10주차 결과보고서

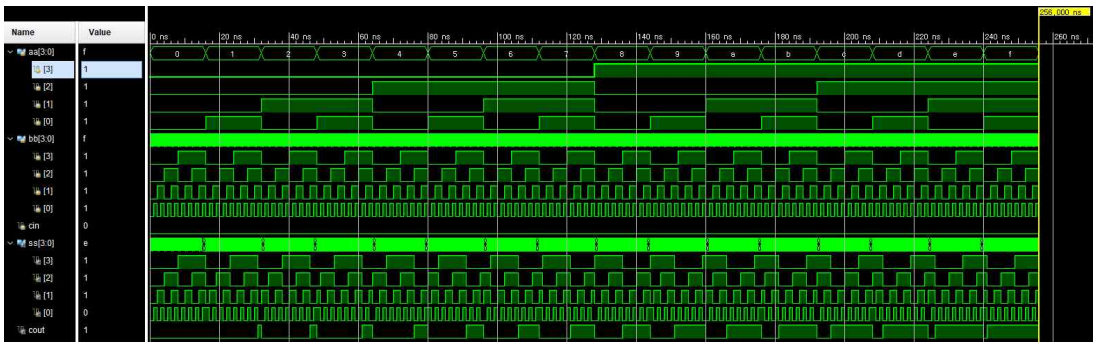
전공: 신문방송학과 학년: 4학년 학번: 20191150 이름: 전현길

1. 4bit Binary Parallel Adder 의 결과 및 Simulation 과정에 대해서 설명하시오. (verilog source, 출력 예시, 과정 상세히 적을 것)

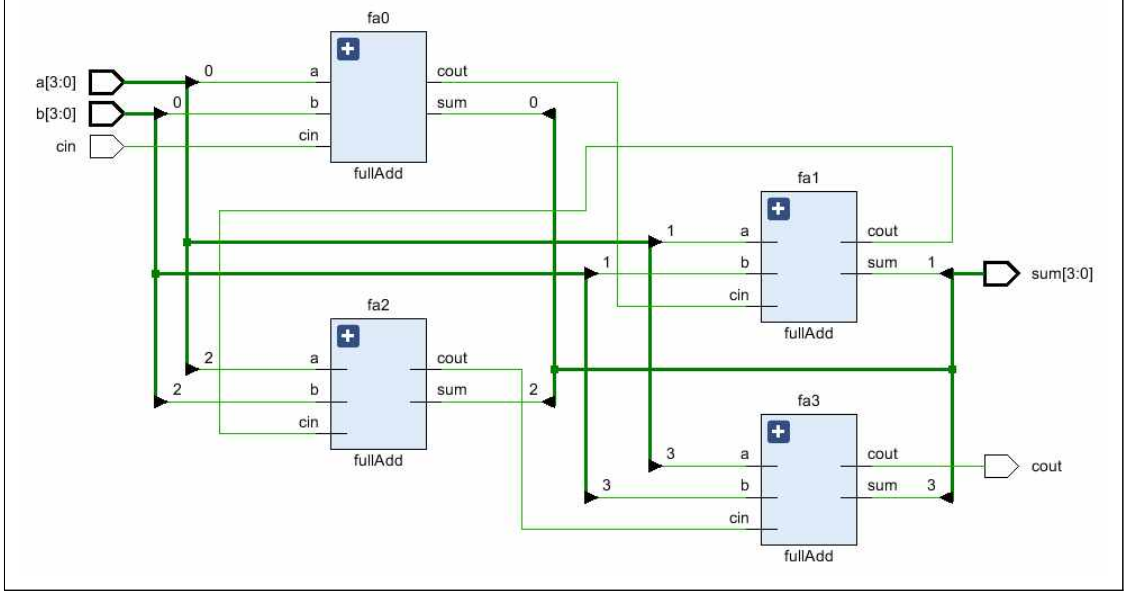
source code	testbench code

`timescale 1ns / 1ps
 	
// full adder
module fullAdd (
input a, b, cin,
output sum, cout
);
assign sum = a^b^cin;
assign cout = (a&b) | ((a^b) & cin);
endmodule
 	
