**컴퓨터 구조 Project**

**0440298 허 인 녕**

**1. 설계 방법 설명**

Simple MIPS의 각 구조들을 module로 설계한 후 top에서 각각의 port들을 이어줌으로써 simple MIPS를 구현하였습니다.

SimpleMIPS.v 파일을 top으로 설정하였고, instmem.v에서는 instruction memory, register.v에서는 regesters, alu.v에서는 alu, datamem.v에서는 data memory, control.v에서는 control signal, mux.v에서는 multiplexer, adder.v에서는 adder를 구현하였습니다.

tb\_SimpleMIPS.v에서 testbench를 만들어주어 각 과정을 확인하도록 하였습니다.

**2. Module 설명**

**SimpleMIPS.v** 에서는 pc값을 rstn(reset) 신호에 따라 rstn = 1일 경우 pc값을 reset 시켜서 pc = 0으로 만들어주고 rstn 1 일 경우 pc는 mux에서 나온 pc 값이 이어지도록 하였습니다. 그리고 im\_write 신호를 받아서 im\_write = 1일 경우 instruction memory에 처음에 지정할 명령어를 저장하도록 설계하였습니다.

**adder.v** 에서는 입력값으로 [31:0] data\_1, [31:0] data\_2를 받은 후 더하기를 하여 결과값을 내보내도록 설계하였습니다.

**mux.v** 에서는 입력값으로 [31:0] data\_1, [31:0] data\_2를 받은 후 control signal(muxctrl) 을 받아서 muxctrl = 0일 경우 data\_1, 1일 경우 data\_2가 결과값으로 나가도록 설계하였습니다.

**instmem.v**에서는 im\_write = 1일 경우, always문에서 처음에 지정할 명령어를 [31:0] mem[1023:0] 에 저장하도록 하였고, im\_write = 0일 경우 assign문을 통해 mem[addr]이 instruction memory의 output인 inst로 나가도록 하였습니다.

**register.v**에서는 reg [31:0] dvalue[4:0]를 설정하여 register 값들을 저장할 수 있도록 설정하였고 instruction memory에서 들어오는 명령어(instdata)를 이용하여 source register(rs\_n)는 instdata [25:21], target register(rt\_n)는 instdata[20:16], destination register (rd\_n)는 instdata [15:11] 로 assign 하였습니다. register에서 나가는 값, Read data1(data\_s) 는 dvalue[rs\_n], Read data2 (data\_t) 는 dvalue[rt\_n]으로 설정하였습니다. 그리고 control signal인 regwrite와 alusrc를 받아서 regwrite = 1일 경우 data를 write하도록 하였고, alusrc = 0 일 경우 alu전에 있는 mux에서 rt값을 받아서 alu에서 연산하는 것이므로 write data에서 저장될 register는 rd가 되도록 하였고, alusrc = 1 일 경우 mux에서 immediate 값을 받아서 addi나 lw 등을 연산하는 것이므로 저장될 register는 rt가 되도록 하였습니다.

**control.v**에서는 instruction memory에서 들어오는 명령어(instdata)를 이용하여 opcode (instdata[31:26]) 를 추출하고 opcode가 1일 경우 load word, 2일 경우 store word, 3일 경우 add, 4일 경우 addi, 5일 경우 sub, 6일 경우 division이 수행되는 control signal이 나가도록 설계하였습니다. Control signal은 총 10개의 bit로 구성이 되어 있는데, sigctrl[0] = PCSrc, sigctrl[1] = RegWrite, sigctrl[2] = ALUSrc, sigctrl[6:3] = ALU operation, sigctrl[7] = MemWrite, sigctrl[8] = MemRead, sigctrl[9] = MemtoReg로 설정하였습니다. 예를 들어, lw인 경우 sigctrl = 10'b 0\_1\_0\_0001\_1\_1\_0 이 되는데, PCSrc는 0, 즉 pc + 4이고, Register에 write하므로 RegWrite는 1, immediate value를 alu 연산에 사용하므로 ALUSrc는 1, ALU에서 source register의 주소값과 immediate 값을 더해주므로 더하기 연산을 해야 하고 결국 ALU operation은 0001, Data memory에는 write하지 않고 read만 하므로 MemWrite는 0, MemRead는 1, 그리고 마지막으로 Data memory에서 나온 값을 register에 보내므로 MemtoReg는 0으로 설정해 주었습니다.

