|  |  |  |  |
| --- | --- | --- | --- |
| 디지털시스템실험 결과보고서 | | | |
| 이름 : 정성훈 | | | 학번 : 2019171009 |
| **실험제목** | Arithmetic Circuit 설계 및 구현 | | |
| **실험목표** | 1. Full Adder와 이를 이용한 Ripple Carry Adder 설계 2. 4-bits Adder/Subtractor 설계 3. 4-bits x 3-bits Multiplier 설계 4. Arithmetic and Logical Unit 설계 | | |
| **실험결과** | | | |
| 1. 4-bits Adder/Subtractor  (1) 구현하고자 하는 논리회로    위 그림과 같이 A3~A0과, B3~B0, 그리고 M을 입력받아서 결과 S3~S0과 Carry out, Overflow flag를 출력하는 4-bits adder/subtractor를 설계하고자 하였다.  (2) Verilog 코드    먼저 Structural modeling을 위하여 1-bit full adder를 gate-level로 설계했다.    이후 위 그림과 같이 선을 연결했다. A는 Full Adder에 그대로 붙이고, B는 M과 XOR을 한 결과를 붙였다. 가장 하위 비트의 Full Adder에는 carry in으로 M이 들어갔다. 최종 carry out으로 C[3]이, overflow flag로 C[2] XOR C[3]이 들어갔다.    위는 테스트벤치 코드다. 모듈에 선을 연결한 후 4+7, 13+12, 13-5, 5-2를 연산하기 위해 값을 넣었다.  (3) 시뮬레이션 파형    위와 같은 결과를 얻었다. 가독성을 위해 A와 B, S는 signed decimal로 병기했다. 설계한 모듈은 4-bits signed integer 두 개의 덧셈/뺄셈을 연산하는 모듈이었기 때문에 가능한 A, B, S값의 범위가 -8 ~ +7까지였다. 그래서 4 + 7 = 11에서 11은 표현할 수 있는 범위를 넘어가기 때문에 overflow가 났다.  두번째에서 A에 13(=4’b1101), B에 12(=4’b1100)을 인가해 13 + 12를 연산하고자 했는데, 이를 모듈이 singed 4-bits로 받아들여 -3(=4’b1101)과 -4(=4’b1100)의 덧셈을 계산해 -7(=4’b1001)을 연산하고, 이는 표현할 수 있는 범위 내기 때문에 overflow flag가 0이었다.  세 번째 역시 비슷하게 A의 13을 -3으로 읽고 -3 - (+5)를 연산해 -8을 출력했고, 이 역시 4-bits signed 범위 내기 때문에 overflow가 나지 않았다. 네 번째는 5 - 2 = 3으로 A와 B, S 모두 범위 내의 연산이었기 때문에 역시 overflow가 나지 않았다.  2. 4-bits x 3-bits Multiplier  (1) 구현하고자 하는 회로    위와 같이 3-bits A와 4-bits B를 받아 둘을 곱한 결과를 출력하는 Multiplier를 설계하고자 했다. 설계 논리는 오른쪽과 같이 B를 모두 A의 각 비트와 and연산을 한 뒤에 결과를 A 각 비트의 자릿수에서 더한 결과가 Multiply가 되는 것을 이용했다.  이를 위해 (A비트 개수 - 1)만큼의 B비트 Adder가 필요했고, 각 Adder에 A의 각 비트와 B의 모든 비트의 and연산을 넣어줘야 하기 때문에 (A의 bit 수) x (B의 bit 수) 만큼의 and gate가 필요했다.  (2) Verilog 코드    먼저 모듈에 4-bits adder가 필요하기 때문에 이를 위해 4-bits ripple carry adder를 설계했다. ripple carry adder에 full adder가 필요하기 때문에 full adder를 먼저 gate-level로 설계한 뒤 structural 하게 ripple carry adder를 설계했다.    