# Lab #5 Kogge Stone Adder

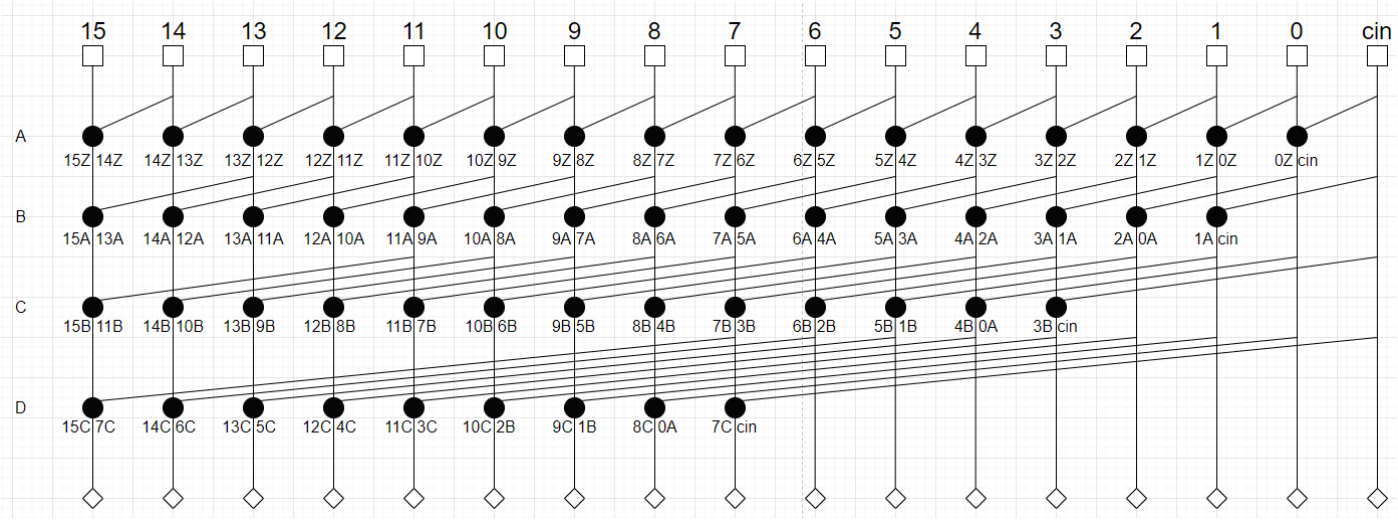
Class : 00

201602004 박태현

I. 실습 목적

Kogge stone adder 의 구현

II. Design procedure




III. Simulation

```
module input_cell(a,b,p,g);
input a,b;
output p,g;

assign p = a ^ b;
assign g = a & b;
endmodule
```

네모 모양의 입력 셀 모듈을 통해 p_i 와 g_i 를 먼저 계산합니다

```
module black_cell(Pim,Gim,Pmj,Gmj,P,G);
input Pim,Gim,Pmj,Gmj;
output P,G;

assign P = Pim & Pmj;
assign G = Gim | (Pim & Gmj);

endmodule
```

검은 원에 해당하는 모듈에서는 상위 비트 집합의 P, G와 하위 비트 집합의 P,G를 합쳐서 두 집합 전체의 P와 G를 생성해서 반환합니다

```
module carry_eval_cell(Pi0,Gi0,c0,ci);
input Gi0,Pi0,c0;
output ci;

assign ci = Gi0 | (Pi0 & c0);
```

마지막으로 계산한 P_i0, G_i0, cin을 바탕으로 각 자리의 캐리를 계산합니다

```
module kogge_stone(x,y,cin,cout,sum);
input [15:0] x,y;
input cin;
output cout;
output [15:0] sum;
wire [15:0]
   G_Z,P_Z, // input_cell
   G_A,P_A, // merge level 1
   G_B,P_B, // merge level 2
   G_C,P_C, // merge level 3
   G_D,P_D; // merge level 4
wire [16:1] c;
```