**datamem.v**에서는 [31:0] mem[1023:0] 를 설정하여 data 값들을 저장할 수 있도록 설정하였고, control signal인 memwrite와 memread를 받아서 store word일 경우 memwrite, load word일 경우 memread 하도록 하였습니다.

**alu.v**에서는 register 또는 immediate value에서 들어온 값들을 연산할 수 있도록 설계하였습니다. 이 때 control signal로 aluctrl을 받아서 aluctrl = 1일 경우 더하기, aluctrl = 2일 경우 빼기, aluctrl = 3일 경우 나누기를 수행하도록 하였습니다.

**3. 나눗셈을 수행하는 알고리즘**

alu.v 안의 module alu 에서 나눗셈을 수행할 때 나누기 과정을 저장할 수 있는 reg [31:0] divresult를 설정하여 주었다. divresult의 초기값은 0으로 지정해 주었다. A 나누기 B의 몫을 구하는 과정은 다음과 같다.

먼저 A와 B의 크기를 비교하여 B가 A보다 더 큰지를 비교한다. 비교 결과, B가 A보다 더 클 경우 A 나누기 B의 몫은 0이다.

A가 B보다 클 경우 B를 한 칸 왼쪽으로 shift 시킨다. 그리고 다시 A와 (B << 1)의 크기를 비교한다. ( B << 1 )이 A보다 클 경우 A - B를 수행하고 A - B의 값을 결과값으로 rd에 내보낸다. 그리고 divresult에 1을 더해준다. 그리고 rd에 보내진 값(A-B)을 다시 B와 비교하여 큰지 작은지를 판단하고 B가 (A-B)보다 클 경우 나눗셈의 몫은 divresult(= 1) + 0 = 1 이 되도록 해준다.

위의 경우를 반복해서 수행한다면, A와, B를 shift시킨 값들의 크기를 계속 비교해서 2진법으로 나타내어지는 몫의 최대 자릿수에서 A - (B의 shift된 값)을 수행하여 주고, 그에 해당하는 몫의 값을 divresult에 더해주며 그 결과값을 다시 크기 비교하여 밑의 자릿수의 몫을 계속 구해나가는 방식으로 계산하면 결국 원하는 최종 나눗셈의 몫을 구할 수 있다.

위의 방식을 예를 들어 보자면 8 / 3을 수행한다고 하자. 8은 이고 3은 이다. 나누기의 첫번째 cycle에서 와 를 비교하여 이 더 크므로 를 한 칸 왼쪽으로 shift 시킨다. 그러면 은 가 되고, 이것을 다시 와 비교한다. 여전히 가 크므로 다시 한 칸 왼쪽으로 를 shift 시키고 그 값은 이 된다. 은 보다 크므로 에서 를 shift 시키기 전 값인 을 빼주고 - 은 가 된다. 그리고 한번 shift 시켜준 값을 뺐으므로 divresult에 을 더해준다. 나누기의 두번째 cycle에서 - 를 다시 와 비교하고 이 더 크므로 divresult에 +0을 해주고 결국 결과값은 2가 되어 나누기의 과정은 종료된다.

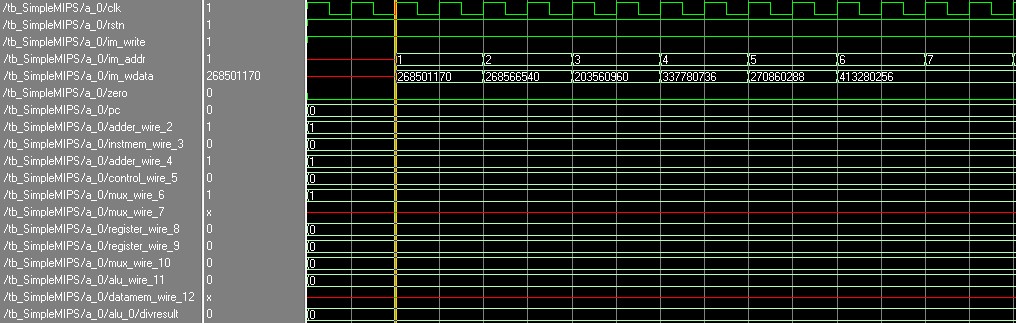
위의 과정은 32bit 숫자들 간에 일어나는 연산이므로 32사이클 안에 수행될 수 있다.

