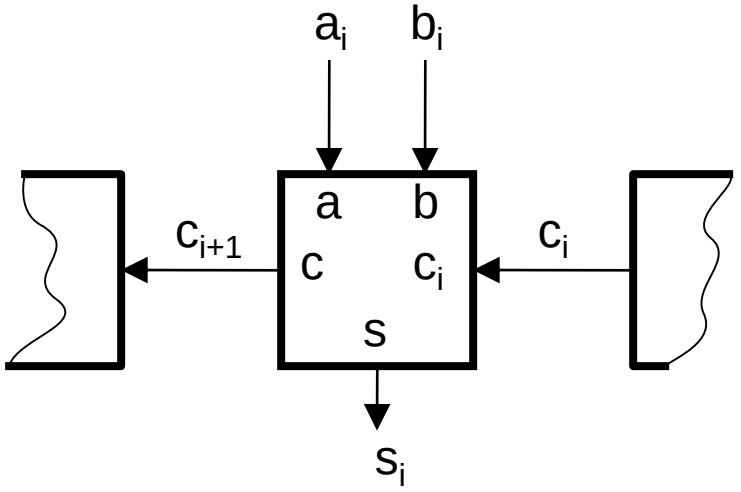
carry computation

- $c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$

  $c_{i+1} = G_i + P_i \cdot c_i$

  generate: $G_i = a_i \cdot b_i$

  propagate: $P_i = a_i \oplus b_i$



- The carry that goes into the s_{i+1} computation, c_{i+1}, is either generated by the previous pair of bits, a_i and b_i, or propagated by them from the c_i output.
- $s_i = a_i \oplus b_i \oplus c_i = P_i \oplus c_i$

