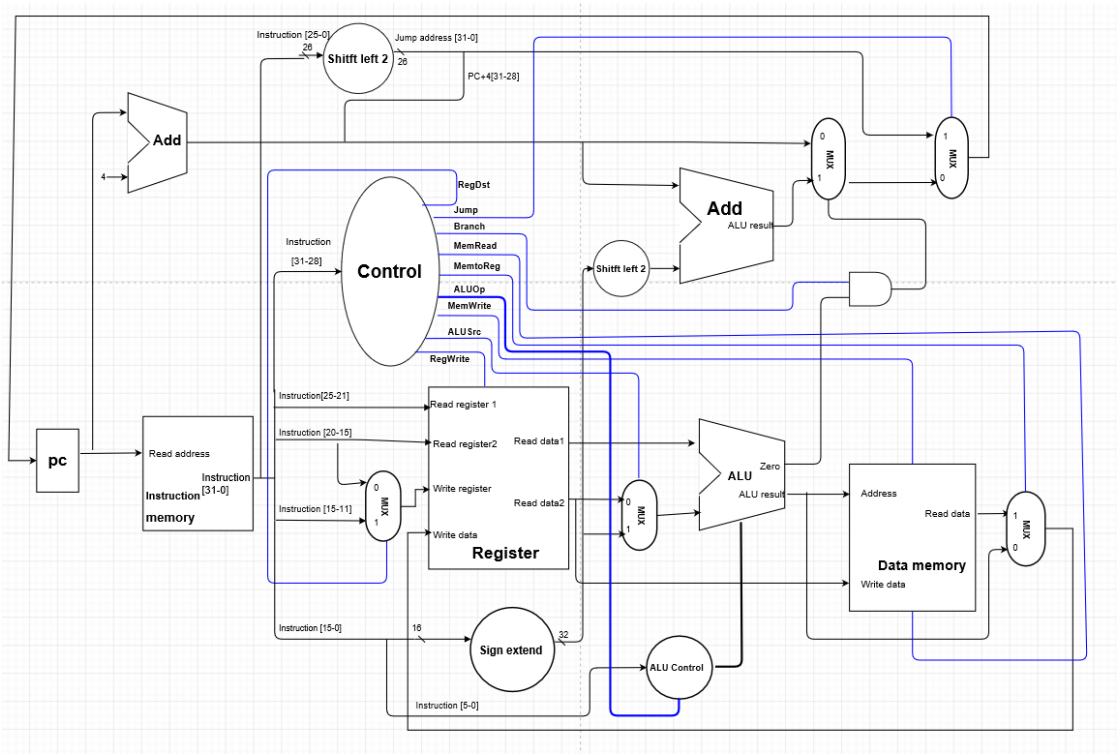
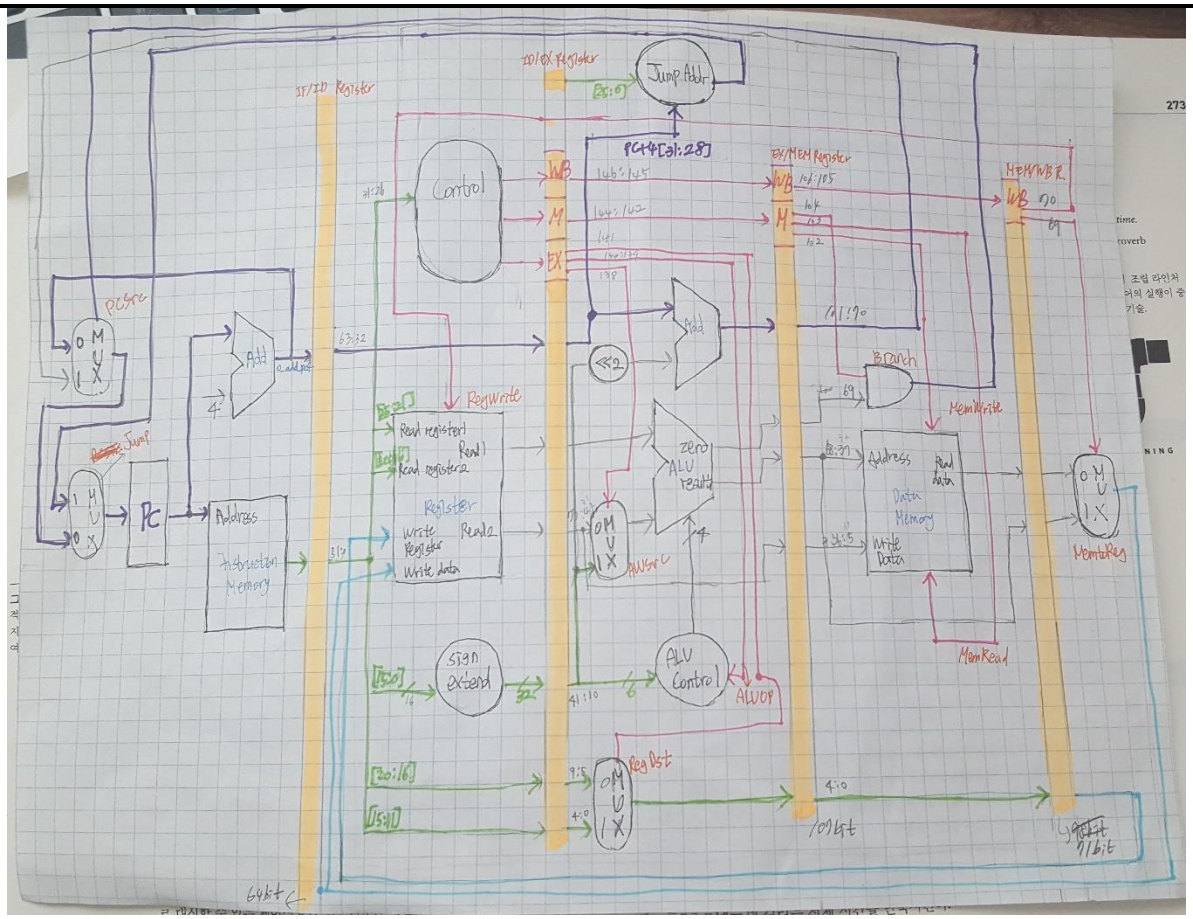
<table>
<tr><td colspan="2">교과목 명</td><td colspan="8">컴퓨터 구조</td></tr>
<tr><td colspan="2">설계 제목</td><td colspan="8">Simple mips</td></tr>
<tr><td colspan="2">설계 기간</td><td colspan="8">2017년도 1학기</td></tr>
<tr><td colspan="2">지도교수</td><td colspan="8">김태환</td></tr>
<tr><td colspan="2" rowspan="3">팀원</td><td>이름</td><td>장윤봉</td><td>학번</td><td>2012122249</td><td>☎</td><td>01097039388</td><td>E-mail</td><td>jjyybb5881@naver.com</td></tr>
<tr><td>이름</td><td>오혜빈</td><td>학번</td><td>2015124129</td><td>☎</td><td>01026995260</td><td>E-mail</td><td>ohb0613@gmail.com</td></tr>
<tr><td>이름</td><td>이다현</td><td>학번</td><td>2015124144</td><td>☎</td><td>01071334561</td><td>E-mail</td><td>eltikei96@gmail.net</td></tr>
<tr><td rowspan="3">목표설정</td><td>설계 배경</td><td colspan="8">Simple Mips 프로젝트는 2017년 1학기 컴퓨터 구조 과목을 수강하면서 배운 내용을 기반으로 CPU CORE를 설계한다. 수업 교재인 Computer Organization and Design의 2장 Instruction:Language of the Computer과 4장 The processor 및 수업내용에 초점을 맞추고 설계를 진행한다. SW와 HW의 Interface인 ISA(Instruction Set Architecture)를 가장 많이 사용하는 MIPS ISA를 예로 들어서 MIPS Instruction의 종류와 format 및 addressing mode 등의 학습한 내용과 Single cycle Architecture를 기반으로 Pipeline Register 및 각종 Unit을 추가하여 5 stage의 pipeline Architecture를 구성하고, Processor안에서 MIPS라는 ISA가 어떻게 구성되고 어떻게 load되고 store되며 CORE에서의 data path와 data path를 제어하는 control path의 control signal 등을 설계한다. MIPS ISA는 Data Memory와 Program Memory가 분리된 Harvard Architecture이며, 설계 간 DM과 IM은 Magnetic disk처럼 용량 당 cost는 낮으면서 capacity는 매우 크고 SRAM처럼 access time이 작은 이상적인 Memory가 있다고 가정하여 설계를 진행한다.</td></tr>
<tr><td>설계 목표</td><td colspan="8">1. KAU Generic Data path Library
Simple MIPS CPU 설계 시 KAU Generic Data path Library에 있는 9개의 module을 기본적으로 사용한다. 이 때 Verilog로 description할 때 reg를 사용하지 않고 wire만을 사용하도록 한다. 또한 Register file module을 사용 할 때 모든 register의 값은 0으로 초기화 되어있다. 단, 29번 Stack Point를 사용하는 SP register는 0X1000의 값으로 초기화한다.

