**FINAL PROJECT.**

# **Microprocessor 및 설계하는 microprocessor 대한 소개**

**-MICROPROCESSOR 란?**

중앙 처리 장치(CPU)라고도 하는 마이크로프로세서는 모든 컴퓨터와 많은 가정 및 전자 장치의 두뇌라고 할 수 있다.

최초의 마이크로프로세서는 1971년에 소개된 Intel 4004였다.

이는 그다지 강력한 성능을 발휘하지 못했으며 간단한 계산기로서의 역할을 수행했다.

80년대가 되어서야 컴퓨터가 각 가정의 집으로 들어왔고 마이크로프로세서 덕분에 사람들의 삶에 큰 영향을 미치게 되었다.

현재의 마이크로프로세서는 매우 정교한 작업을 수행할 수 있으며 훨씬 적은 공간을 차지할 뿐만 아니라 뛰어난 성능을 제공한다.

**-RISC VS CISC (ISA)**

ISA(Instruction Set Architecture)는 크게 RISC(Reduced Instruction Set Computer) 철학과 CISC(Complex Instruction Set Computer) 철학으로 구분지을 수 있다.

초기의 많은 프로세서는 CISC 철학으로 설계 되었으며 다양한 기능을 한번에 구현해주는 명령어와 주소지정 방식을 포함했다.

이는 프로그램 설계자에게 쉽게 더 많은 기능을 효율적으로 구현할 수 있도록 만들어주었다.

하지만 이는 하드웨어적으로 복잡한 설계 결과를 초래하게 되었고 특히 제어부를 복잡하게 만들었다.

또한 CISC 구조는 명령어마다 다른 길이를 가지며 전체적인 길이도 길었다.

명령어의 기능이 고정되어 최적화가 불가능하여 한 번에 여러가지 기능을 한번에 수행해야 하였고 실제로 사용하는 명령어는 많지 않았다.

또한 하드웨어의 구조가 복잡했고 이에 따라 전력소모도 많았다.

이러한 문제에 1970년대 말과 1980년대 초 RISC 철학을 등장하게 하였다.

RISC 철학은 하나의 명령어가 하나의 동작을 수행하도록 하는 것이다.

즉 명령어가 여러가지 기능을 하지 못하도록 하고 명령어당 클럭의 길이를 동일하게 맞추는 철학이다.

이로 인해 특정 기능을 구현하기 위해서는 명령어를 조합하는 컴파일러의 중요성이 높아지게 되었다.

컴파일러는 최적화 과정을 통해 최적의 명령어 셋을 만들어내고 이에 따라 명령어를 수행하며 주어진 동작을 수행할 수 있도록 하였다.

이러한 구조 덕분에 RISC는 균등한 명령어 길이, 적은 수의 명령어 형식, 적은 수의 주소지정 방식, 많은 수의 레지스터, 로드/저장구조, 암묵 피연산자와 같은 특성을 가지게 되었다.

요즘에는 high level language가 등장하였고 컴파일러의 최적화 능력이 매우 많이 개선되어 어셈블리어의 사용 빈도가 줄어들어 CISC 구조의 장점이 대부분 사라졌다.

이에 따라 저전력, 저 발열이 중요한 모바일 기기에서 유리한 RISC 철학에 기반하는 ARM에 기반한 프로세서를 주로 사용하고 있다.

아래 표는 RISC와 CISC를 비교한 표이다.

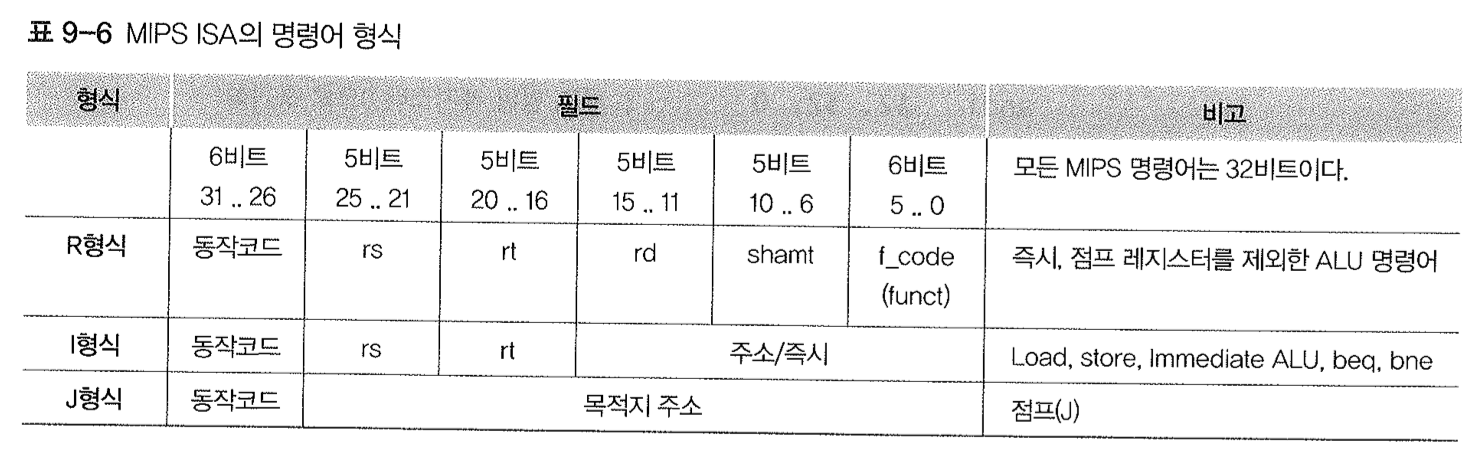
|  |  |
| --- | --- |
| RISC | CISC |
| 소프트웨어 강조 | 하드웨어 강조 |
| 단일 클럭, 간단한 명령어 셋 | 다중 클럭, 복잡한 명령어 포함 |
| 레지스터에서 레지스터:  로드와 스토어에 독립적인 명령어 | 메모리에서 메모리:  로드와 스토어가 명령어에 통합됨 |
| 초당 낮은 주기, 긴 코드 길이 | 작은 코드 사이즈, 초당 높은 사이클 |
| 메모리 레지스터에 많은 트랜지스터가 필요 | 복잡한 명령을 저장하는데 트랜지스터가 쓰임 |

**-설계하는 Microprocessor에 대한 설명**

Final project에서 주어진 Instruction Set을 동작할 수 있는 Microprocessor를 설계하였다.

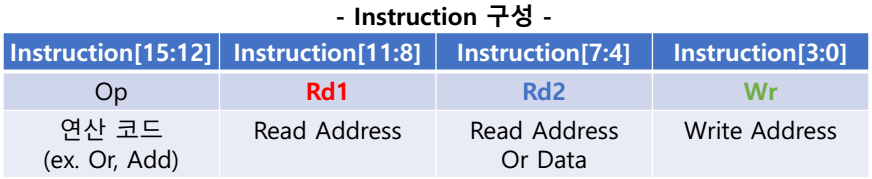
**-ISA**

먼저 주어진 Instruction Set과 RISC 구조인 MIPS와 비교해보았다.



MIPS ISA는 R-type, I-type, J-type 3가지 instruction 형식을 가진다.