입력으로 16비트 x,y를 받도록 했습니다

```
// input level
input_cell level_Z0(x[0],y[0],P_Z[0],G_Z[0]);
input_cell level_Z1(x[1],y[1],P_Z[1],G_Z[1]);
input_cell level_Z2(x[2],y[2],P_Z[2],G_Z[2]);
input_cell level_Z3(x[3],y[3],P_Z[3],G_Z[3]);
input_cell level_Z4(x[4],y[4],P_Z[4],G_Z[4]);
input_cell level_Z5(x[5],y[5],P_Z[5],G_Z[5]);
input_cell level_Z6(x[6],y[6],P_Z[6],G_Z[6]);
input_cell level_Z7(x[7],y[7],P_Z[7],G_Z[7]);
input_cell level_Z8(x[8],y[8],P_Z[8],G_Z[8]);
input_cell level_Z9(x[9],y[9],P_Z[9],G_Z[9]);
input_cell level_Z10(x[10],y[10],P_Z[10],G_Z[10]);
input_cell level_Z11(x[11],y[11],P_Z[11],G_Z[11]);
input_cell level_Z12(x[12],y[12],P_Z[12],G_Z[12]);
input_cell level_Z13(x[13],y[13],P_Z[13],G_Z[13]);
input_cell level_Z14(x[14],y[14],P_Z[14],G_Z[14]);
input_cell level_Z15(x[15],y[15],P_Z[15],G_Z[15]);
```

입력이 들어오면 첫번째로 입력 셀을 계산해줍니다.

입력 셀에 계산된 P_Z, G_Z를 바탕으로 A 레벨을 계산합니다

```
// level 1
black_cell level_0A(P_Z[0],G_Z[0],0     ,cin   ,P_A[0],G_A[0]);
black_cell level_1A(P_Z[1],G_Z[1],P_Z[0],G_Z[0],P_A[1],G_A[1]);
black_cell level_2A(P_Z[2],G_Z[2],P_Z[1],G_Z[1],P_A[2],G_A[2]);
black_cell level_3A(P_Z[3],G_Z[3],P_Z[2],G_Z[2],P_A[3],G_A[3]);
black_cell level_4A(P_Z[4],G_Z[4],P_Z[3],G_Z[3],P_A[4],G_A[4]);
black_cell level_5A(P_Z[5],G_Z[5],P_Z[4],G_Z[4],P_A[5],G_A[5]);
black_cell level_6A(P_Z[6],G_Z[6],P_Z[5],G_Z[5],P_A[6],G_A[6]);
black_cell level_7A(P_Z[7],G_Z[7],P_Z[6],G_Z[6],P_A[7],G_A[7]);
black_cell level_8A(P_Z[8],G_Z[8],P_Z[7],G_Z[7],P_A[8],G_A[8]);
black_cell level_9A(P_Z[9],G_Z[9],P_Z[8],G_Z[8],P_A[9],G_A[9]);
black_cell level_10A(P_Z[10],G_Z[10],P_Z[9],G_Z[9],P_A[10],G_A[10]);
black_cell level_11A(P_Z[11],G_Z[11],P_Z[10],G_Z[10],P_A[11],G_A[11]);
black_cell level_12A(P_Z[12],G_Z[12],P_Z[11],G_Z[11],P_A[12],G_A[12]);
black_cell level_13A(P_Z[13],G_Z[13],P_Z[12],G_Z[12],P_A[13],G_A[13]);
black_cell level_14A(P_Z[14],G_Z[14],P_Z[13],G_Z[13],P_A[14],G_A[14]);
black_cell level_15A(P_Z[15],G_Z[15],P_Z[14],G_Z[14],P_A[15],G_A[15]);
```

