디지털 논리회로

miniCPU 프로젝트 (6조)

최종보고서

담당교수: 박재근

조장: 임효준 (학점교류, 20246096)
조원: 김희균 (학점교류, 20236079)
김효경 (학점교류, 20246036)

제출일자: 2024.06.11.

목차

1) 역할분담

2024학년도 1학기 숭실대학교 '디지털 논리회로' 과목의 mini CPU 제작 팀 프로젝트를 진행함에 있어서 각 조원이 아래와 같이 역할을 분담하여 진행하기로 하였음.

임효준 - 프로그램 설계(코딩)
김희균 - 보고서 작성, 발표
김효경 - 자료조사, 내용기록

위의 역할분담은 해당 역할을 맡은 인원만이 그 역할을 수행한다는 것이 아닌 주로 그 역할을 수행하는 것으로, 조원끼리 부족한 부분을 서로 도와가며 프로젝트를 진행하기로 하였음.

2) 회의록

1차)

역할분담 실시

CPU의 기본 구조와 원리에 대해 살핌

2차)

각 조원이 인터넷 검색을 통하여 간단한 verilog CPU 코드들을 가져와 조사 후 분석하여 충분히 숙지하는 시간을 가짐

추후 조원들에게 설명해주고 서로 조사한 각 코드의 공통점과 차이점을 알아내어 토론하는 시간을 가질 예정

3차)

2차 때 회의했던 코드에 대해 각자 설명하였고 공통점과 차이점에 대해 토론함

해당주차 수업시간에 학습한 내용(Half-Adder, Full-Adder)에 대해 복습함

4차)

해당주차 수업시간에 학습한 내용(D Latch, D flip-flop)에 대해 복습함

5차)

지난주차에 학습했던 내용(D Latch, D flip-flop)에 관한 verilog 코드들을 Chat-GPT를 활용하여 구현해보고 vivado 프로그램을 사용하여 파형을 시뮬레이션 해봄

해당주차 수업시간에 학습한 내용(FSM)에 대해 심도 있게 복습함

6차)

ZOOM 회의 간 미래도서관 그룹스터디룸에서 대면 회의 계획

github로 16bit CPU 코드 작성 및 ALU 시뮬레이션 실행하였으나 오류발생 후 수정할 방법 고안하였음

7차)

중간보고서 발표 간 교수님께서 피드백해주신 내용 검토 및 수정

(입출력 코드 가독성 개선 ex)MB -> MemoryBus, DA->DataReg_A)

13주차 과제 간 궁금한 부분을 서로 질문하고 알려줌.

최종보고서 발표 대비 대면 회의 계획

8차)

최종보고서 발표 대비 6월 9일 대면 회의

입출력 코드 가독성 개선

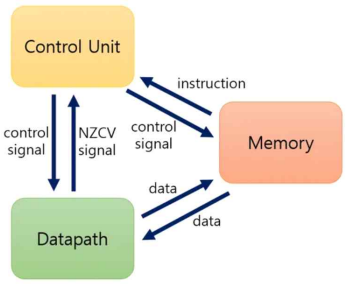
최종보고서 및 PPT자료 작성 (Run synthesis 실행 및 schematic 추가)

3) 프로젝트 개요

mini CPU 프로젝트의 주제로 mini CPU를 Xilinx사의 vivado 또는 Intel사의 Quartus 프로그램을 활용하여 HDL(verilog)로 설계하는 프로젝트로 우리 6조는 vivado 프로그램을 활용하여 간단한 16bits CPU를 주제로 설계하기로 결정하였음.

Instruction fetch - Decode - Execution 순의 CPU 처리과정을 기반으로 하여 설계.

4) 프로젝트 진행현황



위 그림에서 볼 수 있듯이 CPU는 크게 컨트롤유닛, 메모리, 데이터패스로 분류할 수 있다. 이때 메모리는 데이터를 저장하는 역할을 하며 데이터패스는 메모리에서 데이터를 가져와(fetch) 연산을 수행(decode, execute)하는 역할을 한다. 컨트롤유닛은 이러한 전반적인 과정을 통제하는 역할을 수행한다.

우리 6조는 이중에서 데이터패스에 집중하여 vivado로 16bits CPU(데이터패스)를 설계하기로 결정했다.

최종보고서의 개선 및 추가사항으로는 다음과 같다.