// 4-bit ripple carry adder
module fullAdd4 (
input [3:0] a, b,
input cin,
output [3:0] sum,
output cout
);
wire [2:0] carry; // 내부 캐리 신호
fullAdd fa0(a[0], b[0], cin, sum[0], carry[0]);
fullAdd fa1(a[1], b[1], carry[0], sum[1], carry[1]);
fullAdd fa2(a[2], b[2], carry[1], sum[2], carry[2]);
fullAdd fa3(a[3], b[3], carry[2], sum[3], cout);
endmodule
	`timescale 1ns / 1ps
	
	module adder_tb;
	
	reg [3:0] aa, bb;
	reg cin;
	wire [3:0] ss;
	wire cout;
	
	// 4-bit ripple carry adder
	fullAdd4 u_test (
	.a (aa), .b (bb), .cin (cin), .sum (ss), .cout (cout)
	);
	
	initial begin
	aa = 4'b0000;
	bb = 4'b0000;
	cin = 1'b0;
	end
	
	always@ (aa[3] or aa[2] or aa[1] or aa[0] or bb[3] or bb[2] or bb[1] or bb[0]) begin
	aa[3] <= #128 ~aa[3];
	aa[2] <= #64 ~aa[2];
	aa[1] <= #32 ~aa[1];
	aa[0] <= #16 ~aa[0];
	bb[3] <= #8 ~bb[3];
	bb[2] <= #4 ~bb[2];
	bb[1] <= #2 ~bb[1];
	bb[0] <= #1 ~bb[0];
	end
	
	initial begin
	#256
	$finish;
	end
	
	endmodule

4-bit pararell binary adder simultation



4-bit pararell binary adder circuit design



4-bit parell binary adder는 **전가산기 4개를 병렬로 배치하여 4bit 이진수 덧셈을 계산**하는 회로이다. 4bit뿐만이 아니라, 전가산기를 n개 배치함으로써 n-bit 이진수 덧셈을 계산할 수 있다. 이 때, **첫 C_{in} bit는 항상 0이 되므로 4-bit pararell binary adder에 입력하는 cin은 simulation에서 생략했다.**

기존까지 작성했던 회로들과 달리, 4-bit parell binary adder는 작성한 회로를 여러 번 재사용하는 방식으로 구현되어 있다. 따라서 full adder module을 먼저 구현한 뒤, full adder 4개를 이어붙여 4-bit ripple carry adder를 만드는, 기존에 시도한 적 없는 방식으로 Verilog 코드를 작성했다.

위 방식으로 Verilog code를 작성하면서 고려할 점은 다음과 같다.

1) full adder를 input [3:0] a, b 문이나 output [3:0] sum 문을 사용하면 bit마다 이름을 따로 지정하는 수고를 덜 수 있다. 모듈에서 모듈로 값을 전달하기도 용이해진다.

2) 4-bit adder 내부의 캐리 신호를 wire문을 통해 각 모듈에 전달했다.

3) fullAdd 모듈 4개를 선언하여 병렬로 연결할 때, 각 모듈에 이름을 붙여야 한다.

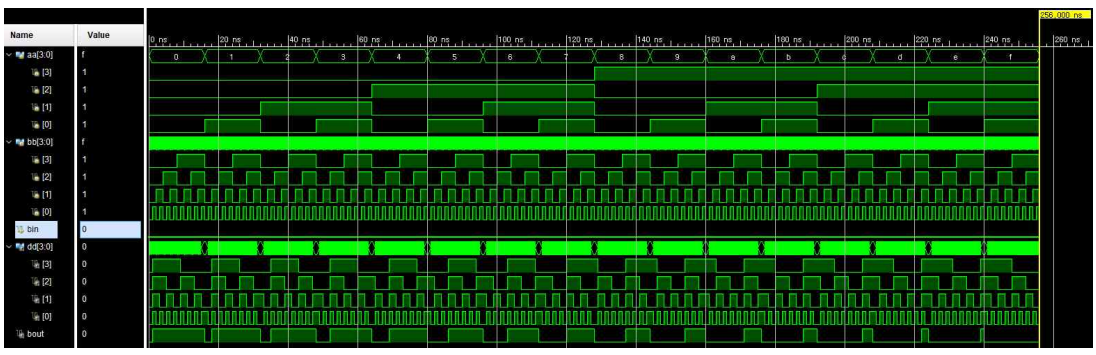
simulation을 살펴보았을 때, 두 4-bit 덧셈 연산이 정상적으로 이루어지고 있으므로 성공적으로 구현되었음을 확인할 수 있다.