이후에 ripple carry adder를 이용해 최종적으로 multiplier를 설계했다. 연산 결과의 최하위 bit는 A[0] & B[0]으로 고정이고, 첫 번째 ripple carry adder의 input으로 A[1]과 B의 모든 비트의 and 결과와 {0, A[0] & B[3:1]}이 들어갔다. 두 번째 ripple carry adder의 input으로는 A[2]와 B의 모든 비트의 and 결과와 첫 번째 adder의 carry와 sum[3:1]이 들어갔다.    위는 testbench 코드로 3-bits A와 4-bits B에 값을 인가해 7-bits C에 결과를 받았다. 14\*5, 8\*3, 13\*7, 10\*2를 연산하기 위해 A와 B에 값을 넣었다.  (3) 시뮬레이션 파형    위는 testbench를 simulation한 파형이다. 가독성을 위해 A, B, C 모두 unsigned decimal로 병기했다.  네 가지 모든 case에서 기대한 값과 같이 A와 B를 곱한 결과가 나온 것을 확인했다.  (4) FPGA 검증  FPGA에 multiplier 모듈을 올렸다. pin은 LED 2~8까지에 C6~0을 넣었고, DIP SWITCH 1~3에 A2~0을, 5~8에 B3~0을 넣었다.    3’b011(3) \* 4’b0010(2) = 7’b0000110(6)이 나왔다.    3’b111(7) \* 4’b0010(2) = 7’b0001110(14)가 나왔다.    3’b111(7) \* 4’b1010(10) = 7’b1000110(70)이 나왔다.    3’b101(5) \* 4’b1011(11) = 7’b0110111(55)가 나왔다.  3. ALU 설계  (1) 구현하고자 하는 회로    위 그림과 같이 두 개의 Operand와 Opcode를 받아서 연산 결과를 출력하는 Arithmetic and Logic Unit을 설계하고자 했다.    Opcode는 위와 같고, operand는 둘 다 4-bits를 갖는 ALU를 설계해야 했다.  (2) Verilog 코드    case 문을 이용해 ALU를 구현했다. Opcode에 따라서 add, sub, mul, or, and, xor을 수행한 뒤 결과를 C에 저장하고, default로 만약 다른 opcode가 들어온다면 C를 undetermined(don’t care condition)으로 지정해 latch를 방지했다. 오른쪽은 주어진 testbench 코드다.  (3) 시뮬레이션 파형    위와 같은 결과를 얻었다. 가독성을 위해 decimal로 병기했다.  opcode가 0일 때(add) 2 + 3 = 5의 결과를 얻었다.  opcode가 1일 때(sub) 7 - 4 = 3의 결과를 얻었다.  opcode가 2일 때(mul) 2 \* 3 = 6의 결과를 얻었다.  opcode가 3일 때(or) 0001 | 0010 = 0011의 결과를 얻었다.  opcode가 4일 때(and) 0111 & 0111 = 0111의 결과를 얻었다.  opcode가 5일 때(xor) 0101 ^ 0010 = 0111의 결과를 얻었다.  시뮬레이션 파형이 기댓값과 일치했다. 이를 통해 모듈의 정상 작동을 확인했다. | | | |
| **토의** | | 이번 주차에는 4-bits adder/subtractor와 4-bits x 3-bits mutiplier, ALU를 설계했다. Multiplier의 경우에는 input이 총 7 bits가 필요해서 fpga의 dip switch에 mapping했지만, adder/subtractor는 input이 9 bits가 필요했고, ALU는 11bits가 필요했어서 fpga에 올리기 난감했다. number pad에 input을 올려봤지만 계속 누르고 있어야 했고, 동시 입력을 인식하지 못해 adder의 경우는 가능했지만, ALU에서는 불가능했다. 앞으로는 8bits의 input만으로는 부족한 상황이 더 나올텐데 다른 방법을 찾아야겠다고 생각했다. | |