1. 입출력 코드의 가독성 개선
2. Immediate(Load, Add), Branch(zero , negative), jump 기능 추가
3. 2.를 구현하기 위해 PC counter와 Memory 부분 추가
4. control word bit에 의해 instruction bit (opcode)가 결정되도록 수정
5. 테스트벤치에서 주어진 FunctionSelect와 DataReg_A, AddressReg_A, AddressReg_B에 의해 instruction이 작동되도록 수정
6. 합성(Synthesis) 수행 및 도식(Schematic) 추가

설계 과정에서 opcode의 [15:13]와 [12:9]를 나누어 전자는 opcode라고 정의하였고, 후자는 FunctionSelect로 정의하였다.

이를 간략히 설명하면,

1. Positive Edge Clock (1ns) 마다 모든 값들은 변화한다.
2. MemoryBus, MemoryData, ReadWrite, MemoryWrite, ProgramReg, Jump, Branch는 Control word bit 로 opcode를 결정한다.
3. 결정된 opcode와 주어진 FunctionSelect, DataReg_A, AddressReg_A, AddressReg_B 값에 따라 Instruction이 실행된다.
4. Instruction code(opcode + FunctionSelect + DataReg_A + AddressReg_A + AddressReg_B를 합친 16bit binary code) 는 ProgramCounter_counter에 저장되며, ProgramCounter 값은 ProgramCounter_counter의 주소값이다.
5. ProgramCounter 값은 default로 매 시행마다 ProgramCounter <= ProgramCounter +1 되며, 예외로 jump, Branch가 일어났을 경우 해당 위치로 ProgramCounter 값이 이동한다.

데이터패스의 verilog code를 다음과 같이 작성하였다.

datapath.v × Test.v × Untitled 1 ×

C:/Users/dlagy/digital_logic_16bits_cpu_final/digital_logic_16bits_cpu_final.srcs/sources_1/new/datapath.v

```
1   `timescale 1ns / 1ps
2
3   module datapath(
4    input clk,MemoryBus,MemoryData,ReadWrite,
5    input [2:0] DataReg_A,AdderssReg_A,AddressReg_B,
6    // opcode 000_FS
7    input ProgramReg,Jump,Branch,MemoryWrite,
8    input [3:0] FunctionSelect,
9    output reg [2:0] ProgramCounter,opcode,
10   output reg [15:0] da,aa,ab
11   );
12   // register 초기값 '1'으로 설정.
13   reg [15:0] register[7:0];
14   integer i;
15   initial begin
16   ProgramCounter = 0;
17   for (i=0;i<8;i=i+1)
18         register[i] <= 16'd1;
19   end
20   // Memory 초기값 '5'으로 설정
21   reg [15:0] Memory[7:0];
22   initial begin
23   for (i=0;i<8;i=i+1)
24         Memory[i] <= 16'd5;
25   end
26   // PC는 PC값들이 저장된 PC_counter의 주소역할.
27   reg [15:0] ProgramCounter_counter[2:0];
28   //opcode 담고 있는 pc counter 설정.
29   always @(posedge clk)
30   begin
31   da = register[DataReg_A]; aa= register[AdderssReg_A]; ab = register[AddressReg_B];
32   // Control word bit에 의한 opcode 설정
33   if(~MemoryBus & ~MemoryData & ReadWrite & ~MemoryWrite & ~ProgramReg)
34       opcode = 3'b000;    //FS
35   else if(~MemoryBus & MemoryData & ReadWrite & ~MemoryWrite & ~ProgramReg)
36       opcode = 3'b001;    //Memory read
37   else if(~MemoryBus & ~ReadWrite & MemoryWrite & ~ProgramReg)
38       opcode = 3'b010;    //Memory Load
39   else if(MemoryBus & ~MemoryData & ReadWrite & ~MemoryWrite & ~ProgramReg)
40       opcode = 3'b100;    //FS constant(Immediate)
41   else if(~ReadWrite & ~MemoryWrite & & ProgramReg & ~Jump & ~Branch)
42       opcode = 3'b110;    //Branch on zero(Z)
43   else if(~ReadWrite & ~MemoryWrite & & ProgramReg & ~Jump & Branch)
44       opcode = 3'b110;    //Branch on negative(N)