2. 4bit Binary Parallel Subtractor의 결과 및 Simulation 과정에 대해서 설명하시오. (verilog source, 출력 예시, 과정 상세히 적을 것)

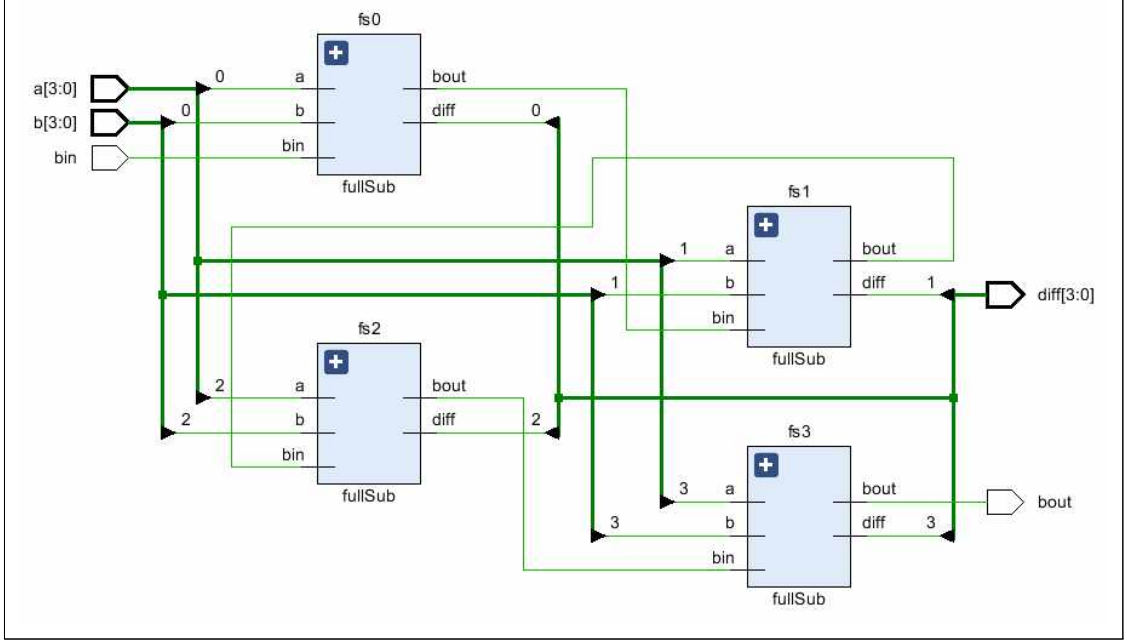
source code	testbench code

`timescale 1ns / 1ps
 	
// full adder
module fullSub (
input a, b, bin,
output diff, bout
);
assign diff = a^b^bin;
assign bout = (~a&b) | (~(a^b) & bin);
endmodule
 	
// 4-bit ripple carry adder
module fullSub4 (
input [3:0] a, b,
input bin,
output [3:0] diff,
output bout
);
wire [2:0] borrow; // 내부 캐리 신호
fullSub fs0(a[0], b[0], bin, diff[0], borrow[0]);
fullSub fs1(a[1], b[1], borrow[0], diff[1], borrow[1]);
fullSub fs2(a[2], b[2], borrow[1], diff[2], borrow[2]);
fullSub fs3(a[3], b[3], borrow[2], diff[3], bout);
endmodule
	`timescale 1ns / 1ps
	
	module subtractor_tb;
	
	reg [3:0] aa, bb;
	reg bin;
	wire [3:0] dd;
	wire bout;
	
	// 4-bit ripple carry adder
	fullSub4 u_test (
	.a (aa), .b (bb), .bin (bin), .diff (dd), .bout (bout)
	);
	
	initial begin
	aa = 4'b0000;
	bb = 4'b0000;
	bin = 1'b0;
	end
	
	always@ (aa[3] or aa[2] or aa[1] or aa[0] or bb[3] or bb[2] or bb[1] or bb[0]) begin
	aa[3] <= #128 ~aa[3];
	aa[2] <= #64 ~aa[2];
	aa[1] <= #32 ~aa[1];
	aa[0] <= #16 ~aa[0];
	
	bb[3] <= #8 ~bb[3];
	bb[2] <= #4 ~bb[2];
	bb[1] <= #2 ~bb[1];
	bb[0] <= #1 ~bb[0];
	end
	
	initial begin
	#256
	$finish;
	end
	
	endmodule

4-bit pararell binary subtractor simulation



4-bit pararell binary subtractor circuit design



4-bit parell binary subtractor는 **전감산기 4개를 병렬로 배치하여 4bit 이진수뺄셈을 계산**하는 회로이다. 4bit뿐만이 아니라, 전감산기를 n개 배치함으로써 n-bit 이진수 뺄셈을 계산할 수 있다. 이 때, **첫 B_{in} bit는 항상 0이 되므로 4-bit pararell binary subtractor에 입력하는 bin은 simulation에서 생략했다.**

가산기와 동일하게, 4-bit parell binary subtractor 역시 작성한 회로를 여러 번 재사용하는 방식으로 구현되어 있다. 따라서 full subtractor module을 먼저 구현한 뒤, full subtractor 4개를 이어붙여 4-bit ripple carry subtractor를 만들었다.