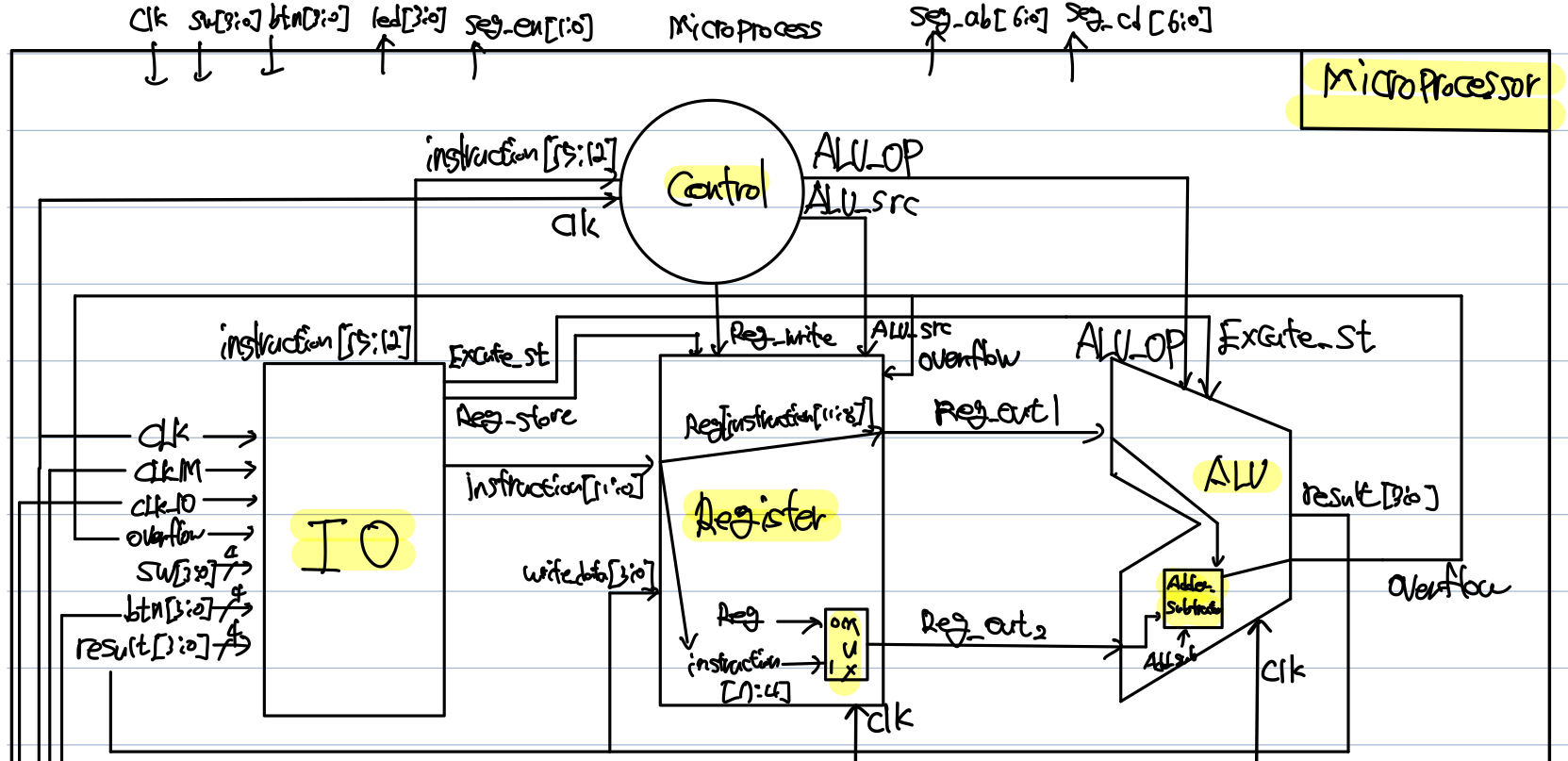
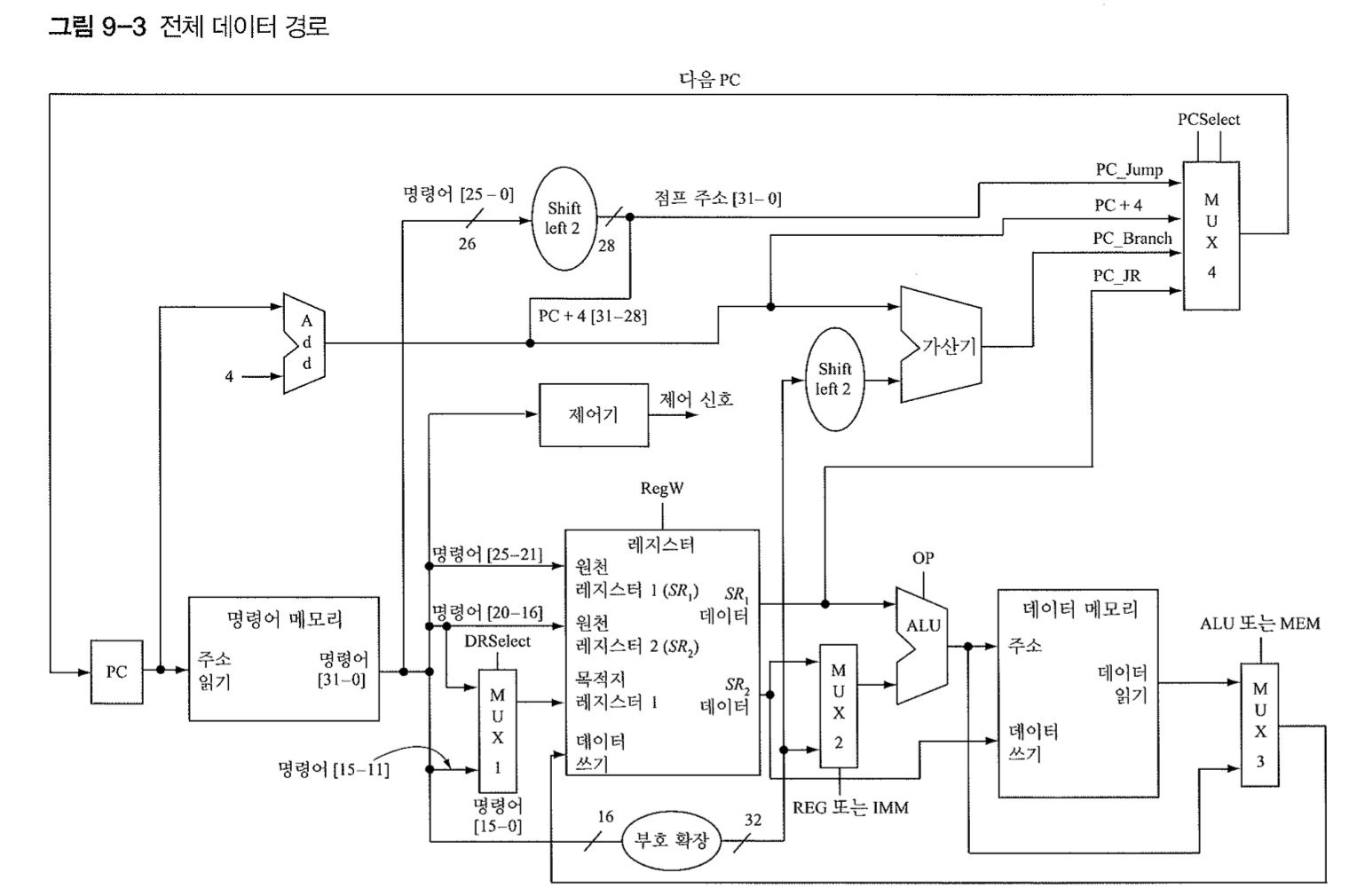
반면 주어진 Instruction Set은 아래와 같이 4bit의 Opcode와 Rd1, Rd2, Wr로 구성되어있는 단일 형식이었다.