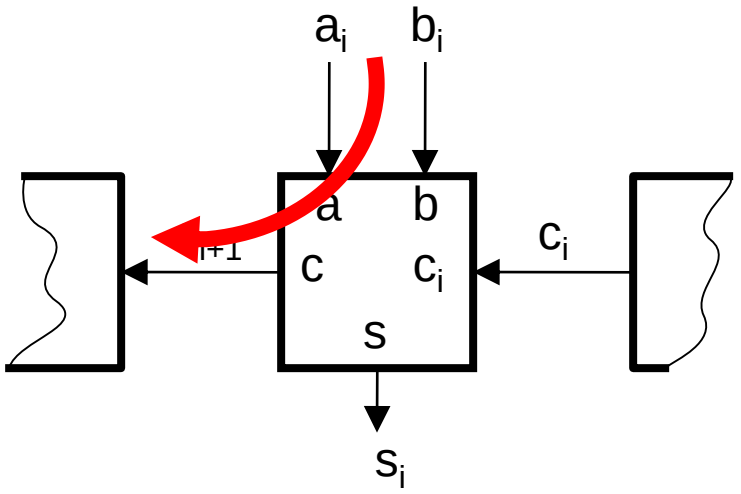
carry computation

- $c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$

  $c_{i+1} = G_i + P_i \cdot c_i$

  generate: $G_i = a_i \cdot b_i$

  propagate: $P_i = a_i \oplus b_i$



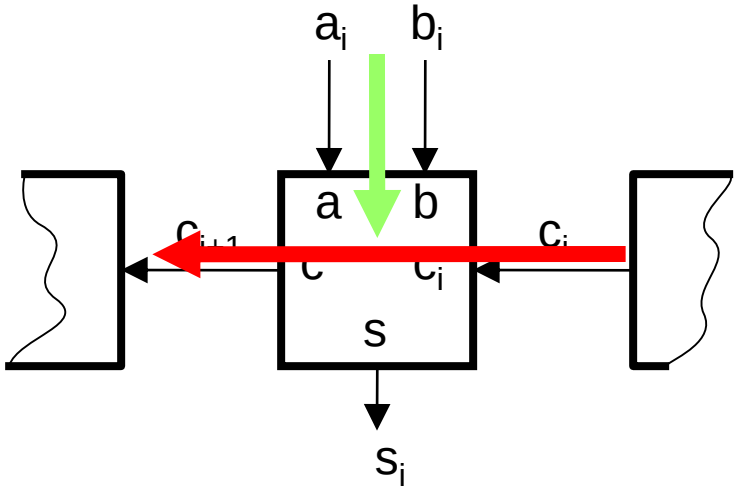
carry computation

- $c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$

  $c_{i+1} = G_i + P_i \cdot c_i$

  generate: $G_i = a_i \cdot b_i$

  propagate: $P_i = a_i \oplus b_i$



carry computation

- $c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$

  $c_{i+1} = G_i + P_i \cdot c_i$

  generate: $G_i = a_i \cdot b_i$

  propagate: $P_i = a_i \oplus b_i$

the input stage bit slice that you will use later

- $c_1 = G_0 + P_0 c_i$
- $c_2 = G_1 + P_1 c_1$

carry computation

- $c_1 = G_0 + P_0 c_i$
- $c_2 = G_1 + P_1 c_1 = \underbrace{G_1 + P_1 G_0}_{G_{10}} + \underbrace{P_1 P_0}_{P_{10}} c_i$
- c_2 is computed not based on c_1, but directly from propagate and generate bits.

carry computation

- $c_{i+1} = G_{im} + P_{im} \cdot c_m$

  G_{im} is the carry that should be generated by the addition of $[a_i ... a_m]$ with $[b_i ... b_m]$.

  P_{im} propagates any carry that would come from the addition of the last i-1 bits.
- $c_1 = G_0 + P_0 c_i$
- $c_2 = G_1 + P_1 c_1 = G_1 + P_1 G_0 + P_1 P_0 c_i = G_{10} + P_{10} c_i$
- $c_3 = G_2 + P_2 c_2$
- $c_4 = G_3 + P_3 c_3 = G_3 + P_3 G_2 + P_3 P_2 c_2 = G_{32} + P_{32} c_2$