2. Instruction
본 팀이 구현한 CPU CORE module에서 41개의 instruction들이 동작하게 한다. 적용 할 41개의 instruction은 다음과 같다. [ADD, ADDI, ADDIU, ADDU, SLT, SLTI, SLTIU, SLTU, SUB, SUBU, BEQ, BNE, J, JAL, JR, JALR, NOP, LB, LBU, LH, LHU, LW, SB, SH, SW, AND, ANDI, LUI, NOR, OR, ORI, XOR, XORI, MOVN, MOVZ, SLL, SLLV, SRA, SRAV, SRL, SRLV]. Instruction 동작 시에 발생하는 Exception, Floating Point, Cache등 은 무시한다.

3. CPU core Architecture Design
Linux 환경에서 Verilog Code를 사용하여 CPU Core를 설계한다. 이에 두가지 방법이 있는데 첫번째는 Single Cycle Architecture있다. One Cycle에 instruction의 5 stage를 실행하는 architecture이다. 두번째는 Pipeline Architecture이다. 이는 하나의 instruction을 5 stage인 ID(Instruction Fetch), IF(Instruction Fetch), EX(Execute Operation & Calculate Address), MEM(Memory access), WB(Write Back)로 나누어 One Cycle에 One stage를 실행하는 Architecture이다. 우리 조의 목표는 Pipeline Architecture의 설계이므로 각 단계에서 한 Cycle씩 소모되는 것이다. 그러므로 결과적으로 CPI=1이될 것이다.

4. Hazard detection 및 Solute
Pipeline Architecture 설계 시 3가지의 Hazard가 발생한다. Structure Hazard는 2개의 Read data와 1개의 Write data를 사용하며 Memory를 instruction memory와 data memory로 분리하는 Havard Architecture를 사용하여 해결한다. Data Hazard의 경우 기본적으로 Bypassing과 Bubble을 사용하는 Stall을 사용하여 해결한다. 마지막으로 Branch Hazard의 경우는 많은 solution 중에서 Branch delay slot을 사용한다. 이는 Branch instruction이 실행 될 경우 branch instruction에 영향을 받지 않는 instruction을 실행한 후 Branch target address로 이동하는 것이다.</td></tr>
<tr><td>합성/분석</td><td>관련 기술 조사, 분석</td><td colspan="8">1. MIPS ISA(instruction set architecture
이번 term project에서 진행하는 CPU core 구현은 MIPS의 ISA를 기반으로 한다. MIPS는 RISC(Reduced Instruction Set Computer)의 한 종류이며 이에 대조적으로 CISC(Complex Instruction Set Architecture)가 있다. RISC의 특징은 Instruction의 크기가 32bit으로 고정적이며 Clock Cycle 또한 동일하고 addressing mode가 simple 하기 때문에 instruction을 분석하는데 있어 매우 편리하다. 또한 복잡한 회로를 단순한 여러 개의 module로 회로를 구성하기 때문에 cycle time이 짧아 연산의 수가 늘어난다. 이는 CISC보다 CPI는 낮고 IC는 높은데 그렇기 때문에 RISC에서</td></tr>
</table>

		는 Pipeline Architecture을 지향한다. 또한 사용하는 Register의 개수는 32개이며 Memory operand가 허용되지 않아 load 및 store를 사용하여 메모리에서 data 값을 register로 불러온다. 또한 Memory가 aligned 되어 있다. 2. Pipeline Architecture CPU의 Core을 구성하여 instruction을 처리하는데 있어서 두 가지의 Architecture가 존재한다. 먼저 Single Cycle Architecture는 하나의 instruction이 하나의 cycle에서 실행됨으로써 하나의 instruction이 완료되어야 다른 instruction을 수행한다. 이는 능률적으로 매우 떨어지므로 이를 보안하기 위해서 Pipeline Architecture을 사용한다, 이는 여러 명령어가 병렬적으로 실행되는 기술을 일컫는다. 실제 Pipeline Architecture을 사용하는 ARM은 1GHz의 속도에 14단계 pipeline을 갖고있다. Pipeline 설계를 Architecture에 적용하면 시스템의 Throughput이 증가하고 latency가 감소하게 되어 Single cycle Architecture에 비해 stage 수 만큼 처리량이 증가하게 된다. 하지만 pipeline을 설계하기 위해선 다수의 register가 필요하고 이는 power consumption과 관련되어 무수히 많은 pipeline register의 추가는 불가능 하다. 3. Hazard Term project인 CPU core를 Pipeline Architecture로 설계하는데 있어서 instruction들 간의 Hazard 문제가 발생할 수 있다. Structure Hazard, Data Hazard, Control Hazard 등 세가지가 있다. Structure Hazard의 해결은 Havard Architecture을 사용하여 처리 가능하다. 이는 Data Memory와 Program Memory가 분리 되어 있는 형태로 data의 값을 동시에 Memory에서 access하는 경우에 출력 값의 혼동이 없다. Data Hazard는 forwarding 기법으로 해결한다. Forwarding은 값이 available한 단계에서 data의 값이 필요한 단계로 불러와 사용하는 것으로 일명 Bypassing 이라고도 한다. Control Hazard를 해결하는데 여러가지 방법이 있다. 대표적으로 Prediction이 있다. Prediction에도 여러가지 방법이 존재하는데 static branch Prediction(taken, not taken)과 Dynamic branch prediction(History)등이 있다. 또 다른 방법으로는 delayed branch slot으로 이번 project에서 고려할 사항이다.
제작	설계 내용	Single datapath를 먼저 설계한 후 4개의 pipeline Register : IF/ID register, ID/EX register, EX/MEM register, MEM/WB register를 추가하여 5stage pipeline datapath를 설계했다.  다음 블락도를 바탕으로 pipeline datapath를 설계했다. IF-ID-EX-MEM-WB의 5-stage 이다. 여기서 기본 R-format, LW/SW,, branch 이외의 I-format, jump instruction 그리고 forwarding unit을 구현 하기 위해 하드웨어를 추가하여 설계했다.