또한 주어진 Instruction에서는 메모리를 참조하거나 메모리를 jump하는 일이 발생하지 않기 때문에 명령어의 구성에서도 차이가 발생하였다.

이러한 부분을 고려하여 설계를 진행하였다.

**-HW 구성**



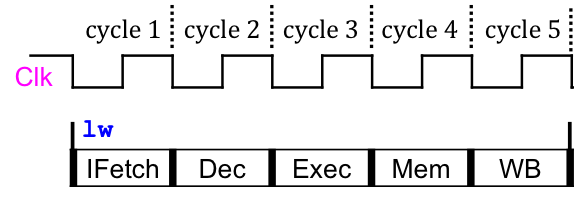
앞서 살펴본 ISA의 차이에서 HW적 차이도 발견할 수 있었다.

Memory에 대한 부분이 주어진 ISA에서는 없었고 메모리 주소에 대한 이동도 없으므로 비교적 간단하게 IO, Control Register, ALU로 코드를 구현할 수 있었다.

MUX는 Reg module 내부에 함께 구현하였는데 이유는 아래에 내용을 정리하면서 서술하였다.

**-cycle**

Multi cycle 기준으로 MIPS는 5개의 Cycle이 필요하다.



이는 lw에서 가장 많은 cycle을 사용하는데 이 cycle이 5개 인데 여기에 맞춰 모든 명령어를 5개의 cycle을 사용하도록 구성하였기 때문이다.

우리가 구현한 코드는 3개의 cycle을 사용하도록 구현하였다.

이는 계산시작, 오버플로우 확인, register 저장을 각각 1개의 cycle에 실행하기 위해서이다.

**-구성한 Microprocessor의 동작**

위와 같은 차이점을 통해 설계한 Microprocessor를 간단하게 설명하면 아래와 같다.

먼저 switch를 통해 Instruction을 받는다.

이 과정에서 한 명령어가 한 번에 입력되는 기존의 Microprocessor와 차이를 보인다.

project에서는 switch를 입력하고 버튼을 눌러야지 다음 state로 넘어가기 때문에 명령어를 4번의 state로 나눠 저장한다.

또한 MIPS와 달리 무조건 Opcode가 Instruction 상위 4bit에 저장되어 가장 먼저 결정되게 된다.

이렇게 먼저 입력된 Opcode를 이용하여 사용할 Register를 먼저 결정할 수 있으므로 Reg module은 delay에서 비교적 자유롭다.

이 때문에 MUX를 Reg module과 함께 구현하여 MUX를 따로 구현하여 발생하는 delay나 ALU와 함께 구성하여 발생하는 delay를 줄일 수 있도록 하였다.

또한 Control에서 어떠한 동작을 할지 선택하고 이를 구현할 수 있는 signal을 출력하는 과정 역시 Opcode가 결정되자마자 수행할 수 있다.

이를 이용하여 모든 signal이 미리 결정되어 연산이 시작될 때까지 대기한다.

다음으로 모든 Instruction이 결정되면 ALU가 계산을 수행을 시작하도록 신호를 출력한다.

이후 ALU에서 연산이 진행된다.

다음 cycle에서는 이 값이 overflow가 일어난 값인지 확인한다.

이후 다음 cycle에서 이 값을 저장할 것인지 판단하여 저장한다.

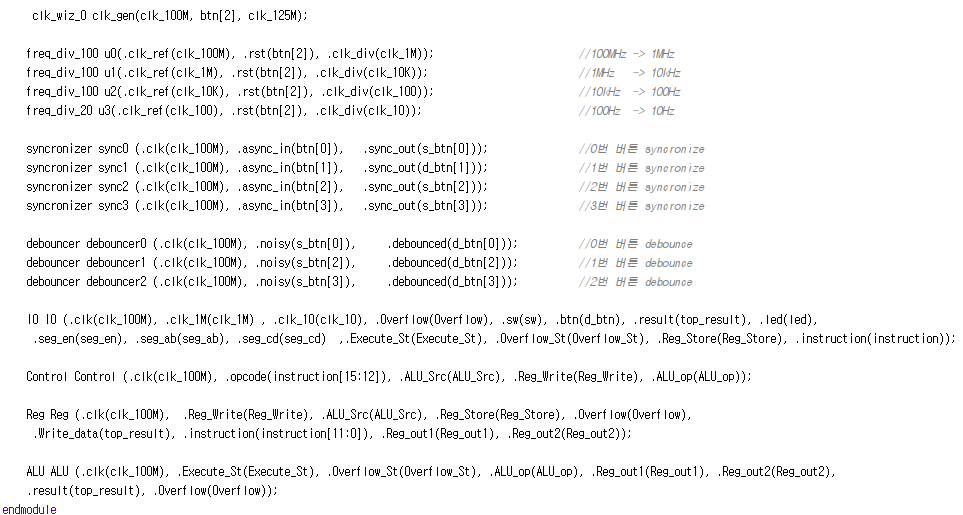
위와 같은 동작을 진행하도록 회로를 설계하였다.