45   else if(~ReadWrite & ~MemoryWrite & & ProgramReg & Jump)
46       opcode = 3'b111;    //Jump
47   // opcode에 따른 데이터 변화
48   case(opcode)
49   3'b000:
50       case(FunctionSelect)                    // function code 설정
51          4'b0000: da = aa;          //Move A
52          4'b0001: da = aa +1;       //increment + 1
53          4'b0010: da = aa + ab;     //Add
54          4'b0011: da = aa + ab + 1; //Add + 1
55          4'b0100: da = aa + ~ab;    //1's Subtract
56          4'b0101: da = aa + ~ab + 1;//2's Subtract
57          4'b0110: da = aa - 1;      //Decrement -1
58          4'b0111: da = aa ;         //Move A
59          4'b1000: da = aa & ab ;    //And
60          4'b1001: da = aa | ab ;    //OR
61          4'b1010: da = aa ^ ab ;    //XOR
62          4'b1011: da = ~aa ;        //Not
63          4'b1100: da = ab ;         //Move B
64          4'b1101: da = ab >> 1;     //Shift Right
65          4'b1110: da = ab << 1;     //Shift Left
66       endcase
67   //constant 값 사용하여 Immediate 사용
68   3'b100:
69          case(FunctionSelect)                    // function code 설정
70          4'b0000: da = aa;          //Move A
71          4'b0001: da = aa +1;       //increment + 1
72          4'b0010: da = aa + AddressReg_B;     //Add immediate
73          4'b0011: da = aa + AddressReg_B + 1; //Add + 1
74          4'b0100: da = aa + ~AddressReg_B;    //1's Subtract
75          4'b0101: da = aa + ~AddressReg_B + 1;//2's Subtract
76          4'b0110: da = aa - 1;      //Decrement -1
77          4'b0111: da = aa ;         //Move A
78          4'b1000: da = aa & AddressReg_B ;    //And
79          4'b1001: da = aa | AddressReg_B ;    //OR
80          4'b1010: da = aa ^ AddressReg_B ;    //XOR
81          4'b1011: da = ~aa ;        //Not
82          4'b1100: da = AddressReg_B ;         //Load Immediate
83          4'b1101: da = AddressReg_B >> 1;     //Shift Right
84          4'b1110: da = AddressReg_B << 1;     //Shift Left
85       endcase
86       3'b001:
87          da = Memory[aa];            //Memory load
88   3'b010:
89       Memory[aa] <= ab ;          //Memory store
90   3'b110:
91       case(FunctionSelect)
92           4'b0000: if(~aa) ProgramCounter = ProgramCounter + ab; else ProgramCounter <= ProgramCounter + 1;     // Branch on Zero
93           4'b0001: if(~aa[15]) ProgramCounter = ProgramCounter + ab; else ProgramCounter <= ProgramCounter + 1; // Branch on negative
94       endcase
95   3'b111:
96       ProgramCounter <= aa;   // aa위치로 PC 이동
97   endcase
98   // 예외 3'b110일 때 제외하고  PC counter increment
99   ProgramCounter_counter[ProgramCounter] <= {opcode,FunctionSelect,DataReg_A,AdderssReg_A,AddressReg_B};
100  register[DataReg_A] = da;
101  if (opcode != 3'b110)
102      ProgramCounter <= ProgramCounter +1;
103  end
104  endmodule
105
```