**4. verilog coding 결과 파형 분석**

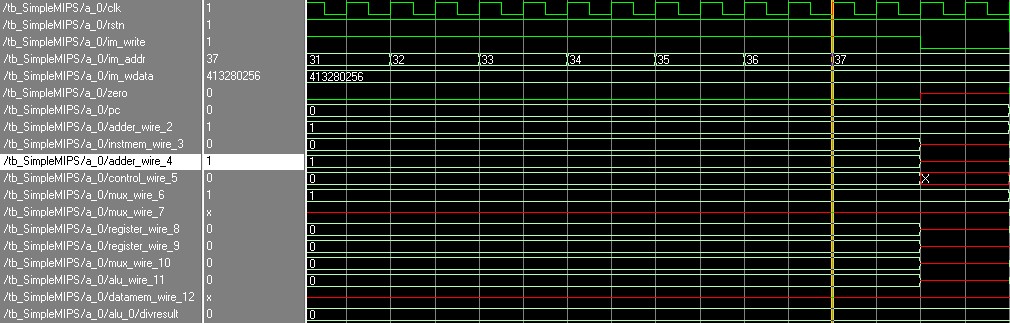
testbench 파일인 tb\_SimpleMIPS.v에서 clock(clk)을 주기가 10ns가 되도록 5ns마다 값이 바뀌게 설정을 해주고, im\_write와 rstn(reset)을 1로 설정하여 초기에 주어져야 하는 명령 data들을 지정해 주었습니다. testbench에서 설정해준 명령어는 총 37개로 밑의 wave graph를 보면 im\_addr이 37까지 증가함을 확인할 수 있습니다. 그런 후 im\_write를 0으로 설정하고 reset신호(rstn)도 0으로 변환시켜 주었습니다.

testbench에서 설정해준 37개의 instruction memory에 들어가는 명령어는 첫번째로 im\_wdata <= 32'b 000100\_00000\_00001\_0000\_0000\_1011\_0010 ( addi $s1, $s0, 178 ) 를 넣어 주었는데, 이것은 opcode[31:26] = 4, 즉 addi 명령을 지정하였고, [25:21] = $s0 = 0, [20:16] = 1 = $s1으로 설정하였으며 immediate value인 im\_wdata[15:0] = 178이 되도록 설정하였습니다. 이와 같은 방식으로 두번째 명령어에는 addi $s2, $s0, 12를 넣어주었고, 이렇게 저장된 $s1 = 178과 $s2 = 12를 이용하여 각종 연산을 할 수 있도록 명령어를 구성하였습니다. 세번째 명령어에는 add $s3, $s1, $s2를 넣어서 덧셈 연산을 시험해 보았고 다음은 sub $s4, $s1, $s2를 넣어서 뺄셈 연산을 시험해 보았으며 마지막으로 프로젝트의 최종 목표인 나눗셈을 수행하도록 addi $s5, $s1, 0를 넣어서 $s5에 피제수 178을 넣어준 후 div $s5, $s5, $s2를 32 cycle동안 수행하여 178 12가 연산되도록 설정해 주었습니다.

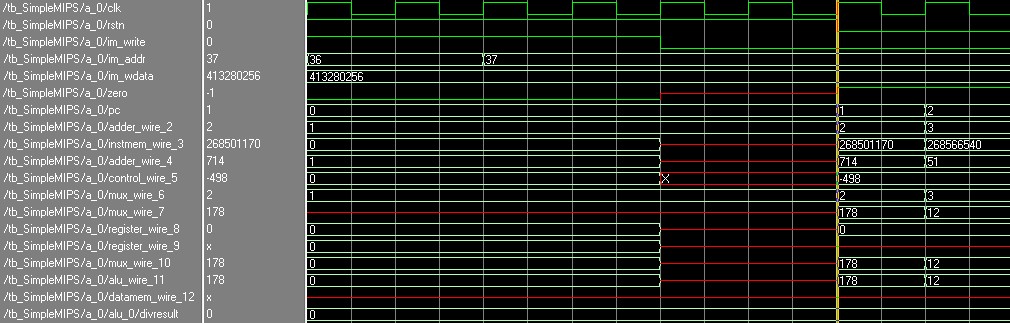
그 결과를 wave로 확인해보면 다음과 같습니다. ( wave를 추가하실 때 나눗셈의 결과값을 확인하기 위하여 add wave sim:/tb\_SimpleMIPS/a\_0/alu\_0/divresult 도 해주시기 바랍니다.)



