교과목 명	컴퓨터 구조							
설계 제목	Simple Mips							
설계 기간	2017년도 1학기							
지도교수	김태환							
팀원	**이름**	장윤봉	**학번**	2012122249	☎	01097039388	**E-mail**	jjyybb5881@naver.com
	이름	오혜빈	학번	2015124129	☎	01026995260	E-mail	5binnn@naver.com
	이름	이다현	학번	2015124144	☎	01071334561	E-mail	eltikei96@gmail.net

목표설정

설계 목표

1. 이번 설계는 2017년도 1학기 컴퓨터 구조 과목의 term project 로서 간단한 RISC(Reduced Instruction Set Computer)을 설계하고 실행시키는 것이다. RISC 중에서도 MIPS32 Instruction Architecture를 이용하여 CPU의 integer processing core을 실현화 하는 것이 최종 목표이다.

2. 구현을 하고자 하는 명령어들은 41개로 [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] 이다. 구현할 시에 exception, floating point unit, cache, MMU등은 고려하지 않는다.

3. CPU core 설계를 5-stage-pipeline으로서 구현할 것이다. CPU core 설계 방법에는 2가지 설계 방식이 있는데, 하나는 Single Cycle Architecture와 Pipeline Architecture이다. Single Cycle Architecture는 하나의 Instruction이 One Cycle에 동작하는 것이며, Pipeline Architecture는 하나의 Instruction을 5 Stage로 나누어 동작하는 방법이다. 5 Stage에는 Instruction Fetch(IF), Instruction Decode(ID), Execute Operation & Calculate Address(EX), Memory Access(MEM), Write Back(WB) 등이 존재하며 5 Stage에서 Instruction이 동작하는 것을 목표로 한다.

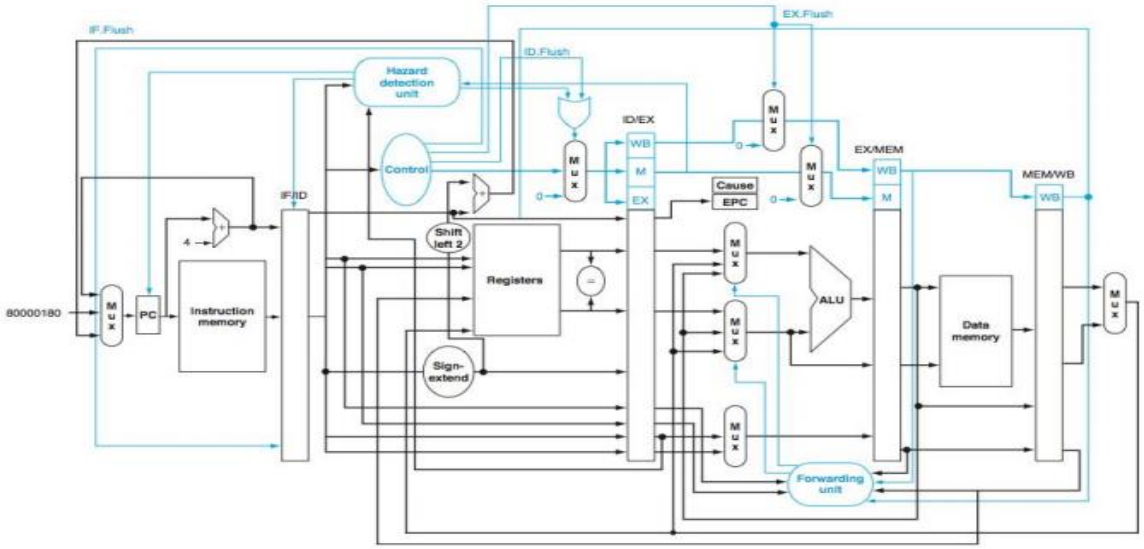
4. KAU Generic Datapath Library 에 있는 Two-read one-write Register file을 사용한다. Register initialization을 하는 것을 목표로 한다. RegFile 중 0번 Register는 항상 Logical Value 0의 값을 유지하며 29번 Stack Point는 0X1000으로 초기화 한다. Special Purpose Register를 사용하는 PC의 초기값은 0이다. 그 외 나머지 Register의 값은 0으로 초기화한다.

5. 이번 Term Project는 Linux 환경에서 Verilog HDL을 이용하여 Hardware를 Description하며, GTK Waveform을 이용하여 Test Bench code를 확인하여 설계를 검증하게 된다. 이때 HDL Description은 완전히 synthesizable 해야 하며 오직 wire와 assign만 Hardware Description에 사용해야한다. Description시 'KAU Generic Data path Library'에 9개의 소자를 기본으로 작성한다.

6. 5-stage-pipeline에서 Instruction을 처리 할 때 발생되는 3가지의 Hazard를 해결하는 것이 목표이다. Hazard에는 Structure Hazard, Data Hazard, Control Hazard 등이 있다. Structure Hazard는 Two-read one-write register을 사용하여 Instruction수행하는 것과 instruction Memory와 Data Memory를 분리한 Harvard Structure를 사용하여 해결 할 것이다. Data Hazard는 Bypassing을 사용하여 해결하며 Load-use data hazard같은 경우에는 Bypassing과 Stall을 동시 적용함으로써 해결할 것이다. Control Hazard의 경우 stall on branch 방법을 이용하여 branch evaluation이 될 때까지 bubble을 삽입한 후 stall하여 control hazard를 해결하는 것이 일차적 목표이다. 이후 구현 성공 시 branch에 의한 hazard penalty를 줄여 CPU 성능을 높일 수 있는 branch prediction을 구현하여 control hazard를 추가로 해결 하려고 한다.