첫번째 IF stage에서는 instruction이 fatch되는 단게로, pc register, instruction memory가 있다. 여기서 pc register는 KAU generic datapath의 Dregister_we_rst를 사용했다. 여기서 pc register의 입력으로는 브랜치할 경우 target address, jump할 경우의 Jump address, jump,branch를 하지않고 다음 pc 값 즉, pc+4 3개의 값중 하나이다. 이는 PCSrc, Jump 2개의 multiplexer 사용하여 결정한다.

Instruction memory는 SRAM으로 만들어져 주어져 있으므로 설계할 필요가 없었다. Pc 가 저장된 레지스터의 output을 instruction memory의 address로 연결하여 instruction을 읽어올 수 있도록 했다. .Adder를 이용하여 pc+4가 계산되도록 했다. Pc +4 , instruction을 IF/ID pipeline Register에 저장했다.

두번째 ID stage 에서는 IF stage에서 읽어온 instruction을 decode한다. GPR이 저장될 32x32 bit Register file 을 만든다. 이 register file은 kau generic datapath library에 있는 2 Read 1 Write인 RegFile_32x32bit을 이용했다. Instruction의 rs, rt 를 read address로 입력받아 register의 data 값을 출력한다. 이 때, register file이 address 에 따른 data값을 출력하기 위해 clock의 positive edge이여야 한다. 그러므로 register의 data를 출력하기 위해 2 cycle이 소비된다. 이를 막기 위해 read address0, read address1에 IF/ID pipeline register를 거치지 않은 instruction[25:21], instruction[20:16]을 각각 바로 연결해 주었다. Instruciton의 opcode를 입력받아 R-format, I-foramt, branch, lw/sw를 구분하여 control signal을 생성했다. 이는 효율성을 위해 LUT를 이용했다. Control signal은 Jump,RegWrite, MemtoReg, Branch, MemWrite, MemRead, RegDst, ALUOp, ALUSrc이다. 이 controlsignal은 multiplexer의 selection signal이다.

RegDst: instruction rt, rs field를 선택하기 위한 signal

Regwrite: register file에 Write data 입력 값을 쓸지 결정하는 signal

ALUSrc: register file 의 rt register의 data , instruciton 의 immediate field의 sign extension값을 사용할지 선택해주는 signal

MemRead : 1일 때, data memory를 읽어온다.

MemWrite: 1일 때 data memory에 address된 register에 data를 쓴다.

MemtoReg: ALU 출력 또는 data memory의 출력이 write back 될지 결정하는 signal이다.

Branch: 1이면 branch 수행(bne, beq)

Jump: 1이면 jump 수행(j, jr)

또한, immediate field의 16bit을 32bit으로 sign extension했다. Control signal, instruction의 rs, rt field, sign extend immediate field, instruction 을 ID/EX pipeline register에 저장했다.

세번째 EX stage에서는 target adderess 계산, ALU를 이용한 연산, Jump address를 계산했다.
ALU는 2개의 32 bit 입력을 받아 alu control signal에 따라 연산을 수행하고 그 결과값과 zero signal을 출력한다. AND, ADD, OR, SUB, XOR, SLT, NOR, SRA, SLL, MOV 종류의 연산을 할 수 있다.
ALU input 1에는 rs register의 data가 연결된다. SLL, SRA,SRL instruction에서는 sign extension된 sa field가 입력되야 되야 한다. 그러므로 ALU input_1에 mux를 연결하여 입력을 선택할 수 있도록 했다. 이 때, mux의 selection signal은 instruction code가 sll, sra, srl에 해당되는지에 따라 설정했다.
Zero signal은 branch instruction을 위한것으로 ALU의 2개의 input값이 같으면 1이 되고 아닐시에 0 이된다.
ALU input 2에는 R format일 경우 rt register의 data, I-format일 경우 sign extension된 immediate filed data가 입력된다. 또, ALU는 alu control signal에 따라 입력된 2개의 data에 대해 연산이 수행된다. 이 ALU control signal은 4비트로 AND, ADD, OR, SUB, XOR, SLT, NOR, SRA, SLL, MOV 연산을 구분해주는 역할을 한다.

```
assign zero=(i_ALU1-i_ALU2==1'b0)? 1'b1:1'b0;

assign result=(control==4'b0000)? (i_ALU1&i_ALU2):
              (control==4'b0001)? (i_ALU1| i_ALU2):
              (control==4'b0010)? (i_ALU1+i_ALU2):
              (control==4'b0110)? (i_ALU1-i_ALU2):
              (control==4'b1100)? ~(i_ALU1| i_ALU2):
              (control==4'b1101)&(i_ALU1!=i_ALU2)? 32'b1:
              (control==4'b1000)? i_ALU2<<i_ALU1:
              (control==4'b1001)? i_ALU2>>i_ALU1:
              (control==4'b1010)? i_ALU1:

              (control==4'b0111)& (i_ALU1< i_ALU2)? 32'b1:32'b0;
```

이 ALU control signal은 instruction의 opcode, control signal의 ALUOp, instruction의 funct field로 결정된다.
R-foramt: opcode=6'b000000 -> ALUop=2'b10
and, or , nor ,sub, add, xor, Shift, Jump, Move 종류의 연산은 funct field 6bit 로 구분했다.

I-foramat: opcode=6'b001xxx-> ALUop=2'b11, addi, addiu, xori, ori, andi, slti, sltiu의 연산 구분은 opcode의 하위 3비트로 구분하였다.

Lw/sw: alu에서 address 연산을 한다. (add 사용)

Branch: adder를 사용해서 target address를 계산한다.
Target address= (pc+4)+ sign_extend_immediate <<2
Jump: jump address를 EX stage에서 계산하고 pc 값을 바로 바꾸어 준다.
Instruction JR 과 J 두 경우 jump address가 달라진다. 그러므로 multiplexer에 2개의 주소를 입력하 instruction에 따라 select하여 값을 출력하고, 이 값은 Jump multiplexer에 연결해준다. 또한, JR instruction의 경우 R format이기 때문에 control signal중 Jump signal이 0이다. Jump multiplexer의 select signal이 0 이 되어 jump를 수행하지 못하게 된다. 그러므로 jump mux의 select signal을 control의 jump signal과 instruction JR일 경우를 or 연산한 것으로 사용한다.
Jump address= {pc+4[31:28], instruction[25:0], 2'b00} ; J instruction의 경우
Jump address= sign extended rs register data ; JR의 경우
EX/MEM pipeline register에는 RegDst multiplexer의 output, ALU result, zero signal, control signal, target address, ID/EX pipeline register를 거친 rt Register의 data값이 저장된다.

다음은 LUT의 control signal data file이다.