일단 처음에 rstn과 im\_write가 1로 설정됨으로써 초기화를 시켜주는 상태로 만들어 주었고, im\_addr(instruction memory에 저장되는 명령어의 주소)가 1부터 2, 3 ∙∙∙ 으로 증가할 때마다 im\_addr = 1부터 32'b 000100\_00000\_00001\_0000\_0000\_1011\_0010 ( addi $s1, $s0, 178 )를 시작으로 tb\_SimpleMIPS.v에서 설정한 명령어들이 입력되고 있음을 확인할 수 있습니다. im\_addr = 6부터는 32'b 000110\_00101\_00010\_0010\_1000\_0000\_0000 ( div $s5, $s5, $s2 )가 입력되고 있고, im\_addr = 37까지 div $s5, $s5, $s2가 입력되고 있음을 밑의 wave 그래프를 통해 확인할 수 있습니다.

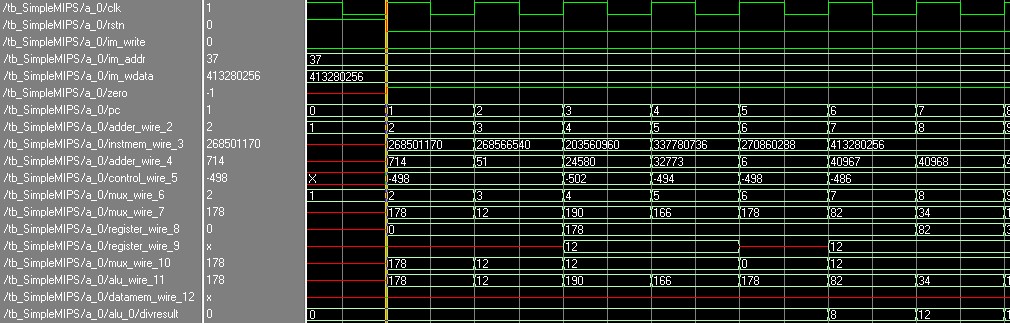


수행되어야 할 명령어들을 instruction memory에 모두 집어넣은 후 im\_write와 rstn을 0으로 설정해 줌으로써 이제 pc로부터 주소값을 받아서 명령어가 수행되도록 하였습니다. 그 과정은 다음과 같습니다.



im\_write와 rstn이 "0"이 된 후 pc값이 0에서부터 1씩 증가하고 있음을 확인할 수 있습니다. 그리고 instmem\_wire\_2(instruction memory에서 나오는 명령어)가 처음에 수행되도록 설정한 명령어를 그대로 내보내고 있음을 확인할 수 있습니다. adder\_wire\_2는 PC의 다음 주소값을 나타내고 있고 adder\_wire\_4는 PCSrc 신호를 받는 Mux에 들어가는 adder로부터 나온 신호를 나타내며, mux\_wire\_6은 PC로 들어가는 신호를 나타냅니다. control\_wire\_5는 control signal을 나타내며 그 밑에 설정된 wire들 또한 모두 simple MIPS의 각 module들을 연결시켜주고 있는 wire들을 나타내고 있습니다.

PC의 주소값 1부터 명령어가 수행되는 과정을 살펴보자면 다음과 같습니다.



PC = 1에서는 addi $s1, $s0, 178를 수행하는데, register\_wire\_8에서 $s0 = 0 값이 나오고 있고 register\_wire\_10에서 mux를 통과한 sign extend된 immediate 값인 178이 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 0+178의 결과값인 178이 나오고 있음을 알 수 있고, mux\_wire\_7에서 나오는 178이 Register(rt = $s1)의 Write data에 저장되도록 연결되어 있습니다. 이 모든 과정은 clock의 한 cycle에서 수행되고 있음을 확인할 수 있습니다.