설계 규격

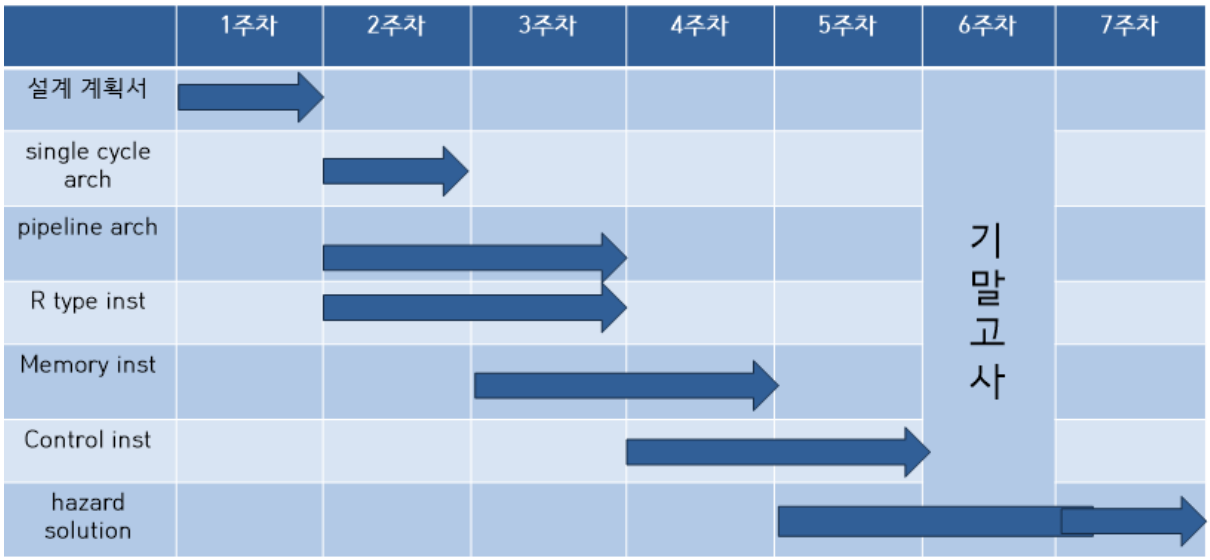
Pipeline Architecture에서는 5단계의 stage를 통하여 명령어를 수행한다. 각각의 stage는 IF,ID,EX,MEM,WB로 이루어진다. IF는 Instruction Memory(Program Memory)에서 PC의 해당 주소에 따른 Instruction을 Fetch(명령어 인출)하는 과정이다. ID는 Fetch해온 Instruction을 Decoding 함과 동시에 register field를 read하는 과정이다. EX는 ALU(Arithmetic Logic Unit)을 이용하여 execute 하거나 effective address를 계산하는 과정이다. MEM은 Data Memory에 접근하여 Memory에 data를 Write하거나 Read하는 과정이다. 마지막으로 WB는 Write Back의 과정으로 Register File에 register update하는 과정이다. Pipeline을 하게 되면 Single cycle model에서보다 instruction throughput이 이상적으로는 stage의 수만큼 배가되어 증대된다. Pipelining을 위해

		서는 각 단계 사이마다 Pipeline register를 넣어야 한다. 또한 정확한 Instruction의 동작을 위해 control signal 도 제어해야 한다. Instruction Decoding 과정(ID)에서 instruction의 상위 6bit(OPcode)를 Input으로 하여 Control Unit을 만들어 내고 9개의 control signal을 만들어 data path를 제어한다. 그중 ALUOp signal(2bit)과 funct field(6bit)을 이용하여 ALU Operation bit를 결정하고 이 4bit은 ALU의 operation을 결정한다. Pipeline 구조에서 다음 명령어가 다음 클럭 사이클에 수행될 수 없는 상황이 존재하는데 이를 hazard라고 하며 종류는 3가지가 있다. 첫번째 hazard는 structural hazard라고 불리며 이는 conflict for use of a resource 때문에 생긴다. 만약 IM과 DM이 분리되어 있지 않다면 Mem stage의 instruction과 3 clock cycle 뒤의 IF stage의 instruction이 Memory라는 자원을 가지고 충돌을 일으킬 것이다. 우리는 IM과 DM이 분리 되어있는 harvard architecture를 선택하여 structural hazard를 극복하려고 한다. 두번째 hazard는 data hazard라고 불리며 명령어가 data dependency를 가질 경우 앞의 단계가 끝나기를 기다려야할 때 발생한다. Data hazard를 극복하는 방법 중 하나는 'forwarding' 혹은 'bypassing', 즉 별도의 하드웨어를 추가하여 정상적으로는 얻을 수 없는 값을 내부 자원으로부터 일찍 받아 오는 것이다. 설계를 진행하면서 data hazard는 bypassing을 이용하여 해결 할 것이다. 하지만 load-use data hazard의 경우는 forwarding으로도 해결하지 못한다. lw의 경우 data dependency를 갖는 register의 data가 Mem단계 이후로 available한데, 다음 명령어의 source operand가 lw의 destination register라고 하면 필요한 data가 최소한 EX단계 직전까지는 available해야하는데 이는 bypassing으로 연결 할 수 없기 때문에 bubble을 삽입하여 pipeline stall을 하여야 한다. 세번째 hazard는 control hazard라고 불리며 conditional branch instruction 이후 pipeline에 이슈 된 명령어가 계획되지 않은 것일 때 발생한다. 이를 해결할 방법으로 branch가 evaluation 되기 전까지 다음 명령어를 fetch하지 않고 bubble을 삽입하는 stall on branch 방법을 이용하여 control hazard를 극복 할 것이다. 다음 명령어를 fetch하지 않고 stall 시키면 다음 명령어가 들어와서 target address에 있는 명령어와 충돌을 일으킬 일이 없기 때문에 control hazard가 발생하지 않을 것이기 때문이다. stall on branch에 의한 CPU 성능 저하 penalty를 줄이기 위하여 branch evaluation이 되는 stage를 EX에서 ID로 옮겨 stall on branch에 의한 penalty를 2 clock cycle에서 1 clock cycle로 줄일 것이다. 일차적 목표 달성 이후 branch prediction(taken prediction or Not taken prediction)을 이용하여 stall의 수를 줄여 CPU성능을 높일 것이다.
합성/분석	관련 기술	 Pipelining이 적용된 CPU는 5개의 stage로 이루어져 있다. 각 단계 사이에는 Pipeline Register가 있다. IF/ID, ID/EX, EX/MEM, MEM/WB라고 한다. 첫번째로, IF 단계이다. memory로부터 instruction을 가져온다. PC와 Instruction memory로 이루어져 있다. 두번째로 ID 단계이다. Instruction decode를 수행하고 register에서 값을 읽어온다. 32bit Register, Control Unit, Hazard Detection Unit이 있다. Control unit은 9개의 신호를 pipeline Register에 전달한다. 세번째로 EX 단계이다. Operation을 수행하고 address를 계산한다. ALU 와 Forwarding Unit으로 이루어져 있다. ALU는 arithmetic Logic Unit으로 산술, 로직 연산을 해준다. 네번째로 MEM 단계이다. Data Memory에 access하는 단계로 Data memory로 구성되어 있다. 마지막으로 WB 단계이다. 결과들을 register에 저장해준다. MEM/WB Register의 값이 Register와, ALU에 입력된다. Structure hazard, Data hazard, Control hazard 3가지의 hazard가 발생했는지 여부를 판별해주는 Hazard detection Unit은 ID 단계에 있다. ID 단계에서는 Instruction을 decoding 하여 operation 종류와 사용되는