자신의 비트와 한자리 아래의 비트의 값을 가져와 결과를 만듭니다

```
// level 2
black_cell level_1B(P_A[1],G_A[1],0     ,cin   ,P_B[1],G_B[1]);
black_cell level_2B(P_A[2],G_A[2],P_A[0],G_A[0],P_B[2],G_B[2]);
black_cell level_3B(P_A[3],G_A[3],P_A[1],G_A[1],P_B[3],G_B[3]);
black_cell level_4B(P_A[4],G_A[4],P_A[2],G_A[2],P_B[4],G_B[4]);
black_cell level_5B(P_A[5],G_A[5],P_A[3],G_A[3],P_B[5],G_B[5]);
black_cell level_6B(P_A[6],G_A[6],P_A[4],G_A[4],P_B[6],G_B[6]);
black_cell level_7B(P_A[7],G_A[7],P_A[5],G_A[5],P_B[7],G_B[7]);
black_cell level_8B(P_A[8],G_A[8],P_A[6],G_A[6],P_B[8],G_B[8]);
black_cell level_9B(P_A[9],G_A[9],P_A[7],G_A[7],P_B[9],G_B[9]);
black_cell level_10B(P_A[10],G_A[10],P_A[8],G_A[8],P_B[10],G_B[10]);
black_cell level_11B(P_A[11],G_A[11],P_A[9],G_A[9],P_B[11],G_B[11]);
black_cell level_12B(P_A[12],G_A[12],P_A[10],G_A[10],P_B[12],G_B[12]);
black_cell level_13B(P_A[13],G_A[13],P_A[11],G_A[11],P_B[13],G_B[13]);
black_cell level_14B(P_A[14],G_A[14],P_A[12],G_A[12],P_B[14],G_B[14]);
black_cell level_15B(P_A[15],G_A[15],P_A[13],G_A[13],P_B[15],G_B[15]);
```

0자리는 두 자리 아래가 없으므로 계산하지 않습니다. 아래에 더 계산할 것이 없다는 것은 모든 캐리 전파를 계산 완료했다는 의미입니다

1자리부터 15자리까지는 두 자리 아래의 비트를 가져와 계산을 합니다

```
// level 3
black_cell level_3C(P_B[3],G_B[3],0,cin        ,P_C[3],G_C[3]);
black_cell level_4C(P_B[4],G_B[4],P_A[0],G_A[0],P_C[4],G_C[4]);
black_cell level_5C(P_B[5],G_B[5],P_B[1],G_B[1],P_C[5],G_C[5]);
black_cell level_6C(P_B[6],G_B[6],P_B[2],G_B[2],P_C[6],G_C[6]);
black_cell level_7C(P_B[7],G_B[7],P_B[3],G_B[3],P_C[7],G_C[7]);
black_cell level_8C(P_B[8],G_B[8],P_B[4],G_B[4],P_C[8],G_C[8]);
black_cell level_9C(P_B[9],G_B[9],P_B[5],G_B[5],P_C[9],G_C[9]);
black_cell level_10C(P_B[10],G_B[10],P_B[6],G_B[6],P_C[10],G_C[10]);
black_cell level_11C(P_B[11],G_B[11],P_B[7],G_B[7],P_C[11],G_C[11]);
black_cell level_12C(P_B[12],G_B[12],P_B[8],G_B[8],P_C[12],G_C[12]);
black_cell level_13C(P_B[13],G_B[13],P_B[9],G_B[9],P_C[13],G_C[13]);
black_cell level_14C(P_B[14],G_B[14],P_B[10],G_B[10],P_C[14],G_C[14]);
black_cell level_15C(P_B[15],G_B[15],P_B[11],G_B[11],P_C[15],G_C[15]);
```

2자리 이하는 4자리 밑의 비트가 없으므로 계산하지 않고, 3자리부터 15자리까지 계산합니다.

이 때 0자리는 레벨 B가 없으므로 레벨 A의 값을 가져옵니다

```
// level 4
black_cell level_7D(P_C[7],G_C[7],0,cin        ,P_D[7],G_D[7]);
black_cell level_8D(P_C[8],G_C[8],P_A[0],G_A[0],P_D[8],G_D[8]);
black_cell level_9D(P_C[9],G_C[9],P_B[1],G_B[1],P_D[9],G_D[9]);
black_cell level_10D(P_C[10],G_C[10],P_B[2],G_B[2],P_D[10],G_D[10]);
black_cell level_11D(P_C[11],G_C[11],P_C[3],G_C[3],P_D[11],G_D[11]);
black_cell level_12D(P_C[12],G_C[12],P_C[4],G_C[4],P_D[12],G_D[12]);
black_cell level_13D(P_C[13],G_C[13],P_C[5],G_C[5],P_D[13],G_D[13]);
black_cell level_14D(P_C[14],G_C[14],P_C[6],G_C[6],P_D[14],G_D[14]);
black_cell level_15D(P_C[15],G_C[15],P_C[7],G_C[7],P_D[15],G_D[15]);
```

6자리 이하는 8자리 밑의 비트가 없으므로 계산하지 않고, 7자리부터 15자리까지 계산합니다.