carry computation

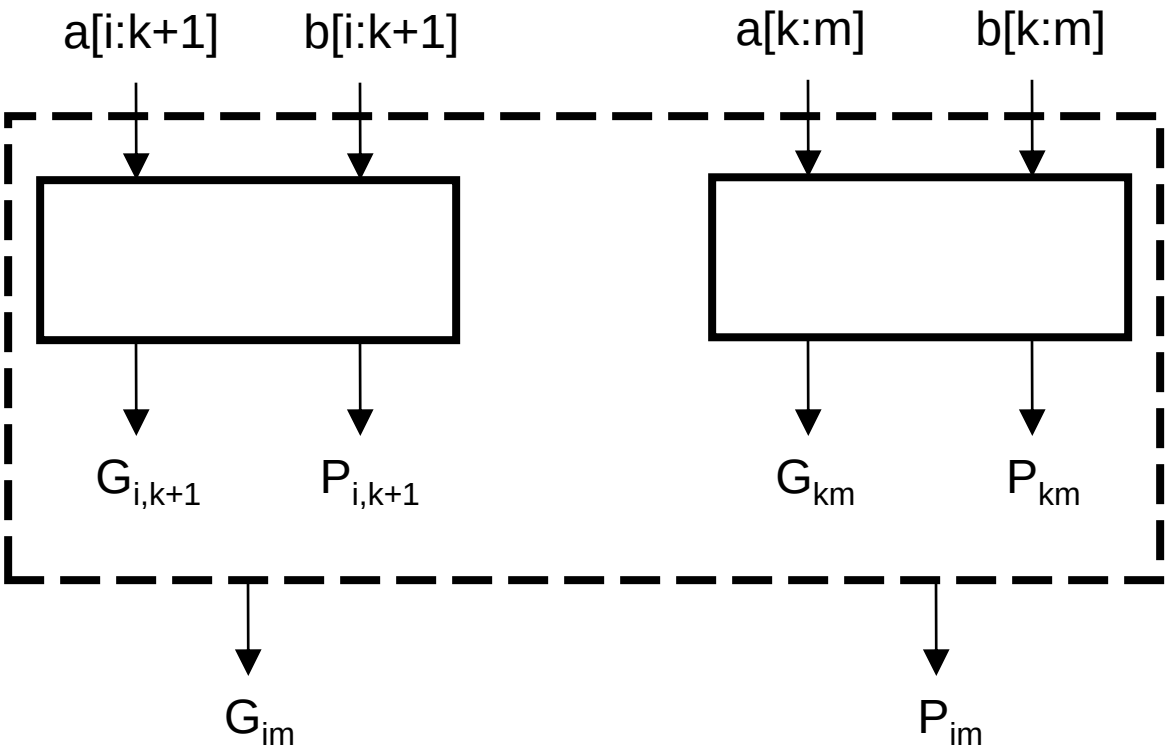
- $c_{i+1} = G_{im} + P_{im} \cdot c_m$

  the circle block
  that you will use later

  G_{im} is the carry that should be generated by the addition of $[a_i ... a_m]$ with $[b_i ... b_m]$.

  P_{im} propagates any carry that would come from the addition of the last i-1 bits.
- $c_1 = G_0 + P_0 c_i$
- $c_2 = G_1 + P_1 c_1 = G_1 + P_1 G_0 + P_1 P_0 c_i = G_{10} + P_{10} c_i$
- $c_3 = G_2 + P_2 c_2$
- $c_4 = G_3 + P_3 c_3 = G_3 + P_3 G_2 + P_3 P_2 c_2 = G_{32} + P_{32} c_2$
  $= G_{32} + P_{32} G_{10} + P_{10} c_i = G_{30} + P_{30} c_i$

generate and propagate composition



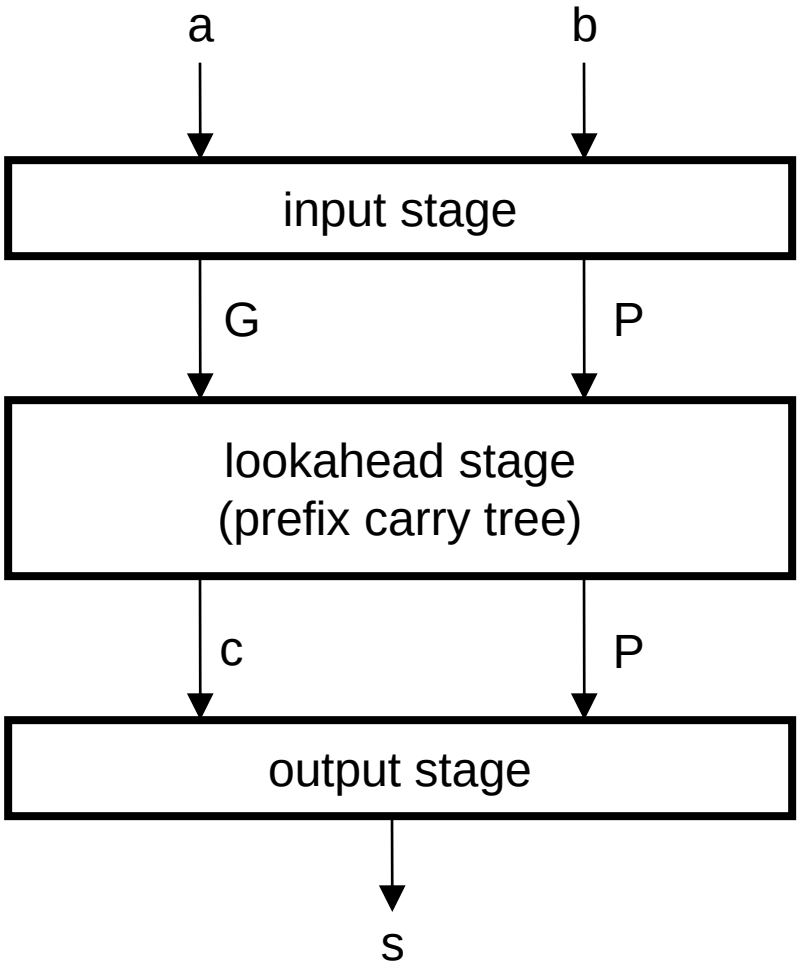
generate and propagate composition

- $c_{i+1} = G_{i,k+1} + P_{i,k+1}c_{k+1} = G_{i,k+1} + P_{i,k+1}(G_{km} + P_{km}c_m) =$
  $G_{i,k+1} + \underbrace{P_{i,k+1}G_{km}}_{G_{im}} + \underbrace{P_{i,k+1}P_{km}}_{P_{im}}c_m$