register를 알게 되므로 hazard가 발생하는지 여부를 판별할 수 있기 때문이다. 레지스터에 새로운 데이터 값이 저장되지 않은 상태로 같은 Register를 사용하는 다음 instruction을 pipeline으로 수행한다면 Data Hazard가 발생한다. 이는 계산된 데이터를 바로 저장해주는 Forwarding(Bypassing)을 사용하여 해결한다. Register에 값이 저장되는 WB단계까지 stall 할 필요가 없게 된다. 하지만 별도의 Forwarding Unit 모듈을 만들어야 한다. 이 Forwarding Unit은 EX 단계에 있고, ALU의 입력으로 연결된다. 마지막으로, control hazard를 해결하기 위해 Control hazard solution unit회로를 추가하여 stall on branch 기법을 이용할것이다.

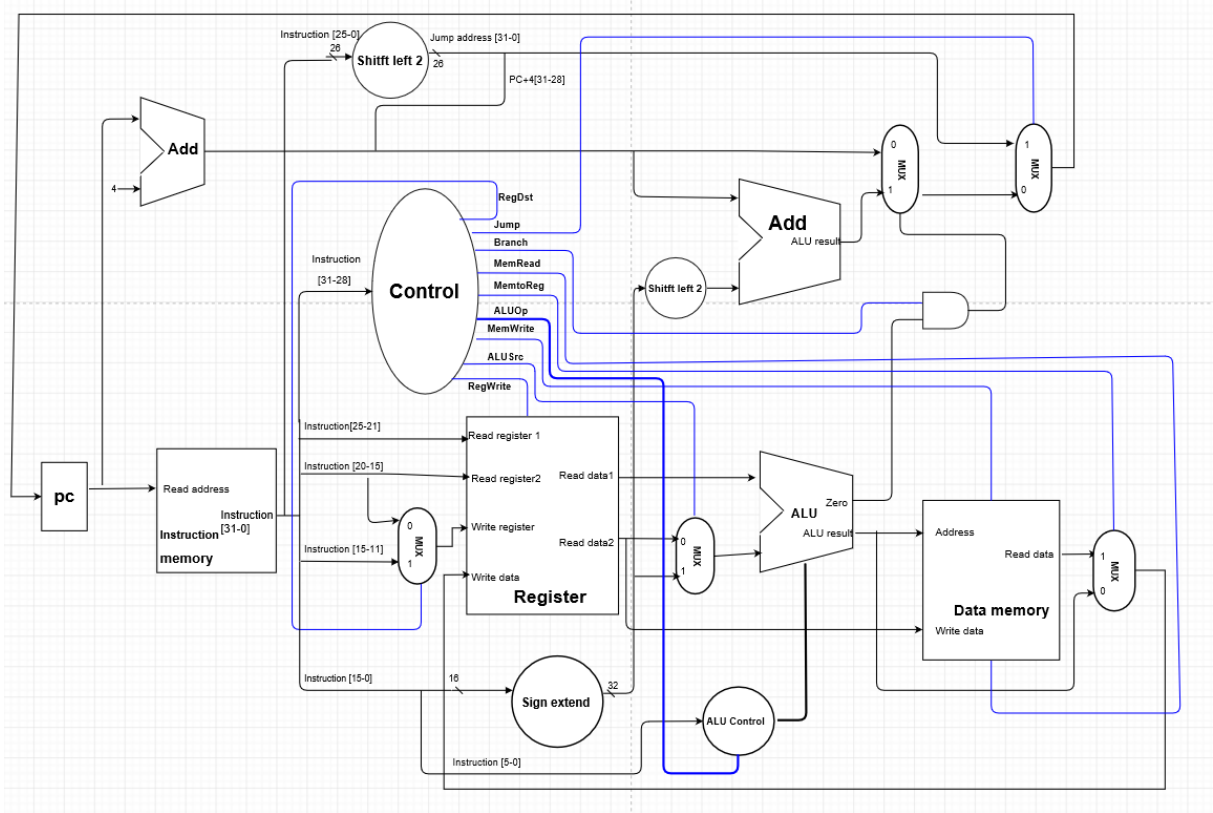
제작 설계 계획

1. 설계 내용 및 절차



이번 term project에서 본 팀은 1주차에 설계 계획서를 작성하면서 전체적인 간략한 구상 및 설계 계획을 정하였다. 2주차에서는 pipeline architecture의 밑바탕이 될 구조인 single cycle arch를 만들고 이를 토대로 3주차에 pipeline architecture를 완성시킨다. R type instruction을 동시에 완성시켜 pipeline architecture를 검정할 것이고 4주차에는 memory instruction, 5주차에는 control instruction, 6주차에 기말고사이므로 7주차에 hazard instruction을 완성 시켜 전체적인 CPU core를 구현해낼 것이다.

2. Single cycle Architecture 설계



Pipeline Architecture를 설계하기에 앞서서 Single cycle Architecture을 설계 할 것이다. Pipeline Architecture

는 Single cycle Architecture를 바탕으로, 각 stage 사이에 Pipeline register를 추가하여 파이프라인의 stage를 분리하고 Data hazard와 Load-use data hazard를 방지하기 위한 Forwarding Unit과 Hazard Detection Unit을 추가함으로써 Pipeline Architecture를 설계 할 것이므로 Single cycle Architecture를 먼저 구상하는 것이다. Single cycle Architecture를 기본적인 R-format instruction(add, sub)을 통해 간단히 검증한후 pipeline Architecture 설계로 넘어 갈 것이다.

3. Opcode에 따른 Control signal 조합