중간보고서의 Datapath 코드를 수정하여 control word bit를 수행하는 ProgramReg, Jump, Branch, MemoryWrite를 추가하였다. 이 값들은 기존 control word bit들(MemoryBus, MemoryData, ReadWrite)과 함께 opcode를 결정하며 opcode에 따라 FunctionSelect의 사용이 달라지도록 설계하였다.

Memory를 16bit*8로 정의하였으며 초기값은 '5'로 설정하였다. 기존에 있던 Datain, Dataout 변수는 Memory의 주소값에 따라 Memory Load, Memory Store가 되도록 하였으므로 삭제하였다.

Opcode에 의해 각각 FunctionSelect(register, constant), Memory(read, Load), Branch(0,-), Jump등이 일어나도록 하였으며, 해당 수행값들은 ProgramCounter_counter에 저장된다.

#1~3ns에 LD R2, R3

#3~5ns에 ADI R2, R2, 1

#5~7ns에 ADD R3, R2, R3를 수행하는 테스트벤치의 verilog code를 다음과 같이 작성하였다.

datapath.v * | Test.v

C:/Users/dlagy/digital_logic_16bits_cpu_final/digital_logic_16bits_cpu_final.srcs/sim_1/new/Test.v

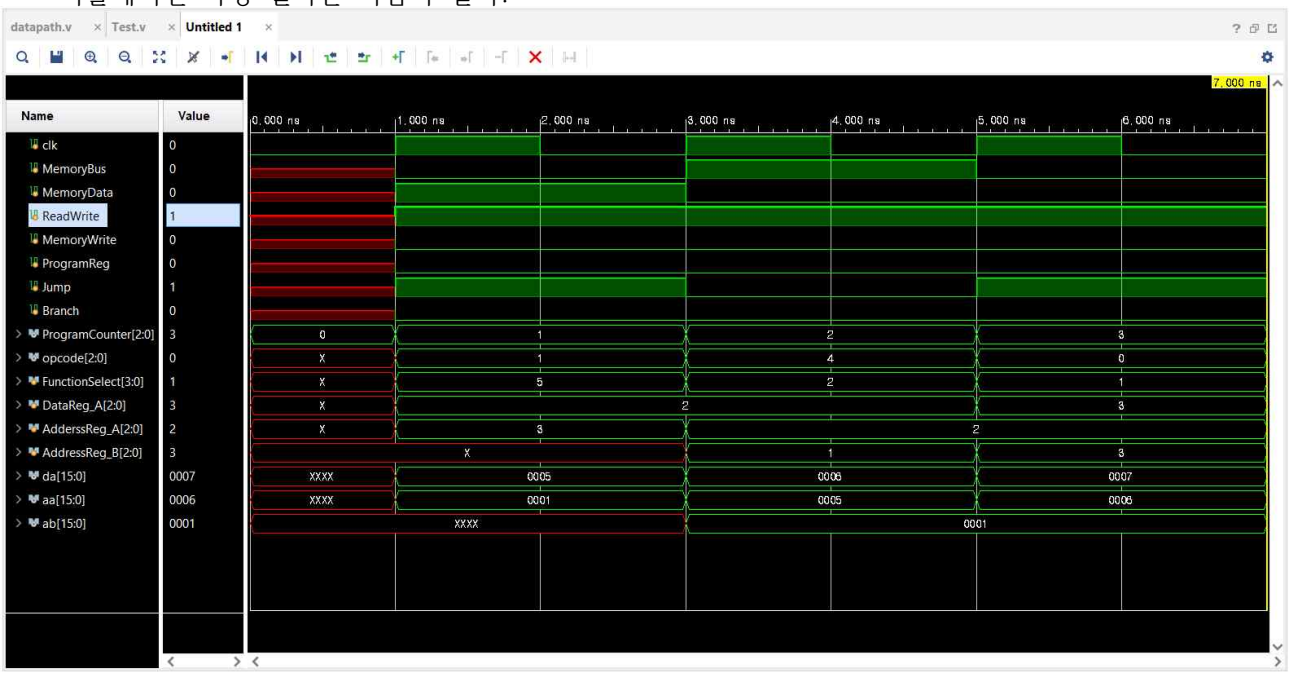
```
1   `timescale 1ns / 1ps
2
3   module testbench;
4       reg clk,MemoryBus,MemoryData,ReadWrite,MemoryWrite,ProgramReg,Jump,Branch;
5       wire [2:0] ProgramCounter;
6       wire [2:0] opcode;
7       reg [3:0] FunctionSelect;
8       reg [2:0]DataReg_A,AdderssReg_A,AddressReg_B;
9       wire [15:0] da,aa,ab;
10          // testbench 변수와 source code counter 변수 연결
11  datapath uut(
12  .clk(clk),.MemoryBus(MemoryBus),.MemoryData(MemoryData),.ReadWrite(ReadWrite),.DataReg_A(DataReg_A),.AdderssReg_A(AdderssReg_A),.AddressReg_B(AddressReg_B),.FunctionSelect(FunctionSelect),.da(da),.aa(aa),.ab(ab),.MemoryWrite(MemoryWrite),
13  .ProgramReg(ProgramReg),.Jump(Jump),.Branch(Branch),.opcode(opcode),.ProgramCounter(ProgramCounter)
14  );
15      always begin
16      #1 clk = ~clk;  // clk 신호는 1ns마다 변화
17      end
18      initial begin
19          clk = 0;
20          #1 DataReg_A = 3'b010; AdderssReg_A = 3'b011; FunctionSelect=4'b0101; MemoryBus =0; MemoryData = 1; ReadWrite = 1; MemoryWrite = 0;  ProgramReg = 0; Jump = 1; Branch = 0;              // LD R2,R3
21          #2 DataReg_A = 3'b010; AdderssReg_A = 3'b010; AddressReg_B = 3'b001; FunctionSelect=4'b0010; MemoryBus =1; MemoryData = 0; ReadWrite = 1; MemoryWrite = 0;  ProgramReg = 0; Jump = 0; Branch = 0;          //ADI R2,R2,1
22          #2 DataReg_A = 3'b011; AdderssReg_A = 3'b010; AddressReg_B = 3'b011; FunctionSelect=4'b0001; MemoryBus =0; MemoryData = 0; ReadWrite = 1; MemoryWrite = 0;  ProgramReg = 0; Jump = 1; Branch = 0;          //ADD R3,R2,R3
23          #2
24          $finish;
25      end
26  endmodule
27
```