- $G_{im} = G_{i,k+1} + P_{i,k+1}G_{km}$
- $P_{im} = P_{i,k+1}P_{km}$

the square block
that you will use later

carry lookahead adder

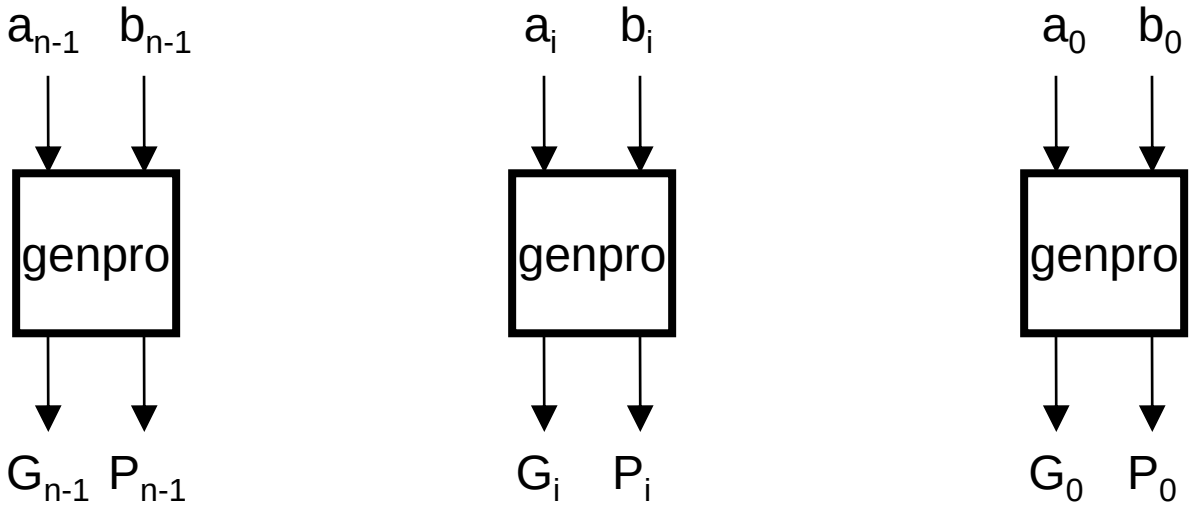


input stage

- bit sliced