Control signal을 관리하는 Control Unit은 Instruction Decoding 과정에서 Instruction의 상위 6비트(opcode)를 input으로 받아 control signal 내보낸다. 41개의 Instruction 마다 다른 Datapath를 제어하기 위해 9개의 control signal 조합을 알아야 한다. 아래의 표에 41개의 명령어의 opcode와 control signal을 정리하였다. 단, 표에는 작성되지 않았지만 설계 과정에서 추가되는 control signal이 있을 수도 있음을 명시한다.

	OPcode	RegDst	RegWrite	ALUSrc	ALUOp	Mem Read	Mem Write	Mem toReg	Branch	Jump
ADD	000000	1	1	0	10	0	0	0	0	0
ADDU	000000									
ADDI	001000	0	1	1	10	0	0	0	0	0
ADDIU	001001									
AND	000000	1	1	0	10	0	0	0	0	0
ANDI	001100	0	1	1	10	0	0	0	0	0
BEQ	000100	0	0	0	01	0	0	0	1	0
BNE	000101	0	0	0	01	0	0	0	1	0
SLT	000000	1	1	0	10	0	0	0	0	0
SLTU	000000									
SLTI	001010	0	1	1	10	0	0	0	0	0
SLTIU	001011									
SUB	000000	1	1	0	10	0	0	0	0	0
SUBU	000000									
SW	101011	0	0	1	00	0	1	0	0	0
SH	101001	0	0	1	00	0	1	0	0	0
SB	101000	0	0	1	00	0	1	0	0	0
LW	100011	0	1	1	00	1	0	1	0	0
LH	100001	0	1	1	00	1	0	1	0	0
LHU	100101									
LB	100000	0	1	1	00	1	0	1	0	0
LBU	100100									
NOP	000000	0	0	0	0	0	0	0	0	0
NOR	000000	1	1	0	10	0	0	0	0	0
OR	000000	1	1	0	10	0	0	0	0	0
ORI	001101	0	1	1	10	0	0	0	0	0
XOR	000000	1	1	0	10	0	0	0	0	0
XORI	001110	0	1	1	10	0	0	0	0	0
MOVN	000000	1	1	0	10	0	0	0	0	0
MOVZ	000000	1	1	0	10	0	0	0	0	0
LUI	001111	0	1	1	10	0	0	0	0	0
J	000010	0	0	0	X	0	0	0	0	1
JAL	000011	0	0	0	X	0	0	0	0	1
JALR	000000	0	0	0	X	0	0	0	0	1

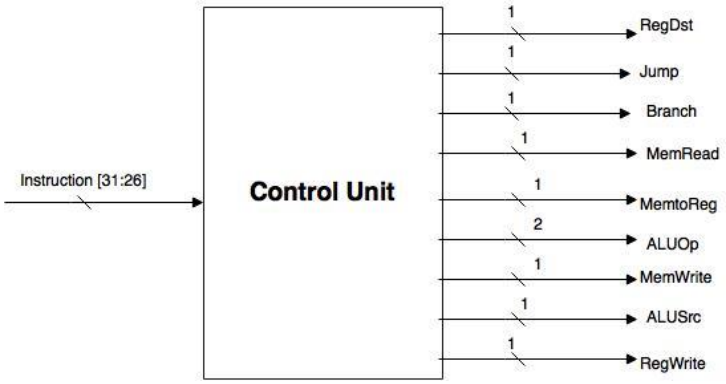
JR	000000	0	0	0	X	0	0	0	0	1
SLL	000000	1	1	0	10	0	0	0	0	0
SLLV	000000	1	1	0	10	0	0	0	0	0
SRA	000000	1	1	0	10	0	0	0	0	0
SRAV	000000	1	1	0	10	0	0	0	0	0
SRL	000000	1	1	0	10	0	0	0	0	0
SRLV	000000	1	1	0	10	0	0	0	0	0