“#1~3ns에 LD R2, R3”는 R3 레지스터 값을 R2 레지스터에 로드(LD),

“#3~5ns에 ADI R2, R2, 1”은 R2 레지스터의 값을 1만큼 증가(ADD Immediate),

“#5~7ns에 ADD R3, R2, R3”는 R2 레지스터와 R3 레지스터의 값을 더한 후 그 결과를 다시 R3 레지스터에 저장한다는 의미이다.

시뮬레이션 파형 결과는 다음과 같다.



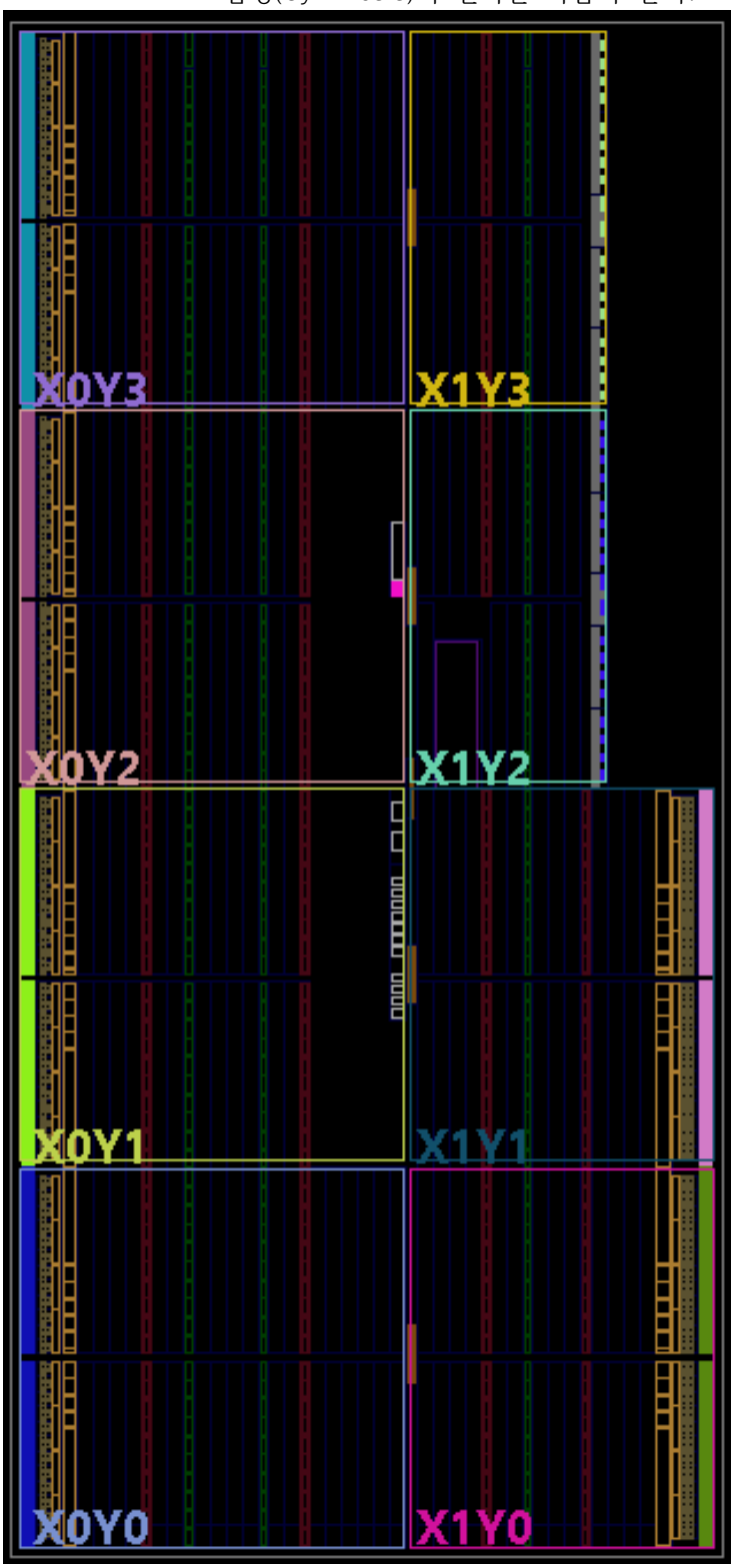
1. 1~3ns에 Load R2(da) = Memory (R3(aa))를 수행한다. Memory값의 초기값이 모두 5이고, register의 초기값은 모두 1이므로 처음 불러온 R2값은 register의 초기값인 1이고, Memory[R3]값을 Load하여 5가 된다.