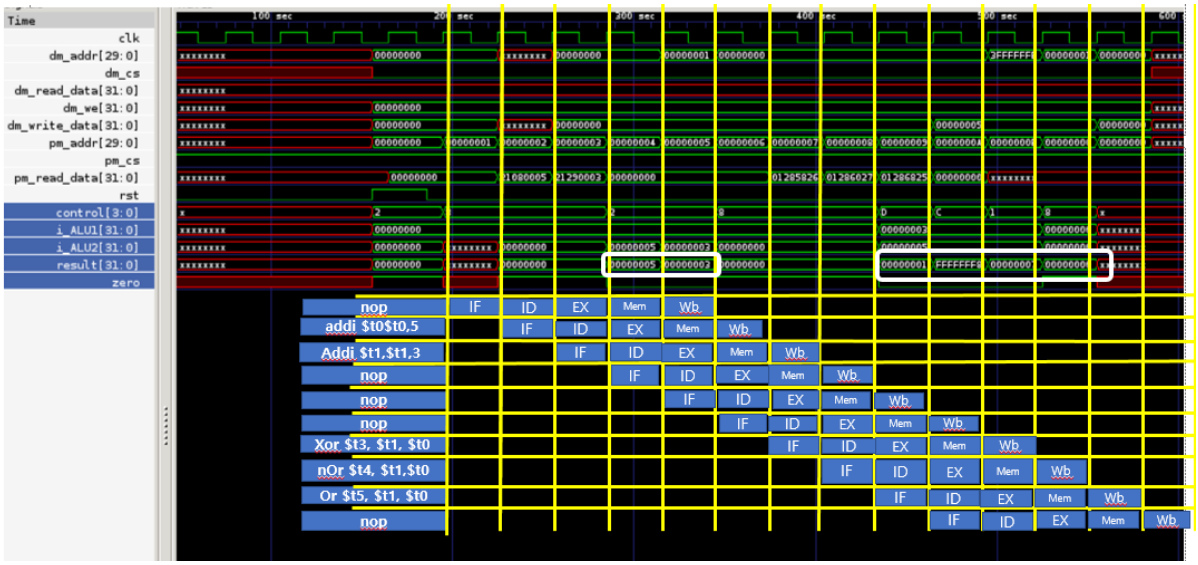
```
 1 0100001100//Rtype00000
 2 0000000000
 3 1000000000//j
 4 0000000000
 5 0001000010//beq000100
 6 0001000010//bne000101
 7 0000000000
 8 0000000000
 9 0100000111//addi
10 0100000111//addiu
11 0100000111//slti
12 0100000111//sltiu
13 0100000111//andi
14 0100000111//ori
15 0100000111//xori
16 0010100010//lui should be changed
17 0000000000
18 0000000000
19 0000000000
20 0000000000
21 0000000000
22 0000000000
23 0000000000
24 0000000000
25 0000000000
26 0000000000
27 0000000000
28 0000000000
29 0000000000
30 0000000000
31 0000000000
32 0000000000
33 0110100001//lb
34 0110100001//lh
35 0000000000
36 0110100001//lw100011
37 0000000000
38 0000000000
39 0000000000
40 0000000000
41 0100010001//sb
42 0100010001//srl
43 0000000000
44 0000010001//sw101011
```

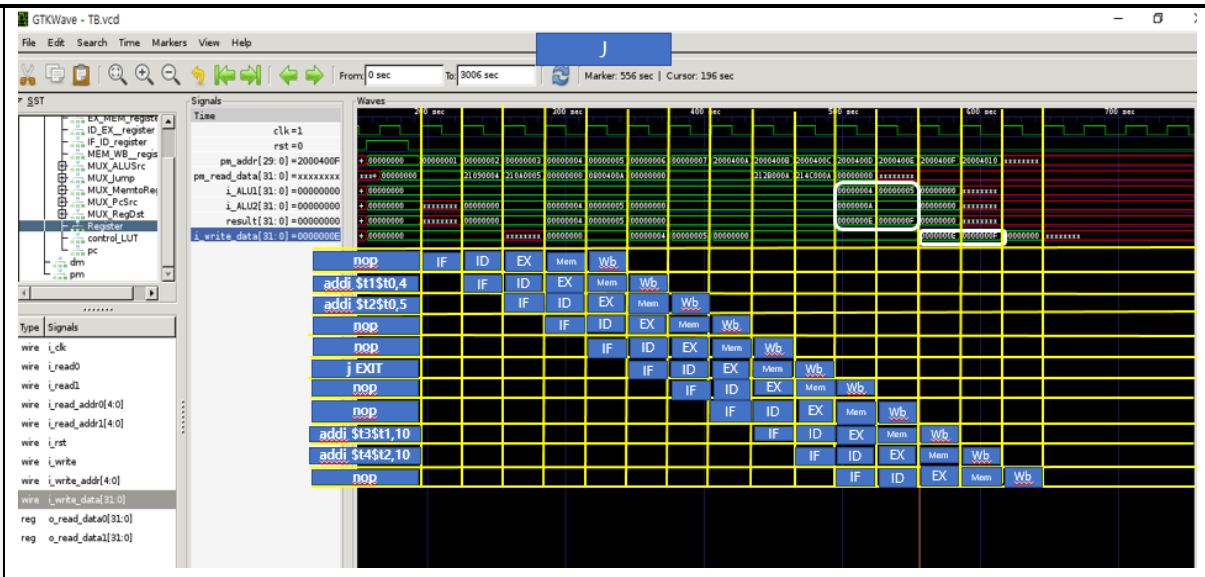
네번째, MEM 단계에서는 PCSrc signal이 결정된다. PCSrc signal은 PCSrc mux의 select signal로 branch 할지를 결정한다. PCSrc signal은 control signal의 branch와 ALU의 zero signal 모두 1일 때 1이 된다.
또, LW instruction에서 data memory에 address를 입력하고 data를 읽는다. SW instruction에서는 datamemory에 address를 입력하여 그 address에 data를 Write한다.
MEM/WB pipline Register에는 Data memory address, address에 따라 Read한 data, control signal이 저장된다.

다섯번째, WB stage에서는 data memory에서 Read한 data 또는 ALU에서 계산된 값이 register file에 Write 된다.이 때 data를 write할 register는 EX stage에서 RegDst multiplexer로 instruction의 rt 또는 rs field의 register인지 결정한다.

기본적인 5 stage pipeline datapath를 설계하면 hazard가 발생하게 된다. 크게 structure hazard, data hazard, control(branch) hazard로 나눌수 있다.
Instruction memory와 datamemory를 따로 설계했으므로 structure hazard는 발생하지 않는다.
Data hazard의 경우 EX_hazard, MEM_hazard, WB_hazard가 발생한다. EX_hazard는 현재의 instruction에서 필요한 rs, rt data값이 바로 전의 instruction에 의해 계산되고 있을 경우 발생한다. 이 때, Forward_A, Forward_B signal을 추가하여 2'b10 으로 설정했다. Forward_A, Forward_B가 2'10일 경우 EX 단계의 ALU결과 값을 ALU의 input으로 forwarding 시켜주었다.
MEM hazard는 현재의 instruction이 사용하고 있는 rs, rt의 data 값이 2 cycle 전의 instruction에 의해 mem stage에서 변경되고 있을 경우 발생한다. 이 때 Forward_A, Forward_B signal을 2'b01로 설정한다.
Forward_A, Forward_B signal이 2'b01일 경우 Write data를 ALU의 input으로 forwarding 해준다. 교과서에는 WB stage에서의 data hazard는 발생하지 않는 다고 되어있는데, 실제 설계해본결과 hazard 가 발생했다. 이로 인해 결과 값들이 달라지는 것을 막기 위해 mips code에 nop instruction을 사용했다.
위에서 말한 forwarding을 실행하기 위해서 EX stage의 ALU의 input_1 전에 Forward_A multiplexer, ALUSrc multiplexer 전에 Forward_B multiplexer를 추가하여 연결했다. Forwarding unit에서 forwarding이 detect된다면 Forward_A, Forward_B signal이 설정된다. 이 signal에 따라 Forward_A MUX, Forward_B MUX의 값은 rs, rt register