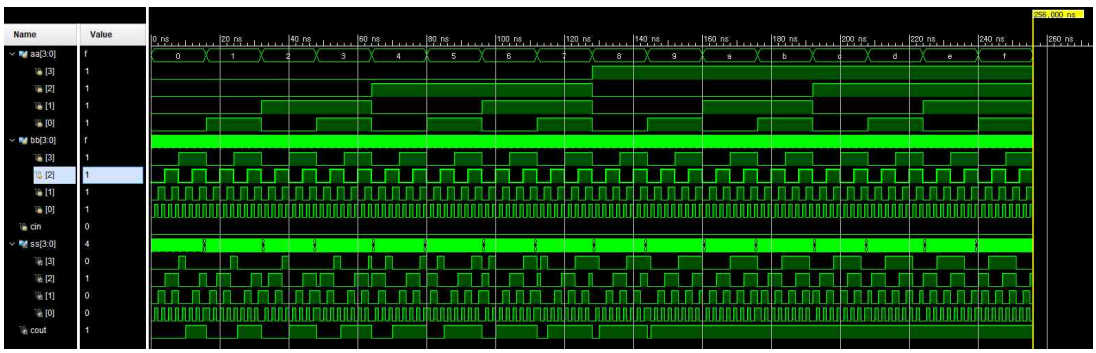
simulation을 살펴보았을 때, 두 4-bit 덧셈 연산이 정상적으로 이루어지고 있으므로 성공적으로 구현되었음을 확인할 수 있다.

3. BCD Adder의 결과 및 Simulation 과정에 대해서 설명하시오. (verilog source, 출력 예시, 과정을 상세히 적을 것)

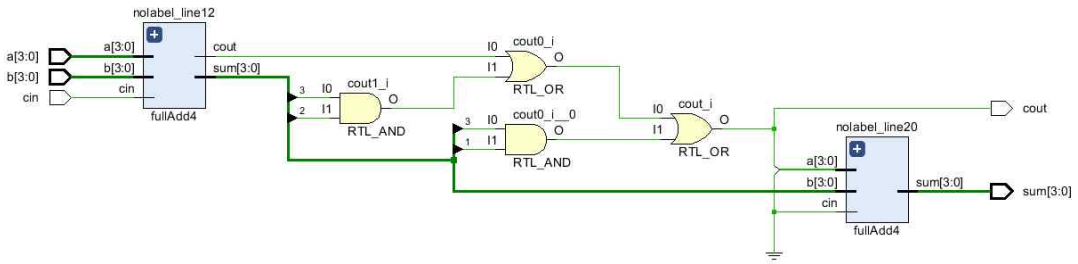
source code	testbench code

`timescale 1ns / 1ps
 	
// full adder
module fullAdd (
input a, b, cin,
output sum, cout
);
assign sum = a^b^cin;
assign cout = (a&b) | ((a^b) & cin);
endmodule
 	
// 4-bit ripple carry adder
module fullAdd4 (
input [3:0] a, b,
input cin,
output [3:0] sum,
output cout
);
wire [2:0] carry; // 내부 캐리 신호
fullAdd fa0(a[0], b[0], cin, sum[0], carry[0]);
fullAdd fa1(a[1], b[1], carry[0], sum[1], carry[1]);
fullAdd fa2(a[2], b[2], carry[1], sum[2], carry[2]);
fullAdd fa3(a[3], b[3], carry[2], sum[3], cout);
endmodule
 	
module BCD (
input [3:0] a, b,
input cin,
output [3:0] sum,
output cout
);
wire [3:0] m_sum, c;
wire m_cout, garbage;
 	
fullAdd4 fa4_0(a, b, cin, m_sum, m_cout);
assign cout = m_cout | (m_sum[3] & m_sum[2]) | (m_sum[3] & m_sum[1]);
 	
assign c[3] = 0;
assign c[2] = cout;
assign c[1] = cout;
assign c[0] = 0;
 	
fullAdd4 fa4_1(c, m_sum, 0, sum, garbage);
endmodule
	`timescale 1ns / 1ps
	
	module BCD_tb;
	
	reg [3:0] aa, bb;
	reg cin;
	wire [3:0] ss;
	wire cout;
	