# **설계한 microprocessor**

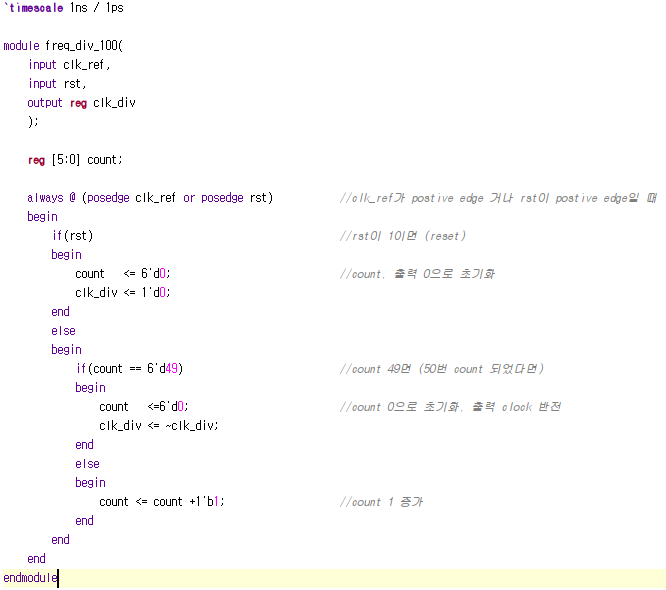
**Verilog Code / 주석**

<Microprocessor>





<freq\_div\_100>



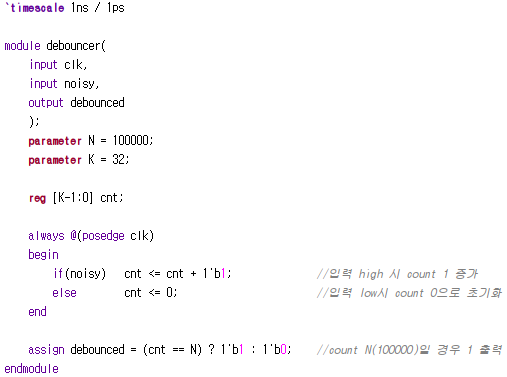
<freq\_div\_20>



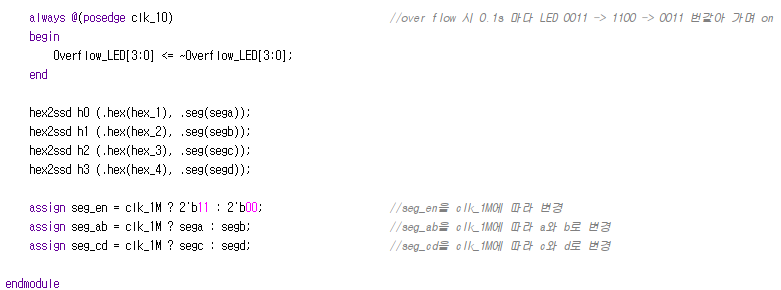
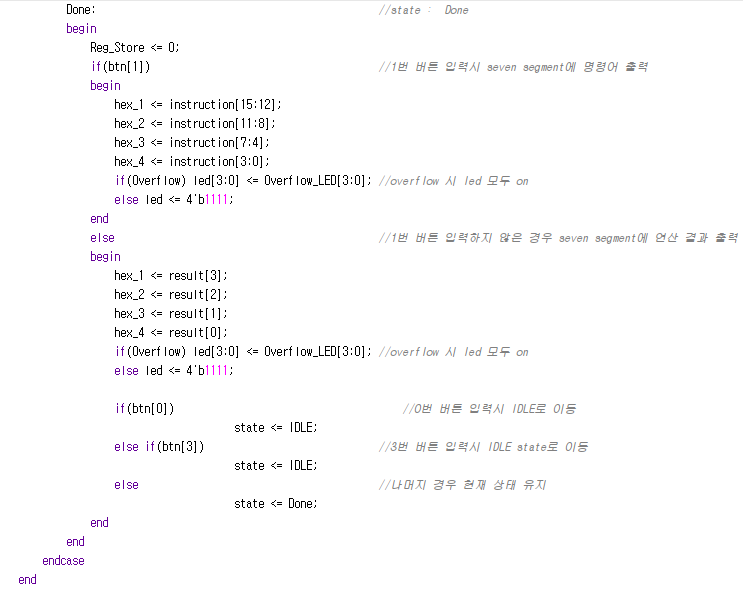
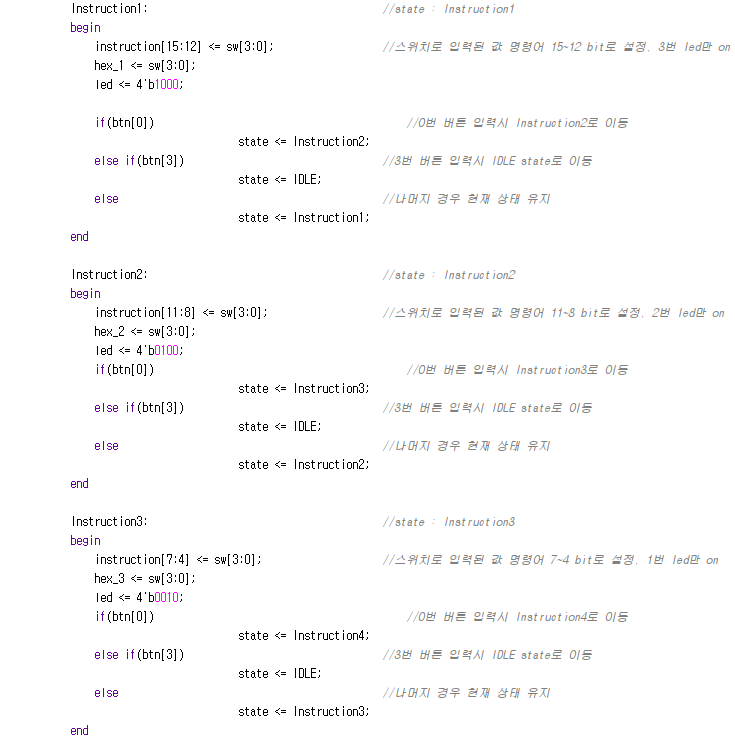
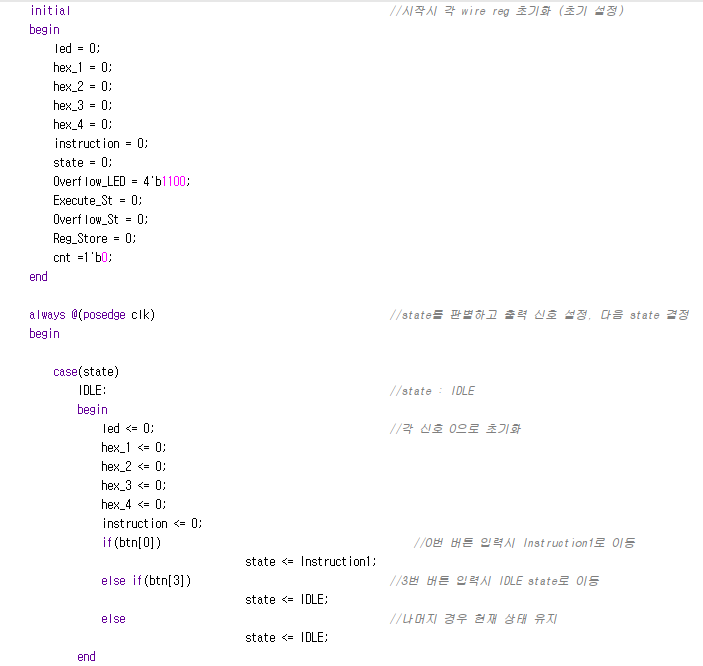
<synchronizer>