  $G_i = a_i \cdot b_i$

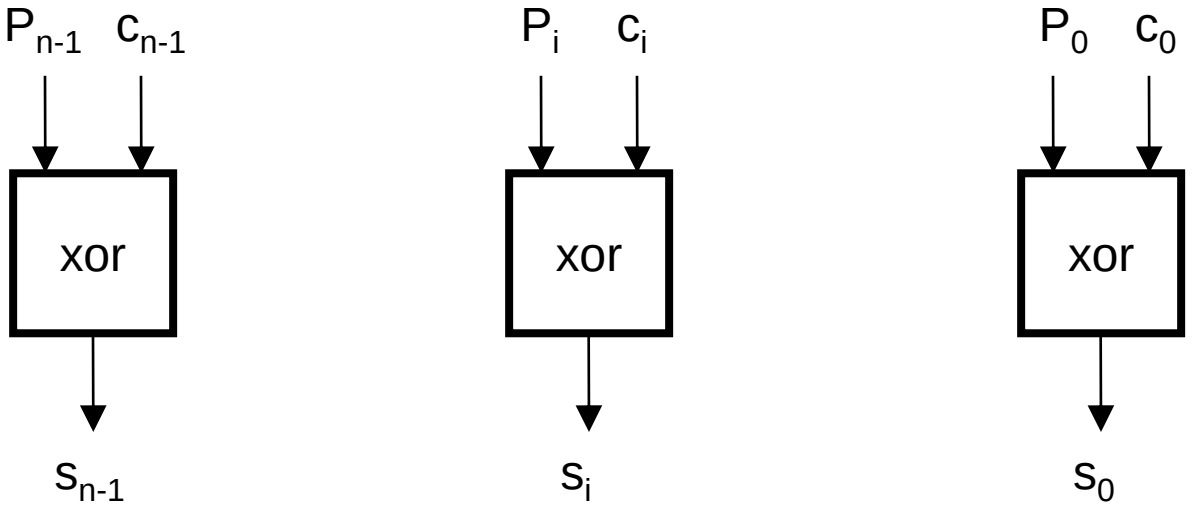
  $P_i = a_i \oplus b_i$



output stage

- bit sliced

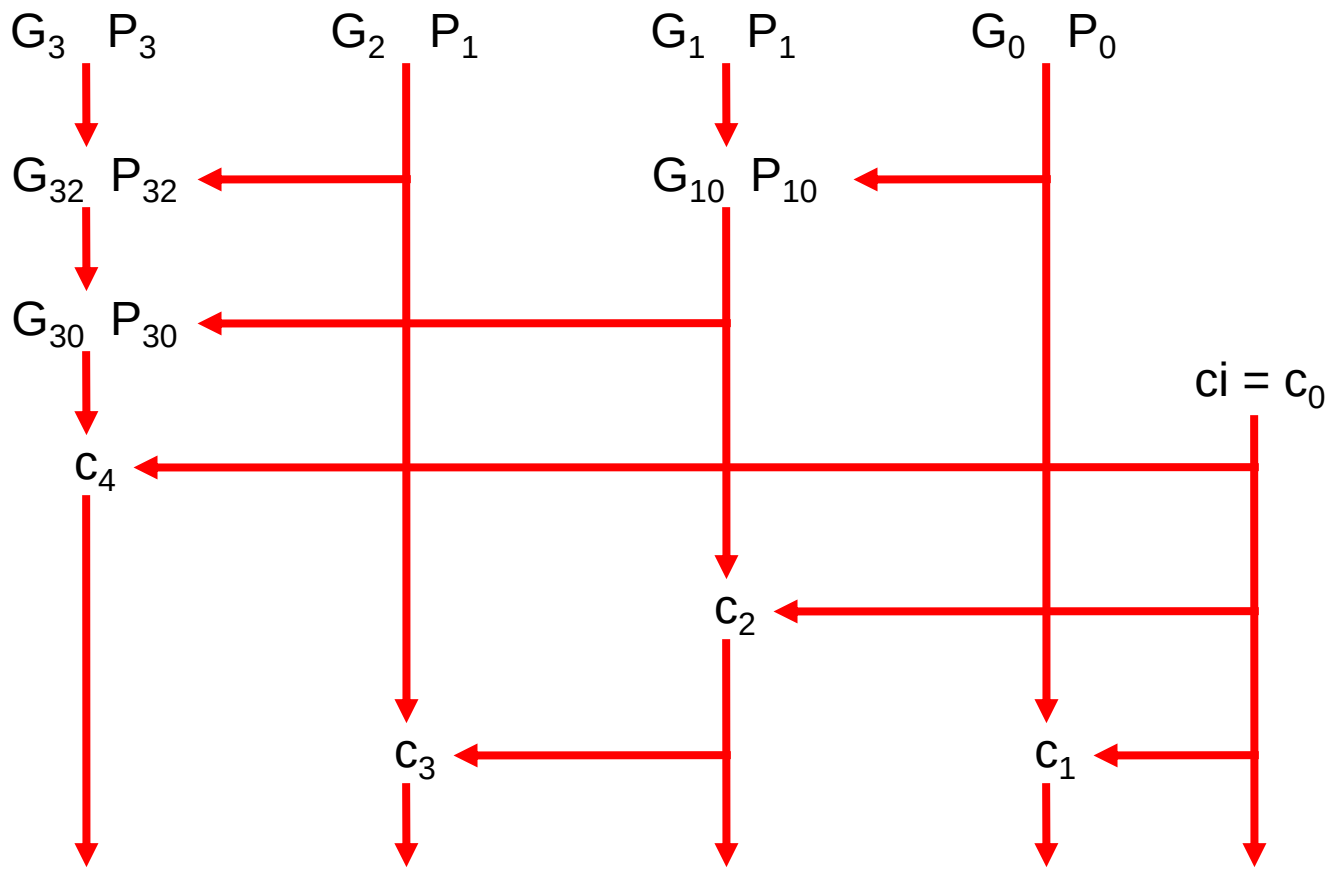
  $s_i = a_i \oplus b_i \oplus c_i = P_i \oplus c_i$