4. Pipeline Architecture 설계

Single cycle architecture를 기반으로 pipeline register 4개와 Hazard 해결을 위한 Unit 및 control hazard 해결을 위한 solution을 추가하여 pipeline Architecture를 설계한다. Pipeline register는 5 stage 사이사이에 넣어 필요로 하는 data를 read하여 다음의 clock에 다음 stage로 data를 넘긴다. 따라서 Instruction Fetch와 Instruction Decode 사이의 IF/ID Register에는 PC+4와 32bit instruction을 저장한다. ID/EX Register에는 Control Signal에서 나온 Signal과 PC+4 및 Forwarding에 필요한 Data들을 넘기게 된다. Pipeline에 대한 Architecture는 앞 장의 이미지를 참고하면 된다. 이때 Single Cycle과의 큰 차이는 Forwarding Unit과 Hazard Detection Unit을 추가하여 Data Hazard를 제거하고 control hazard를 극복하는 것이다.

5. Control Unit

Control Unit 은 Pipeline의 두번째 stage에 있는데, IF/ID Pipeline Register에서 Register Read를 하고 Instruction을 읽어와 Instruction decoding을 한다. Instruction의 상위 6비트 [31:26]에 해당하는 Opcode의 값을 input으로 하여 R-type, Load/Store, Branch..등등의 format에 따라 해석하여 datapath를 제어하기 위한 signal 들을 생성해준다. RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Jump 9개의 output을 갖는다. 단 설계를 진행하면서 추가되는 control signal이 있을수 있음을 명시한다. 블락도는 다음과 같다.



6. Forwarding Unit

데이터가 Register파일에서 읽을 수 있게 되기 전에 계산된 데이터를 Forwarding해주면 지연 없이 실행될 수 있다. 앞의 Instruction에서 WB단계에서 쓰기를 하려는 register를 그 다음의 instruction이 EX단계에서 사용하려고 하면 ALU의 입력으로 계산된 값을 forwarding 시켜주어야 한다. 그렇지 않으면 계산되기 이전의 register값을 Instruction에 사용하게 되어 틀린 결과가 나올것이다. 다음과 같은 해저드 조건을 알 수 있다.

1a. EX/MEM.RegisterRd= ID/EX.RegisterRs

1b. EX/MEM.RegisterRd= iD/EX.RegisterRt

2a. MEM/WB.RegisterRd= ID/EX.RegisterRs

2b. MEM/WB.RegisterRd= ID/EX>RegisterRt

위의 조건은 forwarding이 필요하지 않은 경우도 포함한다. 첫째로 , 어떤 명령어들은 Register에 쓰기를 하지 않기 때문에 forwarding이 필요하지 않게 된다. 이는 Regwrite 신호가 활성화되어 있는지 확인하여 EX단계와 MEM 단계 동안에 파이프라인 Register의 WB control Field를 조사하면 Regwrite 신호가 인가되어 있는지를 알 수 있다. 둘째로, $0 Register는 항상 상수0의 값이기 때문에 그 값이 바뀌지 않는다. 두번째로 수행

될 Instruction의 destination Register가 $0라면 forwarding 해줄 필요가 없다. 이는 위의 조건에 EX/Wb.RegisterRd!=0을 추가하여 해결한다. 이를 다시 종합하여 조건들을 써보면 다음과 같다.
(교과서 296p~299p 인용)
1. EX hazard
a. If (EX/MEM.RegWrite)
And (EX/MEM.RegisterRd ≠0)
And(EX/MEM.RegisterRd = ID/EX.RegisterRs)
Forward A = 10
b. If (EX/MEM.RegWrite)
And(EX/MEM.RegisterRd ≠0)
And(EX/MEM.RegisterRd = ID/EX.RegisterRt)
Forward B = 10
2. MEM hazard
a. If (MEM/WB.RegWrite)
And (MEM/WB.RegisterRd ≠ 0)
And not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
And (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
And (MEM/WB.RegisterRd = ID/EX.RegisterRs)
Forward A = 01
b. If (MEM/WB.RegWrite)
And (MEM/WB.RegisterRd ≠ 0)
And not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
And (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))
And (MEM/WB.RegisterRd = ID/EX.RegisterRt)
Forward B = 01
그러므로 Forwarding Unit의 입력은 ID/EX.RegisterRS, ID/EX.RegisterRt , ID/EX.RegisterRt , ID/EX.RegisterRd , EX/MEM.RegisterRd, EX/MEM.Rt.Regwrite, MEM/WEB.Regwrite 와 연결되고 출력은 Forward A, Forward B로 ALU에 연결된 2개의 MUX의 입력이 된다.

7. Hazard Detection Unit
6번에서 설명했듯이 대부분의 Data Hazard는 Forwarding Unit으로 해결 할 수 있다. 하지만 Load 명령어의 경우 Forwarding으로 hazard를 해결 못하는 경우가 있다. 이를 load-use data hazard라고 하며 이를 해결하기 위해 Hazard Detection Unit이 필요하다.

If(ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
Stall the pipeline
즉, 이 유닛은 ID 단계에서 Instruction Decoding하면서 load instruction인지 확인 한 후, load instruction의 destination register와 그 다음의 instruction의 source operand register 사이에 data dependency가 있는 지를 확인하고 만약 있다면 1 clock cycle 동안 bubble을 추가하여 pipeline을 stall한다(data hazard와 load-use data hazard의 차이). 이후 Forwarding Unit으로 data hazard를 해결한다.