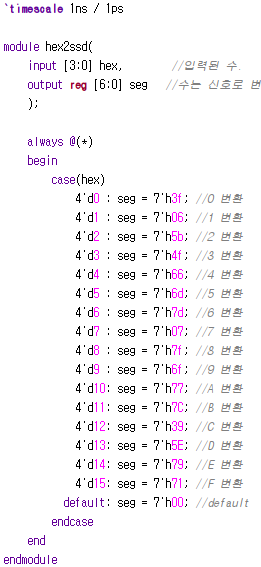
<debouncer>



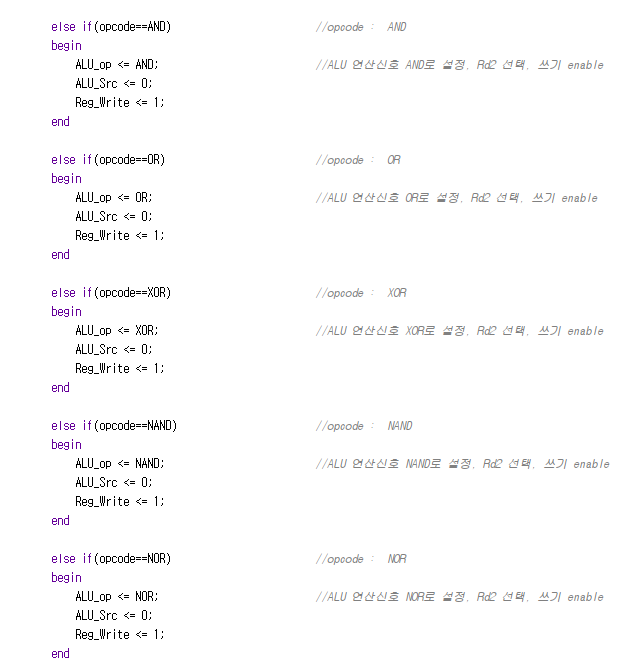
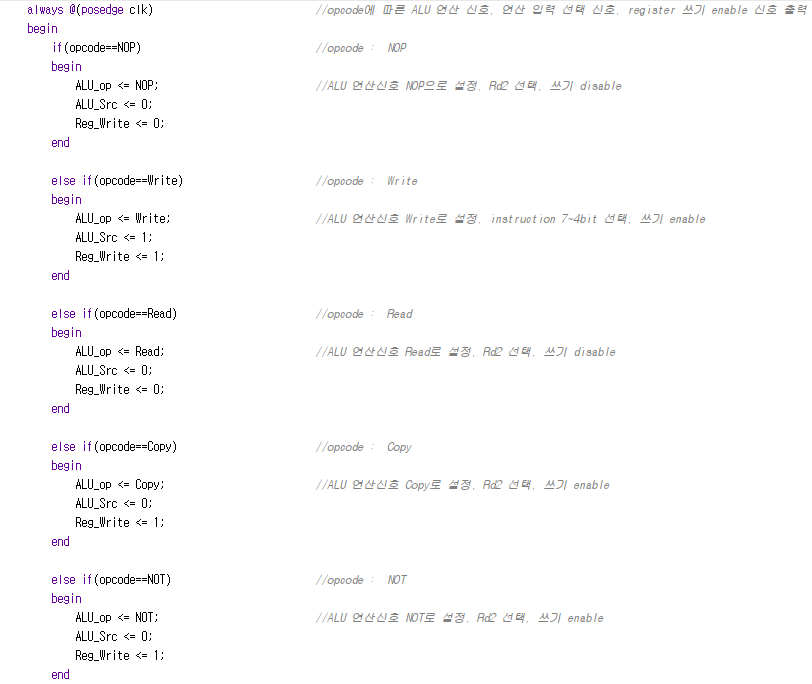
<IO>



<hex2ssd>



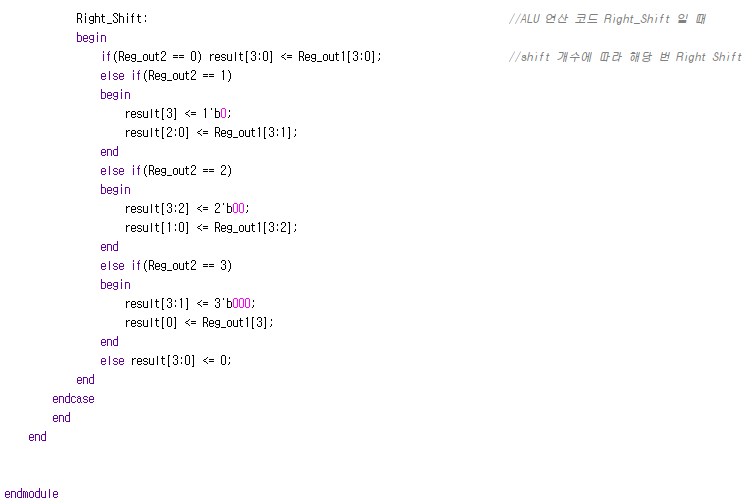
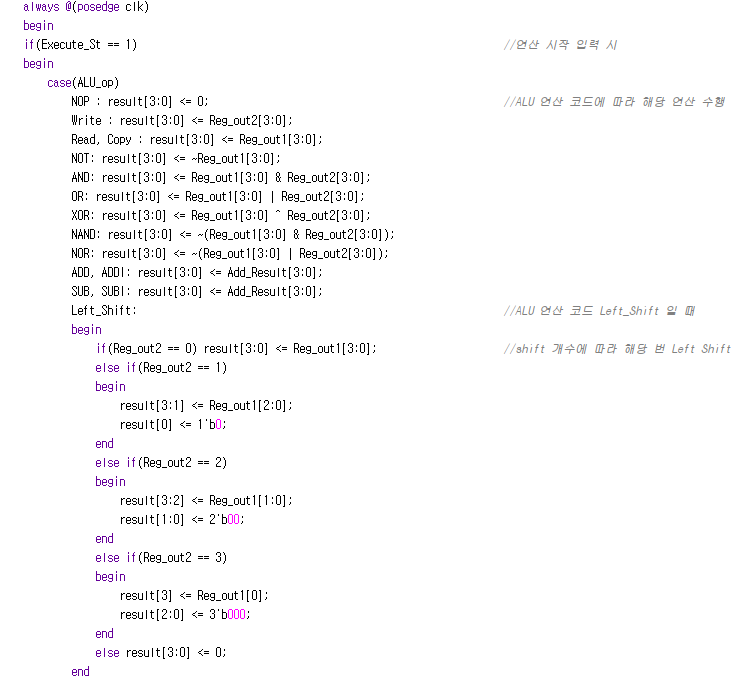
<Control>