2. 3~5ns에 ADD Immediate(constant) R2(da) = R2(aa) + 1(AddressReg_B, constant)를 수행한다. R2(aa)는 1의 수행결과 현재 5이며, R2값에 AddressReg_B를 constant로 사용하여 1을 더해준다. 결과값은 R2(da) = 6이 된다.

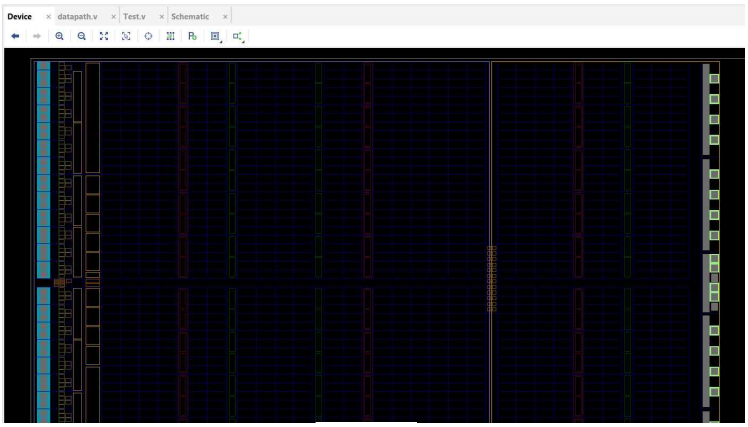
3. 5ns~7ns에서 ADD R3(da) = R2(aa) + R3(ba)를 수행한다. R2(aa)는 2의 수행결과 현재 6이며 R3(ab)값은 초기 register값 1이므로 결과값은 R3(da) = 7이 된다.

따라서 결과 파형이 문제없이 동작했음을 알 수 있다.

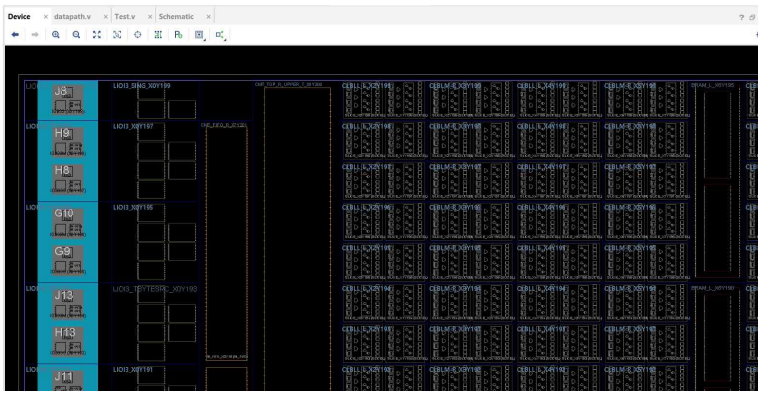
합성(Synthesis)의 결과는 다음과 같다.



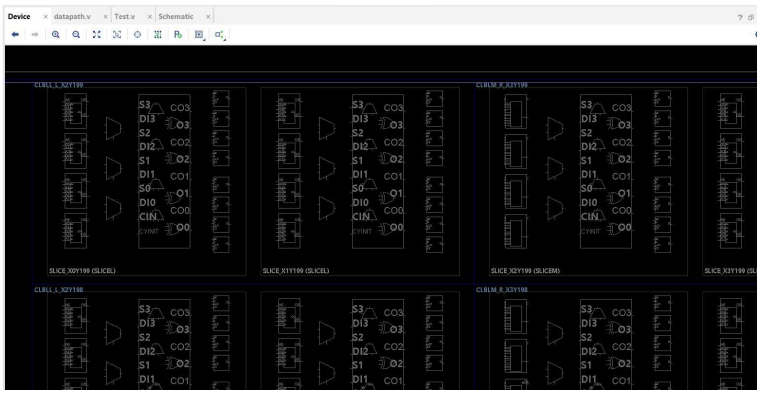
→
(X0Y3
확대)