lookahead stage

- uses a logarithmic number of stages to maximize speed.
- various architectures for prefix tree, that optimize depth, fan-out, wire tracks, regularity or other figure of merit.
  - Brent-Kung
  - Sklansky
  - Kogge-Stone
  - Han-Carlson
  - Knowles
  - Ladner-Fischer
  - higher valency variants of the above

Brent-Kung prefix tree



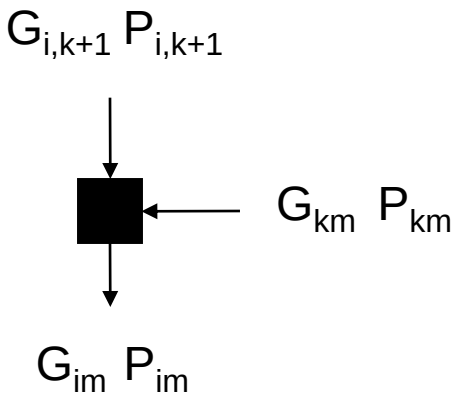
notations

- square block

  $G_{im} = G_{i,k+1} + P_{i,k+1}G_{km}$

  $P_{im} = P_{i,k+1}P_{km}$

- $G_{ii} = G_i$
- $P_{ii} = P_i$

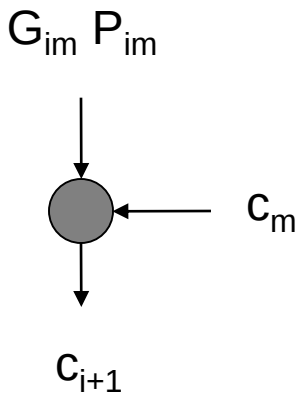


notations

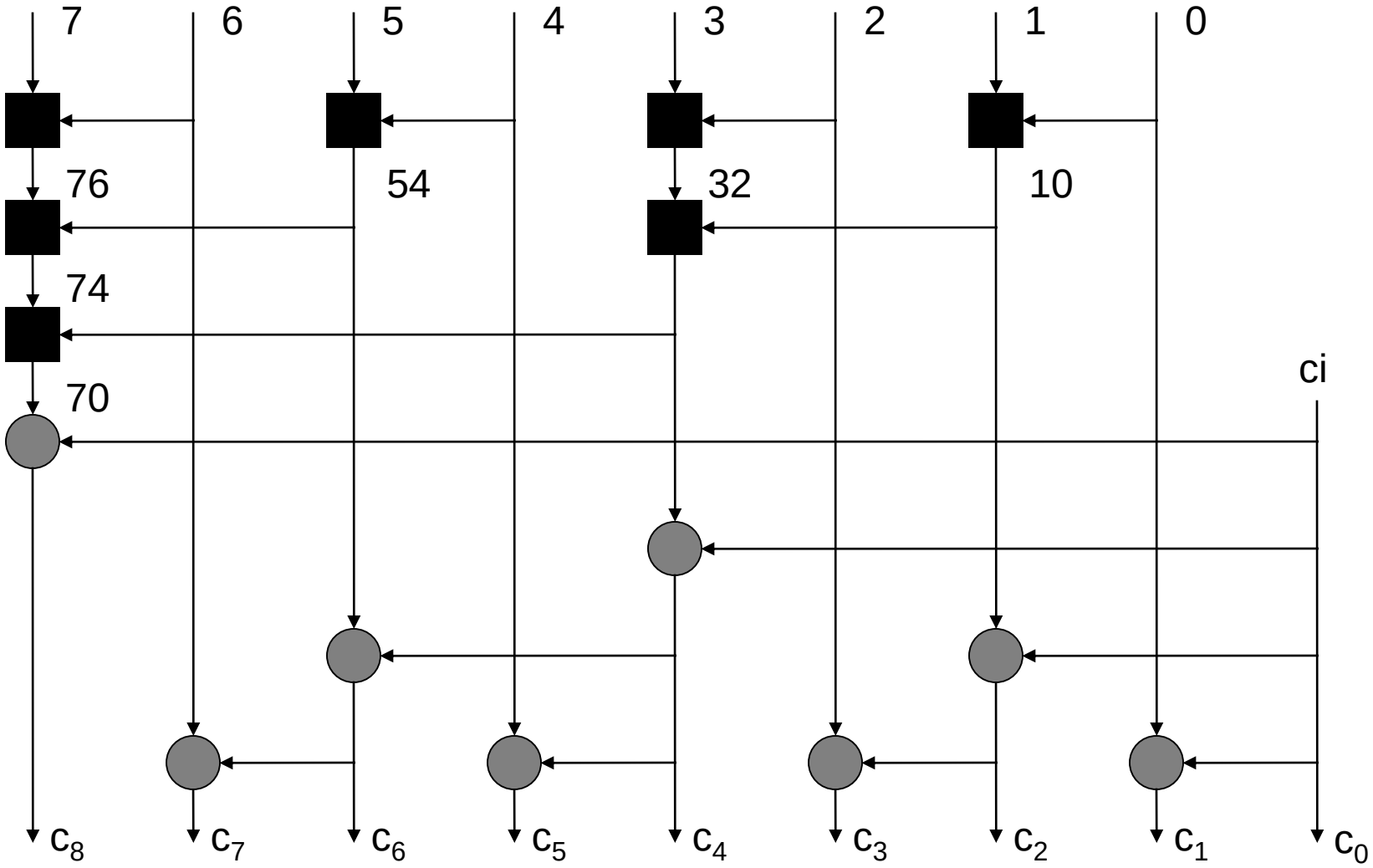
- circle block

  $c_{i+1} = G_{im} + P_{im}c_m$

- $G_{ii} = G_i$
- $P_{ii} = P_i$



8 bit Brent-Kung prefix tree



array of wires

- rectangular array of N x L wires:

```
wire [N-1:0] p [0:L-1]; // L wires of N bits
```

  - simple to declare and use
- irregular array (N bits wire, then N/2 bits, N/4 a.s.o.)

```
genvar g;
generate
    for(g = 0; N/(1<<g) > 0; g = g + 1) begin : mywire
        wire [(N/(1<<g))-1:0] p;
    end
end
endgenerate
```

  - convenient for some structures

internals

```
module brent_kung #(parameter N = 4) (
    input  [(1<<N)-1:0] a,
    input  [(1<<N)-1:0] b,
    output [(1<<N)-1:0] s,
    output co
);

wire [(1<<N)-1:0] g [0:N];