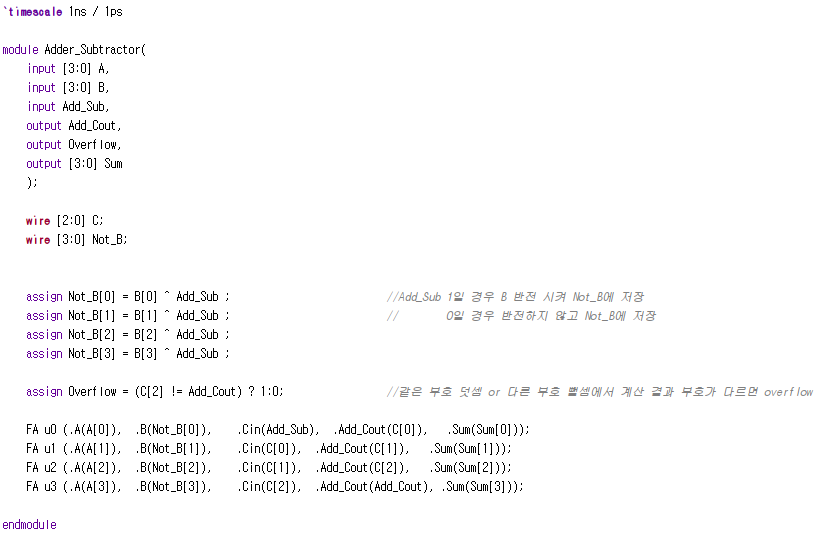
<Register>



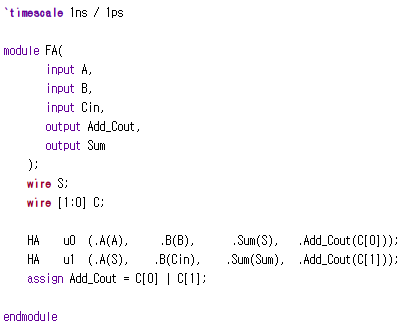
<ALU>



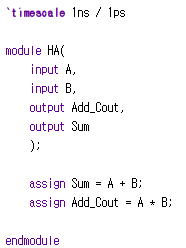
<Adder\_Subtractor>



<FA>

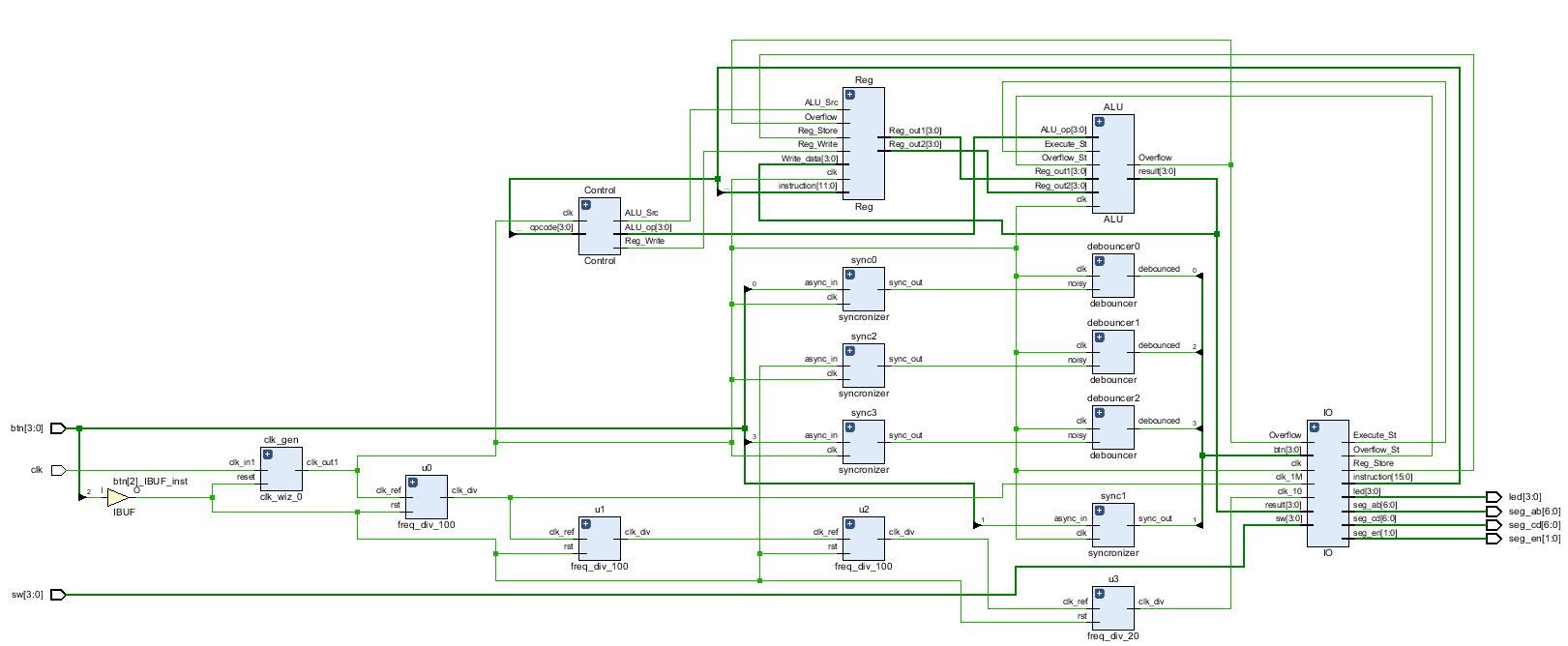


<HA>

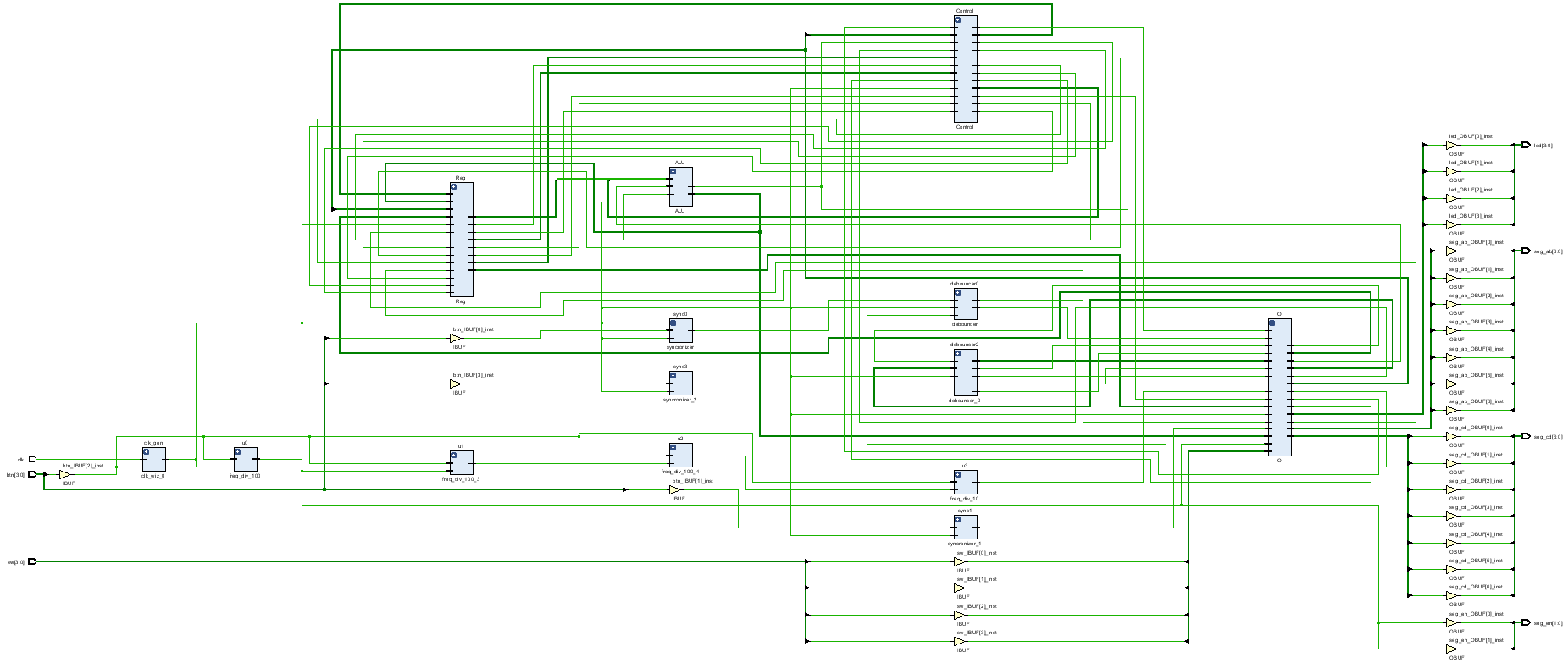