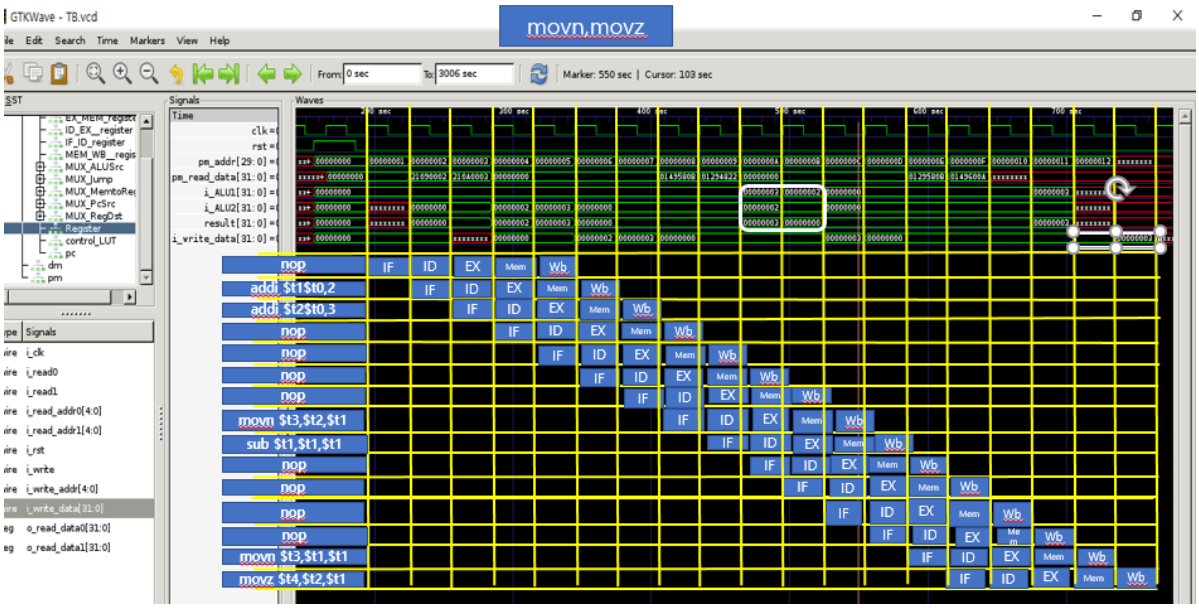
		data, ALU result, Write data의 3개 중 하나의 값이 선택 된다. Lw instruction 다음의 instruction이 lw instruction에서 사용하는 register를 read하려고 할 때 load use data hazard가 발생하게 된다. 이는, ID/EX pipeline의 control_MemRead signal이 1일 때, EX stage의 rt register와 ID stage의 rs 또는 rt register가 같다면 발생한다. 이 때, pipe line을 stall 시켜 주어야 한다. Stall을 시키기 위해서는 pc register와 IF/ID register의 값은 유지되도록 하고, 나머지 뒷 부분의 pipeline은 nop이 실행되게 해야한다. control signal을 0으로 주면 되는데, 그러기 위해서 ID stage의 control signal 다음에 multiplexer를 연결하여 load use data hazard가 detect 될 때에 ID/EX pipeline register에 control signal이 0 이 저장되도록 했다. Control hazard의 경우 branch가 EX stage에서 target address가 계산되고 mem stage에서 branch가 결정되기 때문에 branch 다음에 instruction 3개가 fetch되어 발생한다. Compiler가 branch전의 연관 없는 instruction을 branch 다음에 실행하는 delayed branch 기법이 사용된다. mips에서는 delayed branch slot이 1개이므로 branch를 ID stage에서 결정하고 수행해야 control hazard를 완전히 없엘 수 있다. 하지만 branch를 ID stage에서 수행하게 된다면 이에 따른 새로운 hazard가 발생한다. 이 hazard를 처리, 해결하기위해 hardware 설계가 복잡해져서 branch 하는 instruction의 경우 그 다음에 NOP instruction을 넣어 control hazard가 발생하는 것을 방지했다. MOVZ, MOVN instruction의 경우 R format이여서 control signal 중 RegWrite 이 1로 인가된다. 하지만, rt register의 data값에 따라 RegWrite signal은 0 이 되어 rd register의 data가 유지되어야 한다. 그러므로 EX/MEM pipeline register에 저장되는 control_RegWrite signal에 mux를 연결하여 MOVZ,MOVN instruction일 경우에 rt register의 값에 따라 0 또는 1 이 되도록 했다. Rd register 값이 바뀐다면 rs register의 값이므로 ALU의 출력은 항상 RS register의 data가 되도록 했다. Add 와 addu, slt 와 sltu 등 unsigned는 구분하지 않고 똑같이 동작하도록 했다. 또한, overflow가 발생하는 경우는 고려하지 않았다. 프로젝트에 있어 주어진 설계 규격(input, output)을 지켰다. 또한 hardware를 리눅스 서버에서 verilog hdl을 사용하여 설계 했고 iveri를 이용해 synthesize 했다. 이 때, reg, if, always를 사용하지 않고 assign과 wire만 사용하여 설계했다. 또한, test bench는 gtkwave를 이용해 분석하였다.
	설계 과정	본 팀은 CPU core를 설계하는데 있어서 제일 먼저 교과서를 수단으로 Architecture에 대한 이해와 수업시간 동안 배운 이론들을 갖고 설계에 어떻게 설계를 할지에 대하여 학습 하였다. 그리고 이를 토대로 팀원 모두 모여 설계 계획서를 작성하며 앞으로 진행 될 term project에 대한 계획을 논의하였다. 이에 따른 팀원들 간의 역할분담을 정하고 설계를 진행하였다. 제일 먼저 진행한 것은 Single Cycle Architecture 설계였다. 이는 CPU의 기본 Architecture로 Pipeline Architecture의 기반이 될 것이기 때문이었다. Single cycle Architecture는 교과서에 나와있는 Design을 바탕으로 제작 하였으며, Program memory와 Data memory는 SRAM으로 제작되어 Core와 분리 되어 존재함을 인지하였다. 그 다음 Program Counter에 대하여 clock이 active edge 일 때마다 PC+4의 값이 Update 되게끔 KAU library에 있는 D register을 사용하여 연결시켰다. Update된 PC에 해당하는 Instruction이 program data에 들어와 출력된 값으로 ALU, Register file, data memory, port들을 control하기 위하여 control signal을 ALU op를 이용한 LUT를 만들어 통제하였다. 이는 pipeline Architecture로 구현하였을 때 더 복잡한 control을 위하여 opcode를 고려하여 통제하였다. Pipeline architecture 설계를 위하여 4개의 pipeline register을 삽입하여 하나의 instruction을 5단계로 나누어 실행하게끔 하였다. 왜냐하면 pipeline register을 통해 signal을 전달하지 않으면 다음 stage에서 필요로 하는 data의 값이 clock과 무관하게 update되어 필요한 값을 사용 할 수 없기 때문이다. 그래서 Pipeline register는 active edge clock일 때 값을 전달해주는 D register을 사용하여 구현 하였다. Pipeline이 추가 된 후에는 architecture에서 R format instruction들의 동작을 확인하고 정상적인 값들을 출력하고자 하였다. 하지만 복잡한 명령어들을 연속으로 작성하여 구동하면 data hazard가 발생하기 때문에 이를 해결해야만 했으므로 간단한 명령어들을 입력하고 후에 hazard를 처리하기로 하였다.