wire [(1<<N)-1:0] p [0:N];
wire [(1<<N)  :0] c;

genvar i;
genvar level;

assign c[0] = 1'b0;
assign co = c[1<<N];

// input and output stages
// brent-kung tree
endmodule
```

input and output stages

```
generate
    for(i = 0; i < (1<<N); i = i + 1) begin: input_stage
        genpro genpro(a[i], b[i], g[0][i], p[0][i]);
    end
endgenerate

generate
    for(i = 0; i < (1<<N); i = i + 1) begin: output_stage
        assign s[i] = p[0][i] ^ c[i];
    end
endgenerate
```

tree of squares

```
generate
    for(level = 1; level <= N; level = level + 1) begin : row
        for(i = (1<<N) - 1; i >= 0; i = i - (1<<level) ) begin : square
            square square(
                g[level-1][i],
                p[level-1][i],
                g[level-1][i - (1<<(level-1))],
                p[level-1][i - (1<<(level-1))],
                g[level][i],
                p[level][i]
            );
        end
    end
endgenerate
```

tree of circles

```
generate
    for(level = 0; level <= N; level = level + 1) begin : second_row
        for(i = (1<<level); i <= (1<<N); i = i + (2<<level) ) begin : circle
            circle circle (
                g[level][i-1],
                p[level][i-1],
                c[i-1],
                c[i]
            );
        end
    end
endgenerate
```