**Synthesis Report**

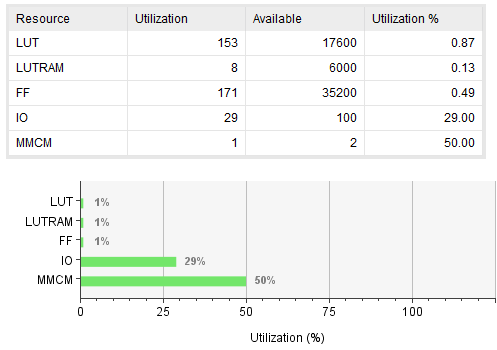
**-RTL Schematic**

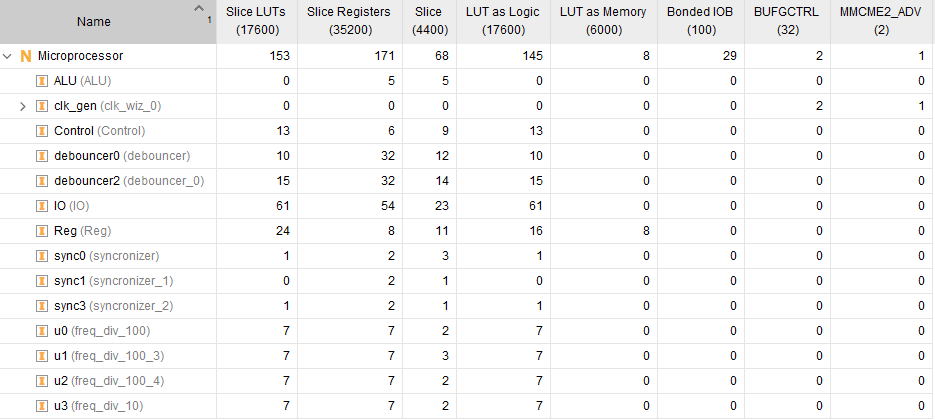
****

**-SYNTHESIS Schematic**

****

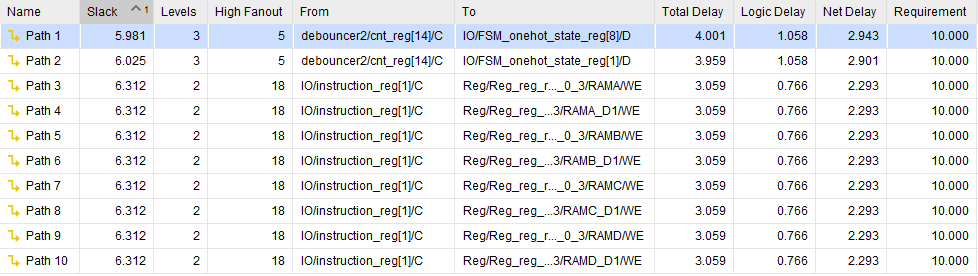
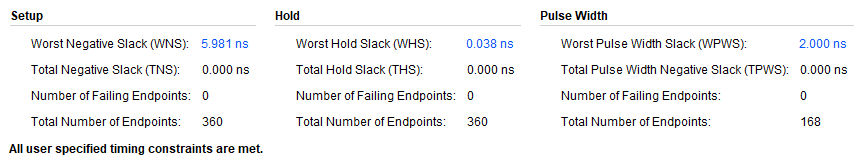
**-Power, Utilization**