8. Control Hazard Solution
Control Hazard를 해결하기 위한 Solution으로 우리 조는 stall on branch 기법을 사용 할 것이다. Stall on branch란 branch 명령어라고 Instruction Decoding 되면 branch evaluation 되기 전까지 bubble을 삽입하여 stall을 시키는 방법이다. 또한 branch로 인해 발생하는 hazard의 penalty를 줄이기 위하여 branch의 evaluation여부와 target address가 계산되는 pipeline stage가 EX stage가 아닌 ID stage가 되게 설계하려고 한다. Control Hazard를 가장 기본적인 solution으로 해결한 후 가능하다면 prediction을 이용하여 control hazard를 해결해 보려고 한다.

시험/평가

검증 계획

1. 전체적인 검증계획

이번 Term project인 CPU integer processing core를 설계를 하고 이를 검증하기 위해서는 Test Bench 환경을 구축하고 작성한 Instruction들이 작동되는지 확인해야한다. Test Bench 환경을 구축하기 위해서 Linux OS에서 nano나 vi 명령어를 사용하여 아래의 PDF에 주어진 Test Bench code 를 작성할 것이다. Code는 Verilog language로 작성되며 Linux OS의 명령어인 iveri를 사용하여 compile하게된다. Test Bench와 core의 동작을 확인하기 위한 waveform은 gtkwave 명령어로 확인한다. GTK simulator는 Linux 서버를 통해 다른 컴퓨터의 서버로서 확인 가능하다. 또한 검증을 위해 어셈블리 명령어를 작성하여야 하는데, 2진수의 어셈블리 명령어를 모두 손으로 작성하는 것에 무리가 있기 때문에 MIPS code를 16진수인 어셈블리로 converting 해주는 명령어인 'mips-as'를 사용하여 TEST Vector를 작성한다.

```
module TB();
localparam INITIAL_FILE = "test.hex";
localparam CLK_PERIOD = 30;
localparam RST_DELAY = CLK_PERIOD * 5.2;
localparam SIMULATION_TIME
= CLK_PERIOD * 100.2;
localparam PM_ADDR_WIDTH = 10;
localparam DM_ADDR_WIDTH = 10;
wire [29:0] pm_addr;
wire [31:0] pm_read_data;
wire        pm_cs;
wire [29:0] dm_addr;
wire [31:0] dm_read_data;
wire [31:0] dm_write_data;
wire [31:0]   dm_we;
wire          dm_cs;
reg rst;
reg clk;
CORE core(
.o_pm_addr (pm_addr),
.o_pm_cs (pm_cs),
.o.pm_data (pm_read_data),
.o_dm_addr (dm_addr),
.o_dm_cs (dm_cs),
.o_dm_we (dm_we),
.o_dm_data (dm_write_data),
.i_dm_data (dm_read_data),
.i_clk (clk),
.i_rst (rst)
);
SRAM # (
.WIDTH(32)
.ADDR_WIDTH (DM_ADDR_WIDTH)
)dm(
.i_data (dm_write_data),
.i_addr (dm_addr[DM_ADDR_WIDTH-1:0]),
.o_data (dm_raed_data),
.i_cs (dm_cs),
```

```
SRAM # (
.INITIAL_FILE(INTITIAL_FILE),
.WIDTH(32)
.ADDR_WIDTH (PM_ADDR_WIDTH)
)pm(
.i_data (32'bx),
.i_addr (pm_addr[PM_ADDR_WIDTH-1:0]),
.o_data (pm_raed_data),
.i_cs (pm_cs),
.i_we (4'd0),
.i_clk (clk)
);
initial begin
rst = 1'b0;
#(RESET_DELAY)
rst = 1'b1;
#(CLK_PERIOD)
rst = 1'b0;
end
initial begin
clk = 0;
forever # (CLK_PERIOD/2) clk = ~clk;
end
initial begin
$dumpfile("TB.vcd");
$dumpvars(0,TB);
#(SIMULATION_TIME)
$dumpflush;
$finish;
End
Endmodule
```

.i_we (dm_we), .i_clk (clk));	

위의 Testbench Code를 이용하여 Simulation을 돌렸을 때 정상적으로 설계되었다면 아래와 같은 결과가 출력되어야 한다. 먼저, Pipeline Architecture를 사용하기 때문에 5개 Stage에서 서로 다른 명령어가 동작되기 때문에 Cycle Per Instruction 즉, CPI가 5가 되어야 한다. 또 3가지 Hazard가 발생하는 경우 Detection이 정상적으로 이루어지는지를 확인해야한다. 즉, Structure hazards, Data hazards, Control hazards가 발생하지 않고 instruction이 정상적으로 처리 되어야 한다. 더 정확히 말하면 Structure hazard는 conflict for use of resource로 발생하는데 우리는 Harvard Architecture로 설계하기 때문에 Architecture가 잘 연결 되었는지를 확인하면 된다. Data hazard는 Forwarding 혹은 Bypassing 해주는 Forwarding Unit의 정상작동 하는 지와 Load-use data hazard를 처리해주는 Hazard Detection Unit이 정상 작동하는지를 확인하여야 한다. Data hazard를 발생시키는 Mips code를 test 프로그램으로 하여 instruction이 정상 작동하는지 확인 할 것이다. Control hazard는 stall on branch 기법을 이용하여 branch instruction인 경우 branch evaluation의 결과가 나오기 전까지 bubble을 삽입하여 instruction이 fetch되지 못하게 하여 control hazard를 해결 한 후 control hazard를 발생하는 Mips-code를 test 프로그램으로 하여 instruction이 정상 작동하는지를 확인 할 것이다.

```
module LUT #(parameter WIDTH=8)(
        input wire [5:0] in,
        output wire [5:0] out);
LUT#(.ROM_FILE("LUT.dat"),.INPUT_WIDTH(8),.OUTPUT_WIDTH(8)) lut(.i(in),.o(out));
endmodule
```

위의 코드는 op code를 LUT형식으로 작성 후 올바르게 작성되었는지를 알기 위한 test bench코드의 일부이다. 이를 통해 LUT table의 완성을 확인한다.