		R format instruction의 구현 후에는 I format instruction들을 구동 시켰다. R-type 명령어를 구현한 후 Load와 Store 같은 Memory Reference 명령어들이 동작할 수 있도록 Program Memory와 Data Memory의 출력 값을 시뮬레이션을 통해서 계속 확인하면서 Data-path를 설계하고 Control Signal이 정상적으로 MEM stage까지 도달하는지를 검증하는 과정을 거쳤다. 초기 Control Unit을 제작할 때 Look Up Table을 작성하였는데, 이때는 Load Word, Store Word만을 고려하여 Control Unit을 제작하였는데, 이후에 Load Word 뿐 아니라 Load Byte와 Load Half Word를 동작하는데 있어 Control Signal을 이용하면 편리하게 제어할 수 있을 것이라는 판단에 따라 Look Up Table을 수정하였다. Control instruction 구현에선 control hazard를 고려하면서 기술하였다. Hazard solution으로 delayed branch slot을 사용하였으며 구현하면서 발생한 오류들은 control signal과 여러가지 code들의 수정으로 바로잡았다. 마지막으로 Hazard Detection Unit과 forwarding Unit을 구현하여 모든 Hazard를 해결하고자 하였다. Forwarding Unit은 책에 나와있는 것을 바탕으로 추가적인 사항들을 고려하여 기술하였다. Assign과 Mux를 통하여 Control signal을 이용하여 Stall을 추가하였다. Hazard detection 또한 책을 참고하여 기술 하였다.
시험/평가	설계 결과 검증, 분석, 평가	이번 Term Project에서 설계한 CPU CORE가 정상적으로 설계되었는지 검증하기 위해서 Test Bench에 Test Vector 넣음에 따라 나오는 data의 출력 값들을 gtkwave의 wave form으로 확인하는 방식을 사용하였다. Gtkwave를 작동시켜 data의 값들을 볼 때 각 5 stage가 차례로 delay없이 출력되고 있는 것과 ALU의 연산에서 정확한 연산이 이루어졌는 것에 중점을 두고 확인하였다. 우리는 주차별로 나누어 완성 시킨 instruction들이 다르기 때문에 주차별로 Test Vector는 본 팀이 확인하고자 하는 instruction들의 구동을 위주로 작성하였다. * andi, xor, nor, or instruction  $t0와 $t1 register에 있는 data 값인 5와 3의 연산으로 xor,or,nor instruction의 ex 단계에서 연산 값들을 볼 수 있다. 또한 i_ALU1과 i_ALU2 port에서 instruction에서 필요한 data 값들이 연결되어 있는 것을 볼 수 있다. 또 그림을 첨부하여 하나의 instruction의 5 stage가 정확한 clock cycle에 동작되며 ALU에서 값이 나오는 순간은 각 instruction의 EX 단계임을 알 수 있다. * j instruction



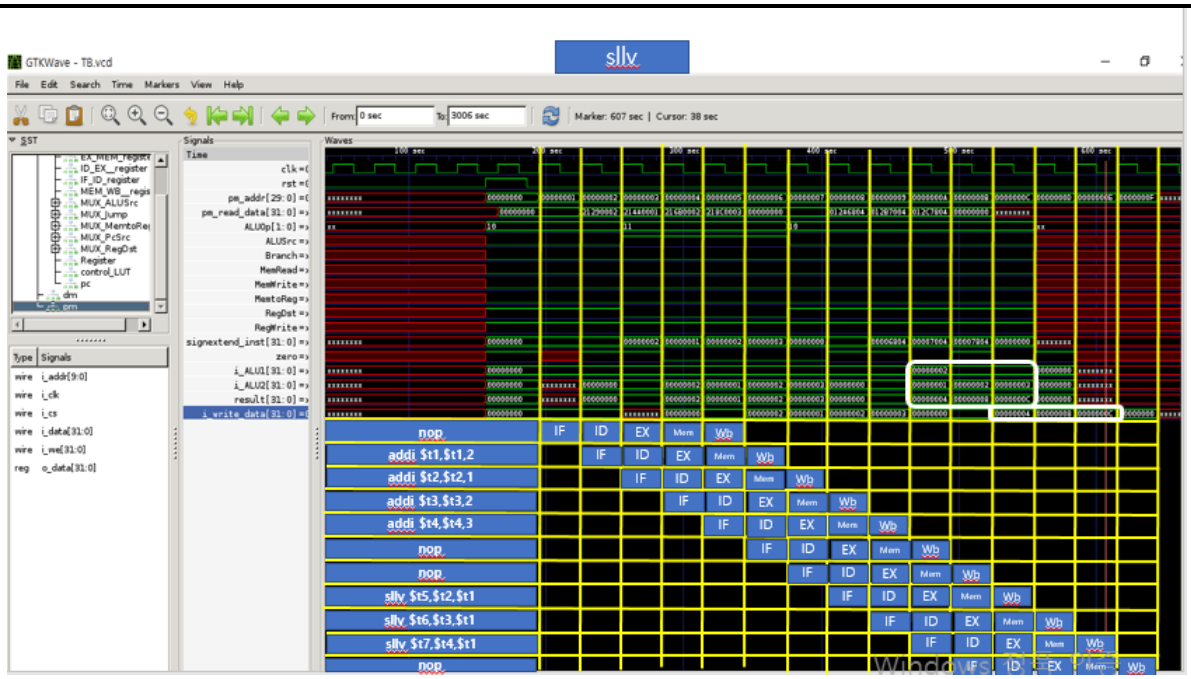
$t1과 $t2에 각각 4와 5값을 넣어준 후 jump를 하여 EXIT 레이블로 branch 하기 때문에 addi $t3,$t1,5와 addi $t4,$t2,5가 수행되지 않고 아래의 addi $t3,$t1,10과 addi $t4,$t2,10 만 수행되어 $t3와 $t4에는 5가 더해지는 것이 아니라 각각 10이 더해 질 것이며 $t3에는 14, $t4에는 15가 저장 될 것이다. 검증은 ALU_result값이 5를 더한것인지 10을 더한것인지 확인하였고 각각의 WB stage에서 update 되는 값을 확인하여 j 명령어 작동을 검증하였다.

* movn, monz instruction



$t1의 값이 변함에 따라 move 가 결정되는 code 이다. 처음 movn은 $t1값이 2이기 때문에 (not zero) $t3의 값이 $t2(=3)으로 update될 것이며. Sub를 거쳐 $t1=0이 되고 movn 에서 $t1은 zero이기 때문에 movn에 의해 move 하지 않을 것이다. 마지막으로 movz에 의해 $t4에 $t2(=3)이 update 될 것이다. 처음 movn에서$t1이므로(not zero) $t3의 값이 $t2(=3)로 저장된다. .sub를 통해 $t1이 0으로 update되고 마지막 movn에서 $t1(=0)이므로 movn은 무시하고 movz에서는 $t1(=0)이므로 $t4에 $t2(=3)이 update되었다. 각각의 EX단계와 wb 단계에서 WB 됨을 확인하였다.(하얀 BOX)