0자리는 레벨 A를, 1,2자리는 레벨 B를 사용합니다(C가 없기 때문)

이제 모든 자리가 최하위 비트로부터의 모든 캐리 전파를 계산했으므로, 이를 통해 각 자리 캐리를 계산합니다

```
carry_eval_cell c1(P_A[0],G_A[0],cin,c[1]);
carry_eval_cell c2(P_B[1],G_B[1],cin,c[2]);
carry_eval_cell c3(P_B[2],G_B[2],cin,c[3]);
carry_eval_cell c4(P_C[3],G_C[3],cin,c[4]);
carry_eval_cell c5(P_C[4],G_C[4],cin,c[5]);
carry_eval_cell c6(P_C[5],G_C[5],cin,c[6]);
carry_eval_cell c7(P_C[6],G_C[6],cin,c[7]);
carry_eval_cell c8(P_D[7],G_D[7],cin,c[8]);
carry_eval_cell c9(P_D[8],G_D[8],cin,c[9]);
carry_eval_cell c10(P_D[9],G_D[9],cin,c[10]);
carry_eval_cell c11(P_D[10],G_D[10],cin,c[11]);
carry_eval_cell c12(P_D[11],G_D[11],cin,c[12]);
carry_eval_cell c13(P_D[12],G_D[12],cin,c[13]);
carry_eval_cell c14(P_D[13],G_D[13],cin,c[14]);
carry_eval_cell c15(P_D[14],G_D[14],cin,c[15]);
carry_eval_cell c16(P_D[15],G_D[15],cin,c[16]);
```

각 자리의 캐리를 cin과 P_i0, G_i0을 통해 직접 계산할 수 있습니다.

각 자리의 캐리를 계산한 뒤, 이를 통해 덧셈 결과를 만듭니다

```
assign sum[0] = cin ^ P_Z[0];
assign sum[1] = c[1] ^ P_Z[1];
assign sum[2] = c[2] ^ P_Z[2];
assign sum[3] = c[3] ^ P_Z[3];
assign sum[4] = c[4] ^ P_Z[4];
assign sum[5] = c[5] ^ P_Z[5];
assign sum[6] = c[6] ^ P_Z[6];
assign sum[7] = c[7] ^ P_Z[7];
assign sum[8] = c[8] ^ P_Z[8];
assign sum[9] = c[9] ^ P_Z[9];
assign sum[10] = c[10] ^ P_Z[10];
assign sum[11] = c[11] ^ P_Z[11];
assign sum[12] = c[12] ^ P_Z[12];
assign sum[13] = c[13] ^ P_Z[13];
assign sum[14] = c[14] ^ P_Z[14];
assign sum[15] = c[15] ^ P_Z[15];
```

마지막으로 최상단 비트의 캐리를 캐리아웃으로 계산합니다

```
assign cout = c[16];
```

- Testbench

```
module kogge_stone_tb;

reg [15:0] x,y;
reg cin;
wire cout;
wire [15:0] sum;
reg [15:0] check;
reg chk_out;

parameter iter = 1000000;
integer i;
integer correct_cnt;
```