	// 4-bit ripple carry adder
	BCD u_test (
	.a (aa), .b (bb), .cin (cin), .sum (ss), .cout (cout)
	);
	
	initial begin
	aa = 4'b0000;
	bb = 4'b0000;
	cin = 1'b0;
	end
	
	always@ (aa[3] or aa[2] or aa[1] or aa[0] or bb[3] or bb[2] or bb[1] or bb[0]) begin
	aa[3] <= #128 ~aa[3];
	aa[2] <= #64 ~aa[2];
	aa[1] <= #32 ~aa[1];
	aa[0] <= #16 ~aa[0];
	
	bb[3] <= #8 ~bb[3];
	bb[2] <= #4 ~bb[2];
	bb[1] <= #2 ~bb[1];
	bb[0] <= #1 ~bb[0];
	end
	
	initial begin
	#256
	$finish;
	end
	
	endmodule

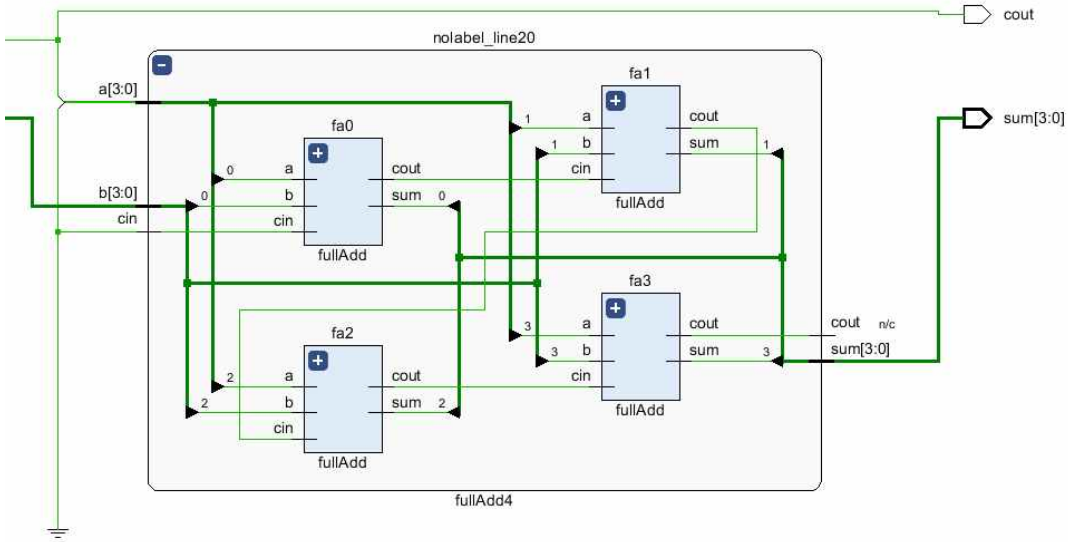
BCD adder simulation



BCD adder circuit design 1

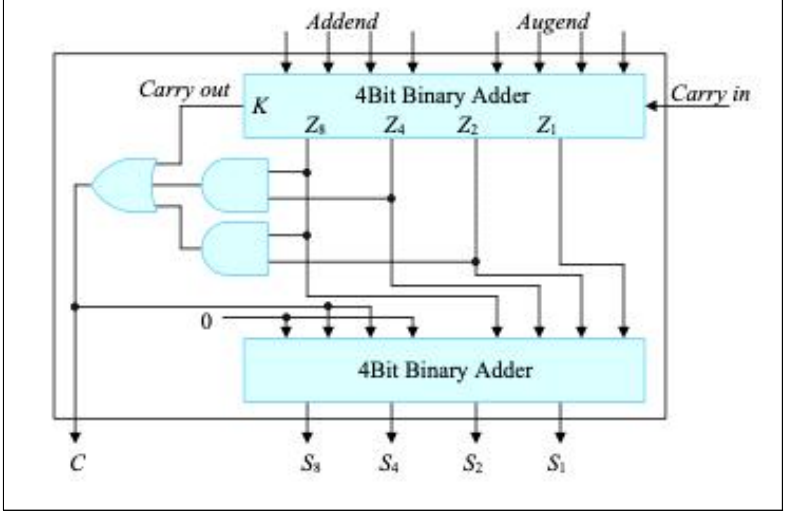


BCD adder circuit design 2



BCD code는 6주차에서 보았듯이, 8421코드를 기준으로 보았을 때 십진수의 한 자릿수를 이진수 4자리를 통해 표현하는 기법이다. BCD adder는 BCD code에 따라 이진수 bit sequence를 계산한다. 이 때문에 BCD adder에서 0101(5) + 1001(9)의 결과는 1110(14)가 아니라, 1 0100 (14)로 저장된다.

BCD adder circuit diagram



구현의 논리적 개괄은 위와 같다. 먼저 4-bit 이진수를 4-bit binary adder를 통해 계산해 S_3, S_2, S_1, S_0을 출력한다. binary adder의 출력 결과를 확인하고, S_3이 1이고 S_2가 1이거나(=12-15), S_3이 1이고 S_1가 1이면(=10-11) 보정이 필요한 값이므로 캐리 출력을 활성화하고 값을 변환한다.

Verilog code상으로는 4-bit ripple carry adder 모듈을 재사용해 구현했다. 구체적인 동작은 다음과 같다.

1) 먼저 4bit 입력 A, B와 C_{in}(=0)을 4-bit adder로 더해 출력값을 m_sum[3:0], m_cout에 담는다.

2) m_cout, m_sum의 값을 보정 회로에 입력해 최종적인 cout을 얻는다.

3) cout 값을 통해 C[3:0](0110 또는 0000)을 구현한다.

4) m_sum[3:0], C[3:0], Cin(=0)을 다시 4-bit adder로 더해 최종적인 출력값을 sum[3:0], garbage에 담는다.