PC = 2에서는 addi $s2, $s0, 12를 수행하는데, PC = 1에서와 같은 방식으로 더하기가 진행되고 있음을 알 수 있습니다. register\_wire\_8에서 $s0 = 0 값이 나오고 있고 register\_wire\_10에서 mux를 통과한 sign extend된 immediate 값인 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 0+12의 결과값인 12가 나오고 있음을 알 수 있고, mux\_wire\_7에서 나오는 12가 Register(rt = $s2)의 Write data에 저장되도록 연결되어 있습니다.

PC = 3에서는 add $s3, $s1, $s2를 수행하는데, register\_wire\_8에서 $s1 = 178이 나오고 있고, register\_wire\_10에서는 mux를 통과한 $s2 = 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 178+12의 결과값인 190이 나오고 있음을 알 수 있고, mux\_wire\_7에서 나오는 190이 Register(rd = $s3)의 Write data에 저장되도록 연결되어 있습니다.

PC = 4에서는 sub $s4, $s1, $s2를 수행하는데, register\_wire\_8에서 $s1 = 178이 나오고 있고, register\_wire\_10에서는 mux를 통과한 $s2 = 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 178-12의 결과값인 166이 나오고 있음을 알 수 있고, mux\_wire\_7에서 나오는 166이 Register(rd = $s4)의 Write data에 저장되도록 연결되어 있습니다.

PC = 5에서는 addi $s5, $s1, 0를 수행하는데, register\_wire\_8에서 $s1 = 178 값이 나오고 있고 register\_wire\_10에서 mux를 통과한 sign extend된 immediate 값인 0이 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 178+0의 결과값인 178이 나오고 있음을 알 수 있고, mux\_wire\_7에서 나오는 178이 Register(rt = $s5)의 Write data에 저장되도록 연결되어 있습니다.

PC = 6에서는 본격적으로 나눗셈인 div $s5, $s5, $s2이 수행되는데, 그 wave 그래프는 다음과 같습니다.



PC = 6에서부터 div $s5, $s5, $s2이 수행되는데 처음에 register\_wire\_8에서 $s5 = 178이 나오고 있고, register\_wire\_10에서는 mux를 통과한 $s2 = 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 나눗셈 알고리즘을 통한 결과값인 82가 나오고 있음을 알 수 있고, mux\_wire\_7에서 나오는 82가 Register(rd = $s5)의 Write data에 새로이 저장되도록 연결되어 있습니다. 여기서 나눗셈 알고리즘을 살펴보자면 $s5 = 178 = 이고 $s2 = 12 = 인데 을 3번 left shift 시킨 후 빼주게 되므로 - = = 82가 되어서 ALU result로 나오고 있습니다. 한편 divresult를 살펴보면 3번 shift 시켜주었으므로 divresult = 0+8 = 8이 됨을 확인할 수 있습니다.

PC = 7에서 다시 div $s5, $s5, $s2을 수행되는데 register\_wire\_8에 PC = 6에서 변환된 값인 $s5 = 82가 나오고 있고 register\_wire\_10에서는 여전히 mux를 통과한 $s2 = 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 나눗셈 알고리즘을 통한 결과값인 34가 나오고 있고, mux\_wire\_7에서 나오는 34가 Register(rd = $s5)의 Write data에 새로이 저장되도록 연결되어 있습니다. 나눗셈 알고리즘을 살펴보자면 $s5 = 82 = 이고 $s2 = 12 = 인데 2번 left shift 시킨 후 빼주게 되므로 - = = 34가 되어서 ALU result로 나오고 있습니다. 한편 divresult를 살펴보면 2번 shift 시켜주었고 기존에 divresult의 값이 8로 저장되어 있었으므로 divresult = 8 + 4 = 12가 됨을 확인할 수 있습니다.