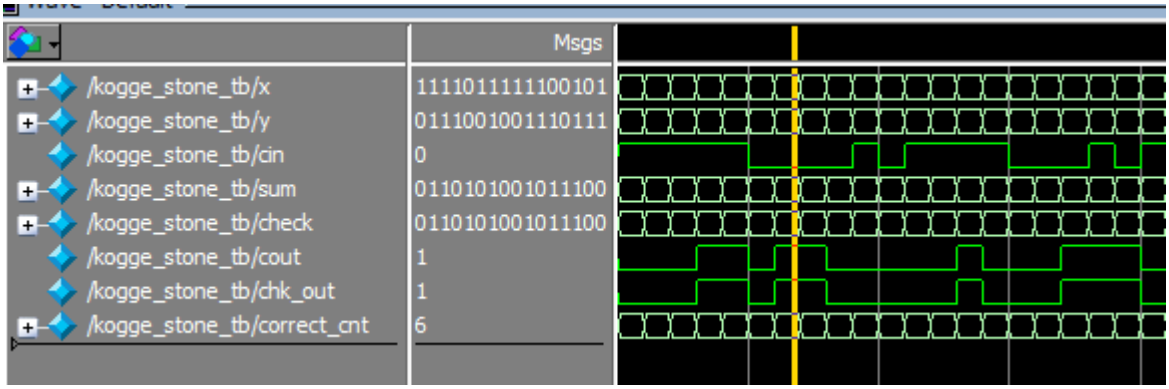
입력값과 검증을 할 때 활용할 변수 check, chk_out을 선언하고. 정답인 경우 카운팅할 변수를 선언했습니다

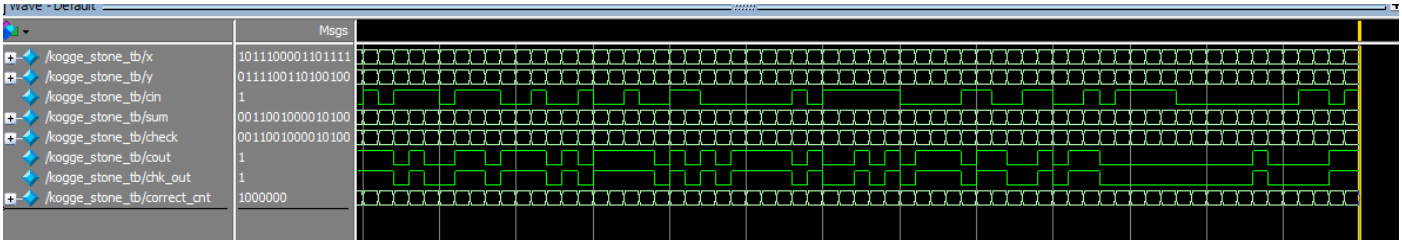
```
kogge_stone i_kogge_stone(x,y,cin,cout,sum);

initial begin
   correct_cnt = 0;
   for(i = 0; i < iter; i=i+1) begin
      x = $random;
      y = $random;
      cin = $random;
      {chk_out, check} = x + y + cin;
      #10
      if({cout, sum} == {chk_out, check})
         correct_cnt = correct_cnt + 1;
      else
         $display($time, " : %d + %d + %d = %d, but was (%d)\n",x,y,cin,{chk_out, check},{cout,sum});
   end

   $display("correct count = %d\n",correct_cnt);
end

endmodule
```

100만번 반복을 하면서 x,y,cin에 랜덤 값을 넣습니다. 그리고 그 결과를 chk_out, check 와이어에 넣어 정답을 wire로 전달합니다. 그리고 그 값이 kogge_stone_adder의 결과와 일치하는지 확인합니다. 만약 다르다면, 실제 정답과 계산한 오답을 출력하게 했습니다.

반복이 끝나면, 총 정답을 맞춘 수를 출력하도록 했습니다

- Waveform




IV. Evaluation

모든 값을 일일이 확인하는 것은 불가능하므로, 일부 값에 대해 완전히 일치하는 지 몇 개를 확인해보았고, correct_cnt가 루프 반복 횟수와 일치하는지 확인했습니다



```
# correct count =     1000000
```

V. Discussions

병렬 덧셈을 활용해 덧셈 연산을 고속화를 어떻게 하는지 알 수 있었습니다.

Structural과 dataflow를 같이 써서 어색했습니다

블록 다이어그램을 그리고 코드를 짰음에도 불구하고 모듈 자체가 복잡해서 커넥션에서 실수를 많이 했습니다