2. 2주차 검증계획

2주차 에서는 single cycle architecture 설계와 이를 밑바탕으로 pipeline Architecture의 구상을 하며 OP code LUT를 작성한다. Single cycle architecture의 검증방법으로 R-type Instruction과 Memory Instruction, Control Instruction에서 대표 instruction들을 넣어 값을 확인하여 검증한다. Single cycle Architecture이 완성되면 이를 토대로 4개의 Pipeline Register을 넣어 5-stage-pipeline구조를 구상한다. 이번 설계에서 signed instruction과 unsigned instruction을 구별하지 않기 때문에 OP code LUT는 [ADD(ADDU), ADDI(ADDIU), SLT(SLTU), SLTI(SLTIU), 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] 와 같은 34개의 항목의 LUT Table을 완성시킨다. 이를 확인하기 위해 LUT를 확인하기 위한 testbench 환경을 만들어 대응되는 출력 값이 나오는지를 검증한다.

addi $t3, $t3, 6 => 216B0006

add $t1, $t3, $t3 => 016B4820 (single cycle architecture를 검증 할 mips & assembly code)

이에 따른 결과로 $t3에는 6이 저장되고 $t1에는 12가 저장되어 있어야한다.

3. 3주차 검증계획

3주차에서는 5-stage-pipeline 구조를 완성시켜 동작하고 R-type Instruction을 구현 시켜 동작하는 것을 검증한다. 5-stage-pipeline 구조를 검증하기 위해서는 간단한 R-format(add) Instruction을 넣어 1 cycle에 1 stage를 수행하는 CPI=1 인 것을 gtkwave simulator을 통해 wave form으로 확인한다. 구현 시켰을 때 오류가 난 부분은 4주차까지 계속 수정하며 4주차에서 완벽하게 구현하여 검증한다. R-type Instruction의 검증방법은 모든 Instruction을 하나씩 확인하여 data 값과 address 값을 확인한다. 또한 여러 개의 Instruction을 연속으로 사용했을 시에 작동하는지를 확인한다. 이 때 3주차에서는 아직 Hazard solution을 실행하지 않았기 때문에 Hazard 발생에 유념하여 확인한다.

addi $t3, $t3, 6 => 216B0006

add $t1, $t3, $t3 => 016B4820 (single cycle architecture를 검증 할 mips & assembly code)

이에 따른 결과로 $t3에는 6이 저장되고 $t1에는 12가 저장되어 있어야한다.

4. 4주차 검증계획

4주차에서는 Pipeline Architecture을 완벽하게 구현하여 3주까지 진행했던 R-type Instruction의 동작과 4주차까지 진행한 Memory instruction에 대한 동작을 검증한다. 이는 gtkwave simulator을 사용하여 wave form을 관찰하여 제대로 구현했는지를 판단한다. Memory instruction의 검증방법은 sw는 저장한 data의 값이 memory에 잘 저장되었는지 lw는 원하는 address에서 불러온 data의 값이 정확한지 gtkwave simulator을 통한 wave form으로 확인한다. 여기까지 구현이 되었고 오류가 발생하지 않았다면 Memory Instruction을 사용한 lw와 sw를 통해 값을 저장하거나 불러오고 R-type instruction을 통해 연산을 수행하는 mips code를 적용시켜 검증한다. 이 때 4주차에서는 아직 Hazard solution을 실행하지 않았기 때문에 Hazard 발생에 유념하여 확인한다. 그러므로 Data Hazard를 발생시키지 않는 Mips code를 적용해야한다.

Lw $s0, 20($t1) => 8D300014

Sw $s0, 20($t1) => AD300014

$t1에 저장된 memory에서 5번째에 저장된 것을 $s0에서 load하고 다시 store 한다. 이를 응용하여 실제 $t1에 값을 넣어 load 및 store를 확인한다.

5. 5주차 검증계획

5주차에서는 Control Instruction 및 기타 Instruction을 구현하여 검증한다. Control Instruction은 정확한 branch evaluation의 결과(ALU의 Zero signal의 정확한 작동 필요)와 더불어 정확한 주소의 PC의 값으로 이동(current PC와 offset을 sign-extension하고 shift left 2bit한 Lable의 주소를 더하는 adder의 정상 작동 필요)했고 그 다음 instruction을 수행 했는지를 gtkwave simulator을 통해 wave form으로 확인하여 검증한다. 기타 Instruction들 또한 조건에 맞는 기능을 잘 수행하였는지를 wave form으로 data 값을 확인한다. 이 때 5주차에서는 아직 Hazard solution을 실행하지 않았기 때문에 Hazard 발생에 유념하여 확인한다. 그러므로 Control Hazard를 발생시키지 않는 Mips code를 적용해야한다.