↓ 확대

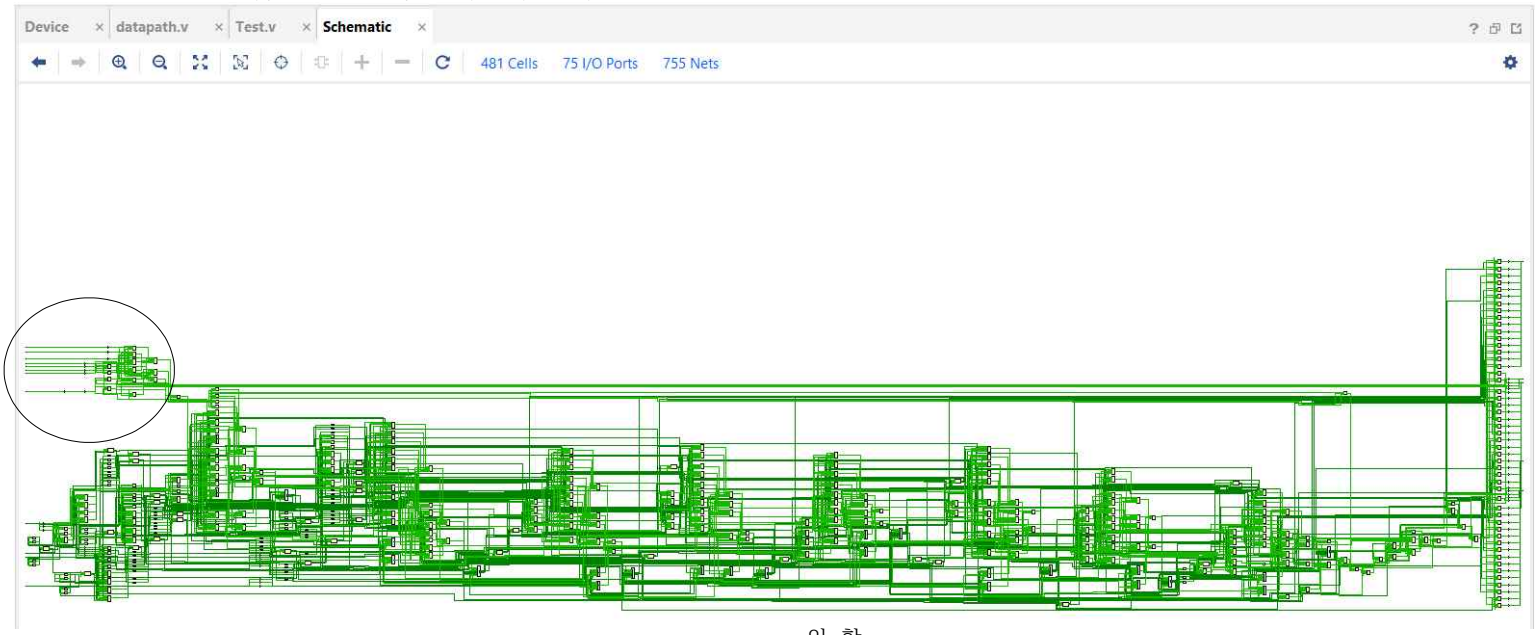


↓ 확대

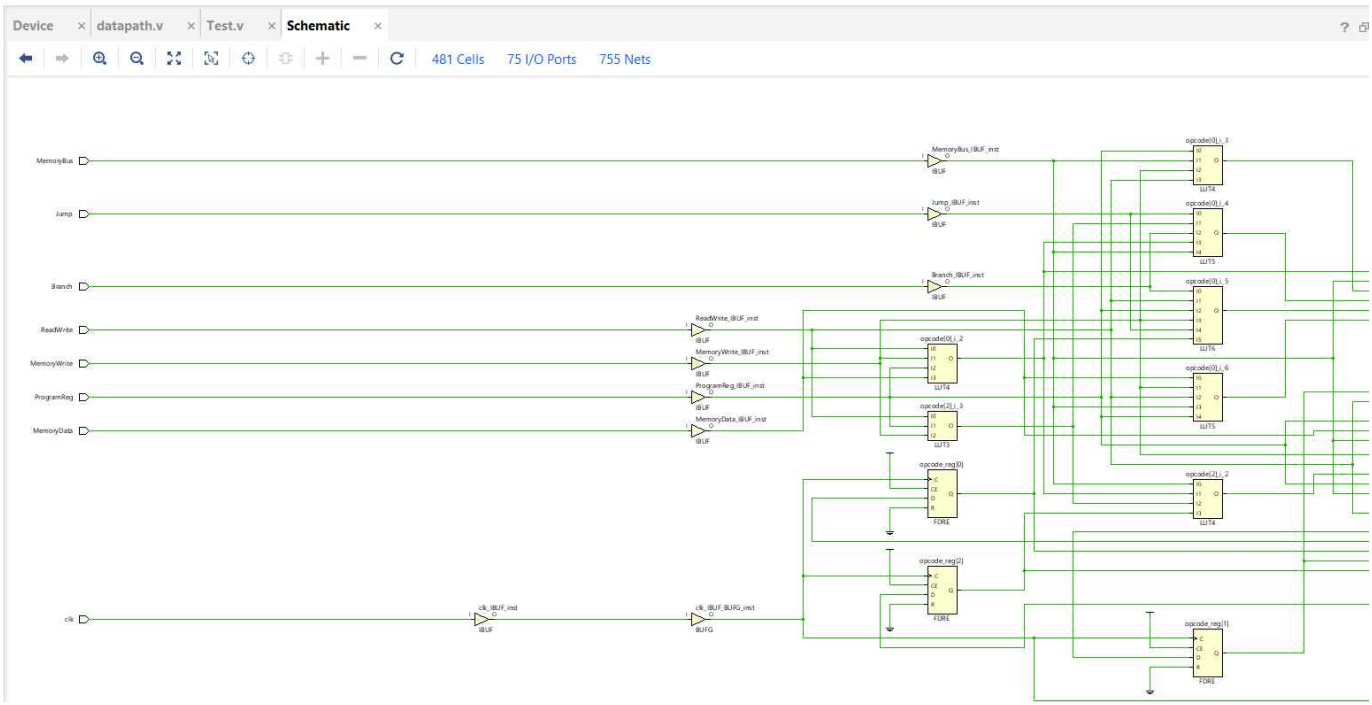


합성(Synthesis)은 논리회로의 구조를 시각적으로 확인할 수 있게 해주는 기능이다. 이를 통해 설계자는 합성된 논리회로의 세부적인 내용, 계층 구조, 사용된 자원 등을 확인할 수 있다.

도식(Schematic)은 다음과 같다.

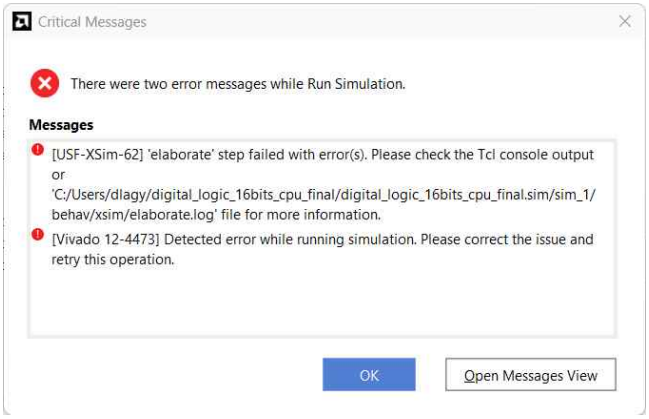