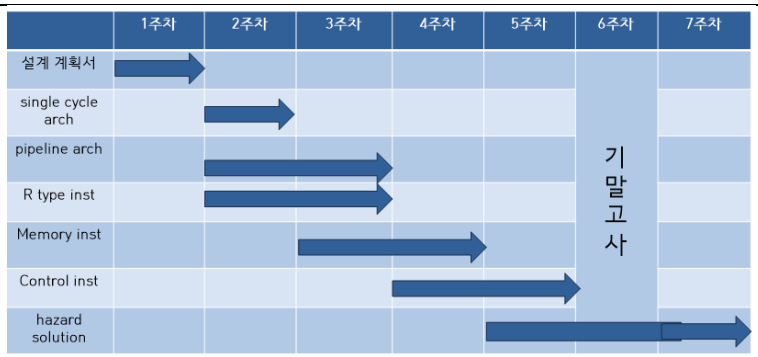
* SLV instruction



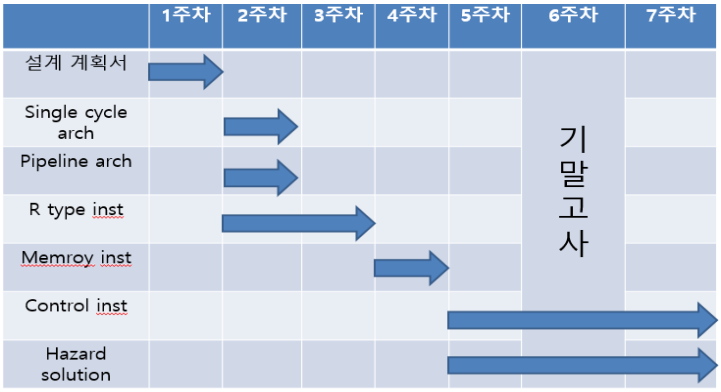
$t2(=1), $t3(=2), $t4(=3)를 $t1(=2) bit 만큼 shift left 시키기 때문에 각 resgiter 값에 x4를 한 결과가 나올 것이다. 따라서 $t5에는 1x4=4, $t6에는 2x4=8, $t7에는 3x4=12이 저장 될 것이다. 이는 $t5, $t6, $t7의 WB stage에서 update되는 값(register write 값)을 확인하여 검증한다. Gtkwave로 $t5에는 1x4=4, $t6에는 2x4=8, $t7에는 3x4=12이 각각 EX 단계의 ALU에서 계산 된 걸 확인하였고(0x00000004=32'd4, 0x00000008=32'd8, 0x0000000C=32'd12), 각각 WB 단계에서 $t5, $t6, $t7에 shift left 결과 값이 update된 걸 확인 하였다. (하얀색 BOX)

위의 test vector 외에도 41개의 instruction을 검증하기위한 test vector을 구상하였으며 이를 종합적으로 사용하여 복잡한 연산을 수행하는 test vector을 만들 수 도 있었다. 복잡한 test vector을 사용함으로써 전체적인 instruction의 동작을 한번에 볼 수 있으며 hazard 발생시 처리가 된 부분을 볼 수 있었다.

추진 일정 대비 진행 결과



<추진 일정>



<진행 결과>

1주차에는 설계 계획서를 작성하여 전체적인 설계 계획을 작성하고 역할 분담 및 구상을 진행하였다. 설계 계획서에 따라 2주차에는 하나의 instruction이 one cycle에 진행되는 single cycle architecture을 설계하였다. Program counter와 register file, program memory, ALU, data memory, 각 필요한 MUX들을 사용하여 구현하였다.

2주차와 3주차에 걸쳐서 single cycle architecture을 기반으로 pipeline architecture을 만들고 이 arch 위에서 R format instruction들이 동작하게끔 계획 하였으며 계획대로 3주차에 두 개의 계획 모두 완성 시킬 수 있었다. Pipeline architecture가 single cycle architecture위에 4개의 pipeline register만 추가하면 되었기 때문에 2주차에서 구현하는 것에 큰 어려움은 없었으나 R format instruction을 구동함에 있어서 문제가 있어 3주차에 완성 할 수 있었다. 2주차부터 R format instruction을 다루며 오류가 있는 부분은 수정하고 추가하여 해결했다.

3주차와 4주차에 걸쳐 Memory type instruction(I format instruction)을 구현하기로 계획하였으나 3주차에서 R type instruction에서 감을 잡느라 오래 걸려 3주차가 아닌 4주차부터 실행하였다. 기존의 R format instruction이 동작하던 pipeline architecture에서 I format instruction에 대한 Control signal 추가와 ALU 기능 향상 등을 통하여 계획에 맞춰 완성 시킬 수 있었다. 구현 하던 도중 궁금증이 생겨 교수님과의 면담을 통해서 해답을 얻었으며 이는 설계에 많은 도움이 되었다.

4주차부터 5주차 까지 Control instruction(ex bne, beq, jump 등)의 구현을 목표로 하였다. 하지만 실제로 4주차의 Memory instruction의 구현 때문에 5주차에 실시하게 되었고 6주차에 있는 기말고사 기간에는 팀원들의 시험을 위하여 설계 도중 잠시 중단 하였다. 그리고 시험이 끝난 후 7주차에 다시 진행하였다. 7주차에 Control instruction과 마찬가지로 계획했다 시험기간 때문에 잠시 중단했던 Hazard solute에 대한 설계를 진행하였다. 이는 6월 25일 최종 시연 전 날까지 진행하였다.

결과적으로 총 2번의 교수님과의 면담을 통하여 설계 도중 해결하지 못한 부분을 해결 하였고 pipeline architecture을 갖는 CPU를 만들어 41개의 MIPS instruction에 대한 구동을 가능하게 하였다. 그리고 이 때 발생하는 여러가지 Hazard들을 해결 하여 목표에 도달 하였다.

역할 분담 및 개인별 공헌 내용

팀원 모두 : 토의를 통하여 전반적인 설계 계획을 구상하고 주 차별 계획을 세웠다. Pipeline architecture의 구현이 목표지만 기본 형태인 single cycle architecture 구현을 먼저 하여 이에 필요한 구상 및 논의를 하였다. PC를 D register로 구현하여 active edge 일 때 PC+4가 되도록 하였고 교수님이 주신 core.v와 tv.v의 code를 보며 이를 분석하여 CPU 구현에 필요한 port들을 생각하였다. 또한 41개의 instruction들을 분석하여 format별로 나누어 구현하기로 하였다.

장윤봉 : 구현한 CPU Architecture Code에서 잘 못된 점을 gtkwave를 실행시켜 분석하여 찾아 이에 따른 해결책을 제시하였다. 또한 instruction들의 opcode를 이용하여 Control signal에 필요한 look up table을 만들어 Core module에 추가하였다. 검증을 위한 사진들을 분석하여 보고서 작성 시 알아보기 쉽게 그림을 그려 첨부하였으며 보고서 최종 검토를 맡으며 보고서의 품질향상을 도모하였다.

오혜빈 : 팀원 모두가 CPU를 구현하면 시간 소비와 팀원들이 각자 구현한 module 간의 data path 혼동이 일어날 것 같아 CPU Core의 Verilog code 기술을 맡았다. CPU core의 Sigle Cycle을 구현 하였고 이를 토대로 Pipeline Architecture를 구현하였다. Gtkwave에서 찾은 문제점의 해결 방안을 제시하며 이에 따른 Verilog code 부분에서 수정하여 제대로 동작 되게끔 하였다. 결과 보고서에서 검증 부분을 맡아 기술하였다.

이다현 : 구현한 CPU Architecture Code에서 잘 못된 점을 gtkwave를 실행시켜 찾았으며 이에 따른 해결책을 제시하였다. 주 차별 계획을 구체적으로 수립하고 세분화 하였다. 또한 매 주차 결과 보고서를 작성하였다. 41개의 instruction이 구현한 Architecture에서 구동이 되는 것을 보이기 위한 Test Vector을 만들었다.