5) garbage 값은 버려지고, cout과 sum[3:0]이 출력된다.

simulation을 살펴보았을 때, 두 BCD 덧셈 연산이 정상적으로 이루어지고 있으므로 성공적으로 구현되었음을 확인할 수 있다.

3. 결과 검토 및 논의 사항

이번 실험에서는 4-bit parallel binary adder, 4-bit pararell binary subtractor, BCD adder 회로를 Verilog 코드로 구현해 보았으며, FPGA 보드를 활용하여 실제 출력 결과를 확인해 보았다. 각 회로들을 구성하는 부분 회로들을 모듈 단위로 코딩하고, 재사용하면서 모듈형 언어인 Verilog의 장점을 체감할 수 있었다. 실험 결과, 구현을 통해 기대되었던 동작이 정상적으로 이루어졌음을 확인했다.

4. 추가 이론 조사 및 작성

MSI(medium scale integrated cicuits)란 반도체 IC에서 회로 내에 중간 정도의 트랜지스터를 갖는 회로들을 부르는 용어이다. 약 10개-100개 정도의 logic gate를 가지고 있을 때 MSI라고 부른다. 이에 대비되는 용어로 SSI, LSI, VLSI가 존재하는데, 각 용어에 따른 트랜지스터 수는 아래와 같다.

Density of Integration / Complexity	Gates per IC
SSI: Small-Scale Integration • Logic Gates (AND, OR, NAND, NOR)	<10
MSI: Medium-Scale Integration • Flip Flops • Adders / Counters • Multiplexers & De-multiplexers	10 – 100
LSI: Large-Scale Integration •Small Memory Chips •Programmable Logic Device	100 – 10,000
VLSI: Very Large-Scale Integration •Large Memory Chips •Complex Programmable Logic Device	10,000 – 100,000
ULSI: Ultra Large-Scale Integration •8 & 16 Bit Microprocessors	100,000 – 1,000,000
GSI: Giga-Scale Integration •Pentium IV Processor	>1,000,000

표에서도 알 수 있듯이 SSI에는 AND, OR, NAND, NOR 게이트 등 기본적인 논리 게이트가 포함되며, MSI에는 플립 플롭, 가감산기, 카운터, 멀티플렉서, 디멀티플렉서 등의 논리적인 기능을 갖지만 여전히 다양한 형태로 응용될 수 있는 low-level에 가까운 회로들이 포함된다. MSI를 넘어서는 수준으로 회로가 복잡해질수록, 회로들은 단순한 몇 가지의 기능을 넘어서 추상적이고 복잡한 동작까지 수행할 수 있게 된다.