입력부분 확대



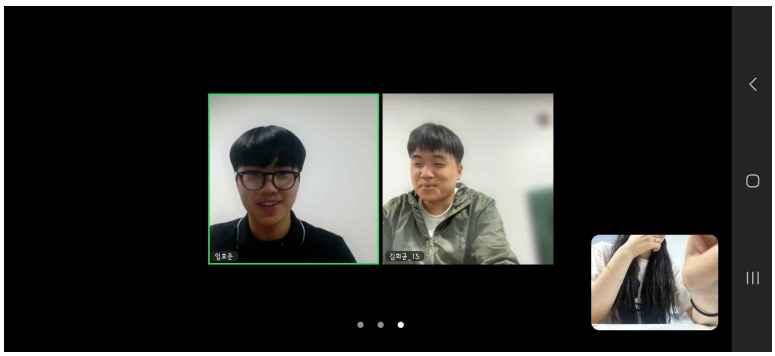
여기서 사용된 Net의 개수는 755개, Cell의 개수는 481개임을 확인할 수 있다.

5) 실패사례



코드의 가독성 개선 중에 입력코드 하나를 변경하지 못하여 오류가 발생하였다.
->오류 메시지를 확인하고, 해당 부분을 수정하여 문제를 해결하였다.

6) 모임 사진



2024.05.22.(수) ZOOM회의

2024.05.28.(화) 대면회의

2024.06.09.(일) 대면회의