현실적 제한 요건 반영 결과

경제요건	팀원 3명의 거주지 지역이 달라 학교에 등교하는 요일 외에는 만나 설계를 진행하는데 한계가 있었다. 또한 각자의 수업 시간표의 구성이 달라 공강 시간 영역도 달랐다. 그래서 본 조는 만나는 날에 토의를 통해 설계할 분량을 정하고 역할 분담을 하여 각자 집에서 구상 및 역할의 임무를 다하였다. 서로 추가적으로 논의할 점과 궁금증 들은 단체 대화방에서 진행하였다.
사회적 영향	Linux 환경에서 CPU core code를 작성함에 있어서 한 명의 팀원이 배려하여 작성한 code를 다른 팀원들이 알기 쉽게 module과 port를 명명하였다. 또한 주석을 달아 각각의 code들이 Pipeline architecture에서 어떠한 단계이며 어떤 module인지 알기 쉽게 명명하였다. 또한 보고서

		를 작성함에 있어서 검증 wave에 추가적인 그림들을 넣어 시각적으로 instruction이 어떠한 단계에서 어떤 cycle에 실행되는지 알기 쉽게 하였다.
	미적 요소	Project를 진행함에 있어서 각자의 개인 공부와 과제 및 시험이 있기 때문에 무리하게 밤을 새거나 스트레스 받아 팀원들 각자의 건강을 헤치지 않기 위해서 역할 분담을 하였고 실현가능한 범위 내의 주차 별 계획을 세워 project를 진행하였다.
	보건 및 안전	이번 프로젝트의 개발 환경은 교수님께서 제작하신 리눅스 서버에서 이루어졌다. 이것은 우리 팀뿐만 아니라 다른 팀 역시 이 서버에서 작업을 한다는 것이다. 서버의 크기가 한정되어 있기 때문에 많은 인원이 접속하여 작업을 하게 되면 서버의 성능이 저하될 수 밖에 없다. 따라서 타 팀에서 접속이 많지 않은 시간대를 선정하여 작업을 진행하였다. 작업 시간은 오전 시간대와 새벽 시간대에 서버의 성능이 가장 원활했고 Project의 진행 중 이용자의 수가 많아 module 인식에 문제가 가끔 발생하긴 했지만 원활하게 프로젝트를 마무리 할 수 있었다.
	내구성	Instruction Set Architecture의 전형적인 Instruction Set으로서 RISC의 가장 대표적인 Instruction Set인 MIPS를 설계하였다. MIPS를 실행시킬 CPU core는 Havard Architecture이면서 Pipeline Architecture인 형태로 설계하였다. 또한 Windows 체제에서 Linux Server로의 접속에 사용되는 응용 프로그램들에 대해서 표준을 살펴보고 이것이 우리 프로젝트에 어떠한 영향을 미쳤는지 살펴보고자 한다. PUTTY는 MIT Simon Tatham에 의해 제작되었으며, SSH, 텔넷, Rlogin, Raw TCP를 위한 클라이언트로 동작하는 자유 및 오픈 소스 단말 에뮬레이터 응용 프로그램이다. 여기서 우는 SSH를 사용하게 되는데, 이것은 Secure Shell의 약어로 공개 키 방식의 암호 방식을 원격지 시스템에 접근하여 암호화된 메시지를 전송할 수 있는 시스템이다.[1]
	산업 표준	Instruction Set Architecture의 전형적인 Instruction Set으로서 RISC의 가장 대표적인 Instruction Set인 MIPS를 설계하였다. MIPS를 실행시킬 CPU core는 Havard Architecture이면서 Pipeline Architecture인 형태로 설계하였다. 또한 Windows 체제에서 Linux Server로의 접속에 사용되는 응용 프로그램들에 대해서 표준을 살펴보고 이것이 우리 프로젝트에 어떠한 영향을 미쳤는지 살펴보고자 한다. PUTTY는 MIT Simon Tatham에 의해 제작되었으며, SSH, 텔넷, Rlogin, Raw TCP를 위한 클라이언트로 동작하는 자유 및 오픈 소스 단말 에뮬레이터 응용 프로그램이다. 여기서 우는 SSH를 사용하게 되는데, 이것은 Secure Shell의 약어로 공개 키 방식의 암호 방식을 원격지 시스템에 접근하여 암호화된 메시지를 전송할 수 있는 시스템이다.[2]
토의	* System architecture 본 team은 이번에 진행된 Simple MIPS CPU Core Architecture을 <computer organization and design-저자 패터슨& 해네시>를 바탕으로 구현하였다. 교재에서 언급하는 Architecture는 Pipeline Arch로 ID단계에서 Program Counter에서 출력된 instruction의 값이 IF/ID pipeline register을 통과하여 Program memory로 입력되어 이에 해당하는 명령어 및 값들이 출력되어 Register File 과 Control Signal로 전달 된다. 이 부분을 구현 할 때 register file에서 one cycle이 delay 되어 EX 단계의 ALU 연산에서 문제가 발생하였다. 이는 Register file module이 clock의 영향을 받는 register로 설계되어 있기 때문에 발생한 문제였다. 이를 해결하기 위해서 본 팀은 IF 단계에서 fetch 된 instruction의 값을 바로 program data에 넣음으로써 해결하였다. * Control Unit Control Unit은 pipeline architecture을 구현함에 있어서 Pipelining이 정상작동 하기 위해 중요한 역할을 하며, 이것은 Program Memory에서 나오는 명령어를 입력으로 사용한다. 교재에 따르면 Control Unit은 명령어의 MSB 6bit인 Op-Code를 이용하여 출력 Signal이 결정된다. 하지만 교재에 있는 Control Unit의 경우 Add, Sub, And, Lw, Sw 등 일부 명령어만이 작동되도록 설계된 것이고, 우리가 40여개의 명령어를 지원하기 위해서는 더욱 정교한 Control Signal 생성이 필요했다. 따라서 우리는 2bit의 ALU op뿐 아니라 또 다른 명령어의 값인 OP-code를 추출하여 이것을 복합적으로 사용하기로 결정했다. 이를 위해서 MIPS ISA manual을 참고하였고, Instruction Format의 LSB 6bit인	

[1] 네이버 지식백과, 컴퓨터인터넷IT용어대사전, 2011. 1. 20., 일진사

[2] 네이버 지식백과, 컴퓨터인터넷IT용어대사전, 2011. 1. 20., 일진사

Function Field를 Op-Code와 함께 Look-Up-Table에 추가하기로 했다. 매주 설계를 진행하면서 Look Up Table로 정의 했던 초기의 Control Signal들이 오류가 있음을 발견하였다. 이는 41개의 instruction들과 Hazard를 검증하면서 발견 된 오류들을 수정하여 완성하였다.

* ALU

Pipeline Architecture에서 EX 단계에 있는 Address 및 data 값들의 연산을 하는 ALU module에 shift left와 shift right, XOR 등 기본적인 사칙 연산 외의 연산을 수행 할 수 있도록 연산 기능을 추가 하였다. 이는 41개의 instruction 구현 중에 사칙연산 외의 연산 기능이 필요한 것을 알았다. 이는 교수님과의 면담을 통하여 ALU module coding을 기술적으로 기술하여 구현하는 해답을 얻었다.

* 고찰

2017년 1학기 컴퓨터구조를 수강하면서 컴퓨터의 구조에 대해서 배우고 RISC의 대표적인 MIPS를 배웠다. MIPS ISA를 바탕으로 CPU Core가 어떻게 구성되고 작동하는지, 특히 실제 CPU들은 Pipeline Architecture를 탑재하고 있고 우리가 사용하는 상용 컴퓨터는 더 복잡할 수 있다는 것을 배웠다. 우리가 비록 Software를 전공하는 것은 아니지만 Hardware를 제어하기 위해서는 그 기저에 있는 CPU와 Memory가 중요하다는 것을 느낄 수 있었다. 또한 컴퓨터구조라는 과목의 잦은 Quiz 덕에 책에 나오는 각각의 선, Data-Path, 각 Block이 어떠한 역할을 하는지, 어떤 의미가 있는지, 이것이 없다면 시스템이 어떤 오류를 유발할지 등 깊이 있게 학습 할 수 있어서 실제 설계에서 많은 도움이 되었다. 그래서 여러 가지 이유가 있더라도 책에 있는 Text와 그림, 도표 등을 무비판적으로 받아들이지 않고, 무의미하게 단순히 암기하는 행위를 지양해야겠다는 것을 느꼈다. 프로젝트를 진행하면서 gtkwave를 보며 해당 Verilog code의 문제점을 찾아내는 능력을 기를 수 있었고 Architecture을 기술하기위한 Verilog coding의 능력을 향상시켰다.