7. 7주차 검증계획

7주차에서는 CPU core에 해당하는 모든 instruction의 구현을 끝내고 최종 시연을 해야 한다. 그동안 진행 했던 R-type Instruction과 Memory Instruction, Control Instruction, 기타 Instruction에 대한 Hazard Solution을 구현하여 Hazard를 제거 후 검증해야 한다. 검증은 MIPS code로 data hazard와 control hazard가 발생하게끔 coding하여 assembly code로 바꾸어 TestBench의 initial file로 적용한다. 이를 gtkwave simulator로 waveform을 확인하여 forwarding Unit의 Bypassing을 통하여 data hazard가 해결 되는 지와 hazard Detection Unit을 통하여 Load-use data hazard가 해결되는지를 확인하여하 한다. 또한 Harvard structure로 구현 했기 때문에 structure hazard가 발생하지 않는다. 그러므로 structure hazard를 발생시키는 code를 넣었을 때 정상작동 하는 것으로 이를 검증한다. 그리고 control hazard solution 즉, stall on branch를 통하여 control hazard가 방지되는지 확인하여야 한다. 이는 branch가 taken 되었을 때와 not taken 되었을 때의 주소에 다른 출력값을 내는 mips code를 coding 하고 이를 testbench의 initial file로 적용한후 검증한다. Stall on branch가 제대로 적용이 된다면 branch 아래의 instruction의 결과를 다르게 하여도 target address의 instruction이 바뀌지 않는다면 결과값은 동일할 것이기 때문이다.

add $s0, $t0, $t1 => 01098020

sub $t2, $s0, $nat3 => 020B5022 (data hazard 확인 Mips code & assembly code)

---

lw $s0, 10($s2) => 8E50000A

add $s2, $s3, 1 => 22720001

add $s2, $s3, 1 => 22720001

add $s2, $s3, 1 => 22720001 (structure hazard 확인 Mips & assembly code)

---

lw $s0, 20($t1) => 8D300014

sub $t2, $s0, $t3 => 020B5022 (load-use data hazard 확인 mips & assembly code)

---

add $s4, $s5, $s6 => 1232C00A
beq $s1, $s2, 40 => 02B6A020
lw $s3, 300($s0) => 8E13012C
add $s7, $s8, 10 => 23D7000A (branch hazard 확인 mips & assembly code)

일정

약 7주차로 진행되는 이번 term project에서는 기말고사를 제외하고 6주차로 진행하기로 팀원들과 협의 하였다. 최종 시연일은 6월 26일이며 중간결과발표가 6월 8일에 실행된다. 팀원들과 의논 결과 진행의 완성도에 따라 발표를 할지 말지 생각하기로 하였다. 각 주차에 따른 진행 내용과 예상 산출물 또는 결과 시연 내용은 다음과 같다.

주차	진행 내용	예상 산출물 또는 결과 시연 내용
1	5월 11일 ~ 16일 - 설계 계획서 작성 - 전체적인 일정 및 설계 목표 단계 결정	- 설계 계획서 16일 제출
2	5월 17일 ~ 23일 - Single Cycle Arch 구현 - Single Cycle Architecture 설계를 기반으로 Pipeline Arch구상 (coding x)	- 진행회의보고서#1 23일 제출 - Single Cycle Arch의 동작 - pipeline Arch 의 구상도
3	5월 24일 ~ 30일 - Pipeline Arch 구현 - R type Instruction Verilog 구현 (SUB, ADD, OR 등) - 발표 유무 결정	- 진행회의보고서#2 30일 제출 - R type Instruction Data path - R type Instruction Verilog code - R type Instruction 동작 (Hazard를 해결하지 않아 Hazard 발생 가능성 있음)
4	5월 31일 ~ 6월 8일 - Memory reference instruction Verilog 구현 (LW, SW 등) - 만약 발표 한다면 발표 ppt 작성 및 발표 준비	- 진행회의보고서#3 8일 제출 - 중간결과발표 8일 (발표 ppt) - Pipeline Arch 검증 - Memory Instruction Data path - Memory Instruction Verilog code - Memory Instruction 동작 (Hazard를 해결하지 않아 Hazard 발생 가능성 있음)
5	6월 9일 ~ 6월 11일 - Control Instruction Verilog 구현 (Jump, beq 등) - 기타 다른 Instruction 구현 (SLL, MOVE 등)	- Control Instruction Data path - Control Instruction Verilog code - Control Instruction 동작 (Hazard를 해결하지 않아 Hazard 발생 가능성 있음)
6	기말고사 기간	x
7	6월 22일 ~ 6월 26일 - Hazard Detection Unit 구현 - Forwarding Unit 구현 - 결과 보고서 작성 - 공학인증 포트폴리오 업로드 - 최종 시연 준비	- 기타 Instruction Verilog code - Hazard 제거 - 결과 보고서 26일 제출 - CPU core에 대한 최종 작동 시연

역할 분담

이번 Term Project에서는 총 3명이 참여하여 진행된다. 7주차 동안 팀원들은 자신의 역할을 분담하여 맡아 프로젝트를 진행하고 각자의 역할 분담은 다음과 같다. 각자 맡은 바를 최우선 하되 프로젝트의 진행속도 및 난이도 등과 각 팀원들의 일정을 고려하여 아래의 역할분담은 언제든지 바뀔 수 있다.

<2012122249 장윤봉>

	- 설계계획서 작성(설계계획, 설계규격) - 진행회의보고서#1 작성 - Pipeline Arch 구상 및 구현 - Control Instruction Verilog code - Test Bench code <2015124129 오혜빈> - 설계계획서 작성(관련기술, Flow chart, 검증계획) - 진행회의보고서#2 작성 - Single Arch 구상 및 구현 - Memory Instruction Verilog code - Test Bench code <2015124144 이다현> - 설계계획서 작성(설계목표, 일정, 역할분담) - 진행회의보고서#3 작성 - R type Instruction Verilog code - Hazard detection 및 Forwarding Unit Verilog 구현 - Test Bench GTK waveform Simulation => 전체적인 구상작업은 팀원들 모두 모여 진행할 예정 => 모든 구상 및 아이디어는 word 파일로 남겨 저장 => 추가적으로 해야 할 업무가 추가 될 수 있음