PC = 8에서도 div $s5, $s5, $s2을 수행되는데 register\_wire\_8에 PC = 7에서 변환된 값인 $s5 = 34가 나오고 있고 register\_wire\_10에서는 여전히 mux를 통과한 $s2 = 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 나눗셈 알고리즘을 통한 결과값인 10이 나오고 있고, mux\_wire\_7에서 나오는 10이 Register(rd = $s5)의 Write data에 새로이 저장되도록 연결되어 있습니다. 나눗셈 알고리즘을 살펴보자면 $s5 = 34 = 이고 $s2 = 12 = 인데 1번 left shift 시킨 후 빼주게 되므로 - = = 10이 되어서 ALU result로 나오고 있습니다. 한편 divresult를 살펴보면 1번 shift 시켜주었고 기존에 divresult의 값이 12로 저장되어 있었으므로 divresult = 12 + 2 = 14가 됨을 확인할 수 있습니다.

PC = 9에서도 div $s5, $s5, $s2을 수행되는데 register\_wire\_8에 PC = 8에서 변환된 값인 $s5 = 10이 나오고 있고 register\_wire\_10에서는 여전히 mux를 통과한 $s2 = 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 나눗셈 알고리즘을 통한 결과값인 0이 나오고 있고, mux\_wire\_7에서 나오는 0이 Register(rd = $s5)의 Write data에 새로이 저장되도록 연결되어 있습니다. 나눗셈 알고리즘을 살펴보자면 이제 피제수인 $s5 = 10이 제수인 $s2 = 12보다 작아졌으므로 alu module안에서 data\_s < data\_t가 되고 divresult != 0이므로 remainder만 -1로 바뀌고 divresult는 14로 변함이 없음을 확인할 수 있습니다. remainder가 -1이 됨으로 인해서 result = 32'b0이 되고 alu\_wire\_11로 나오는 값도 0이 됨을 확인할 수 있습니다.

PC = 10에서도 div $s5, $s5, $s2을 수행되는데 register\_wire\_8에 PC = 9에서 변환된 값인 $s5 = 0이 나오고 있고 register\_wire\_10에서는 여전히 mux를 통과한 $s2 = 12가 나오고 있습니다. 이 값들이 alu를 통과한 후 alu\_wire\_11에서 나눗셈 알고리즘을 통한 결과값인 0이 나오고 있고, mux\_wire\_7에서 나오는 0이 Register(rd = $s5)의 Write data에 새로이 저장되도록 연결되어 있습니다. 여기서 알 수 있는 것은 $s5 = 0이 들어가면 항상 다시 $s5 = 0이 나오게 되므로 이제 나눗셈의 과정이 종료되었음을 확인할 수 있습니다. 나눗셈 알고리즘을 살펴보자면 피제수인 $s5 = 0이 제수인 $s2 = 12보다 작으므로 alu module안에서 data\_s < data\_t가 되고 divresult != 0이므로 remainder가 여전히 -1로 유지되고 divresult 또한 14로 변함이 없음을 확인할 수 있습니다. remainder가 -1이 됨으로 인해서 result = 32'b0이 되고 alu\_wire\_11로 나오는 값도 0이 됨을 확인할 수 있습니다.

이제 다음부터 수행되는 PC = 11, 12, ∙∙∙ 에 저장된 div $s5, $s5, $s2은 모두 같은 결과값을 내보내게 될 것이고, 나눗셈의 몫은 divresult에 저장되어 있는 14임을 확인할 수 있습니다. 실제로 178 12 를 해보면 몫이 14가 나옴을 알 수 있습니다.

**5. 프로젝트를 마치며 느낀 점.**

사실 이번 프로젝트를 시작하면서 verilog 언어를 거의 처음 다루어 보았기 때문에 백지 상태에서 시작했고, 주변 친구들도 verilog 언어에 대해 잘 아는 사람이 없어서 코딩을 하는데 상당히 힘이 들었다. 중간에 wave 그래프가 뜨지도 않고 프로그램이 에러가 나서 포기하고 싶은 생각도 들었지만 그래도 끝까지 해보겠다고 덤벼 들은 것이 이렇게 어느 정도의 성과로 나타나게 돼서 너무 기쁘다. 앞으로는 verilog 언어를 사용하여 coding을 함에 있어서 자신감이 생길 것 같고 verilog 언어에 대해 아무것도 모르는 저를 친절히 잘 가르쳐주신 조교님께 너무 감사하다는 말씀을 드리고 싶다.