****

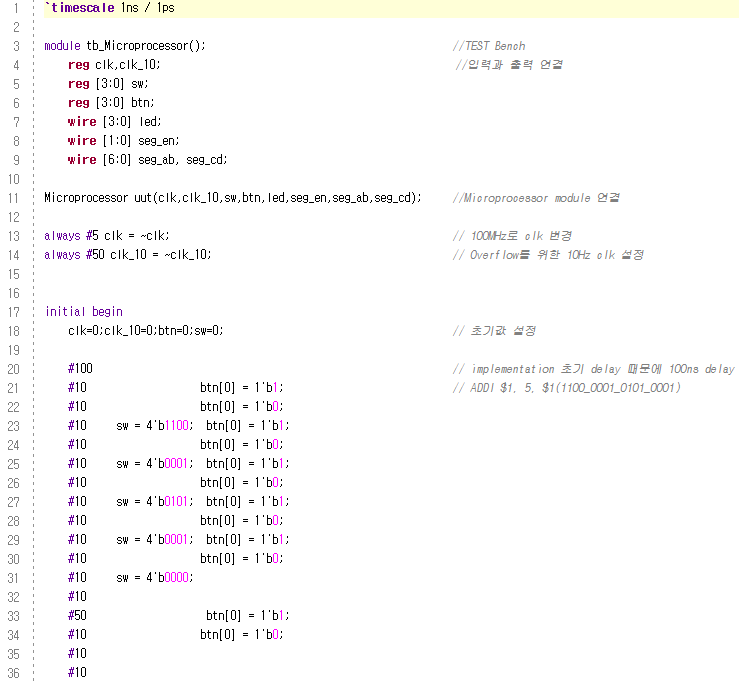
****

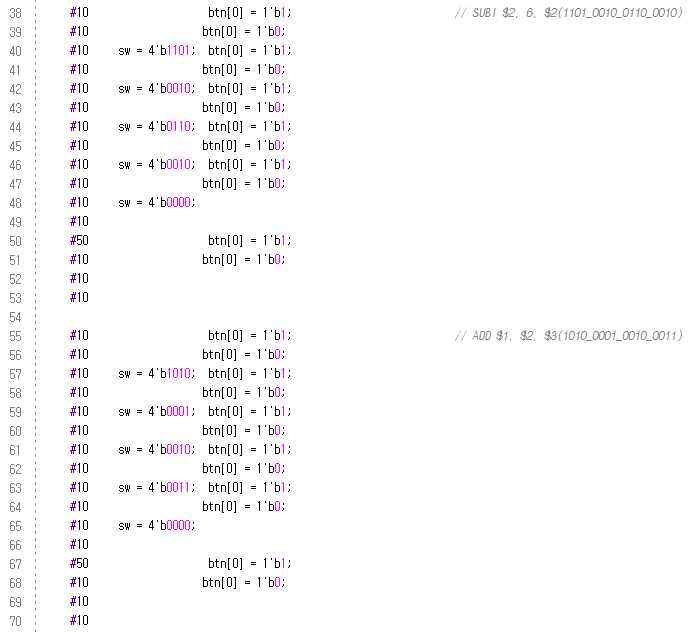
****

**-Delay**

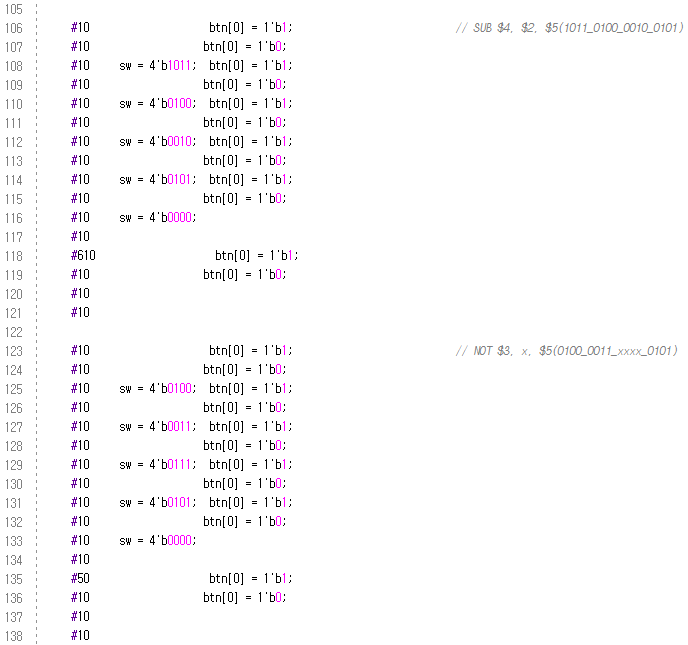
****

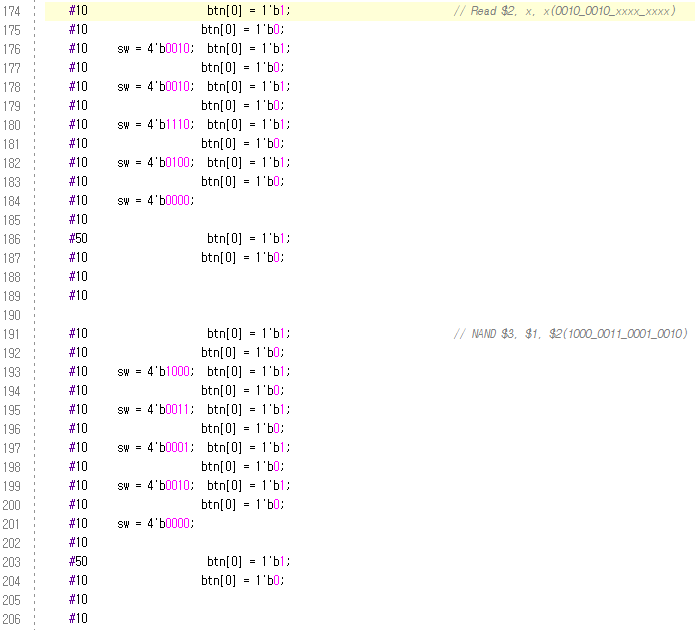
**Test Bench Code**

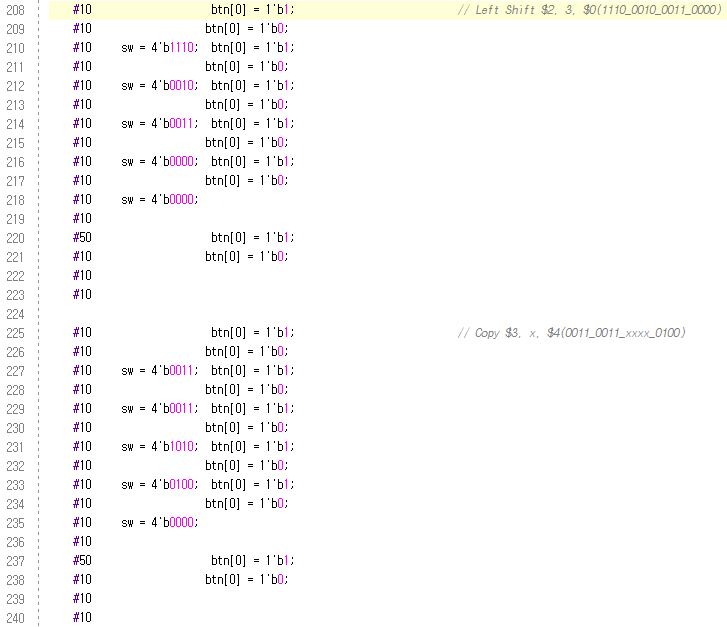


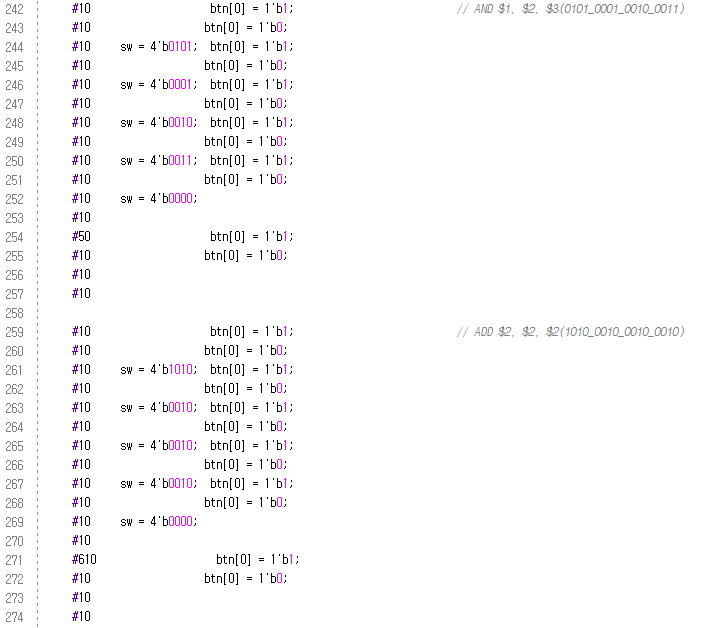


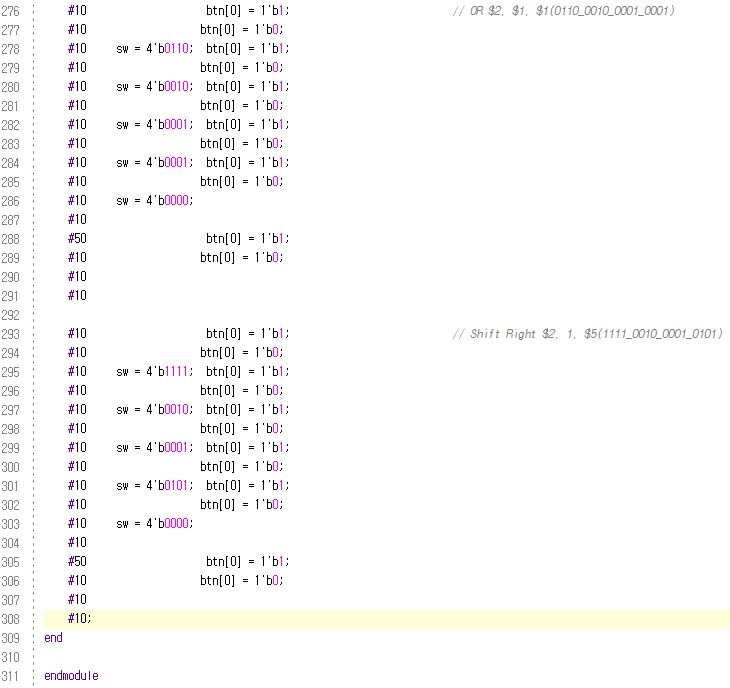




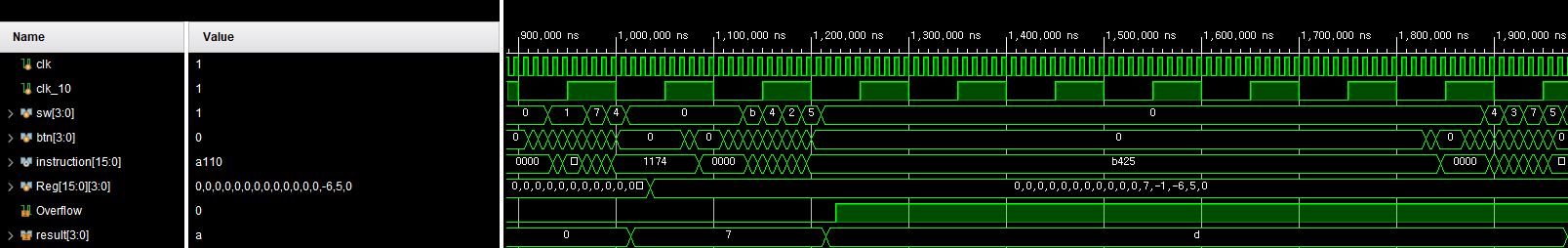


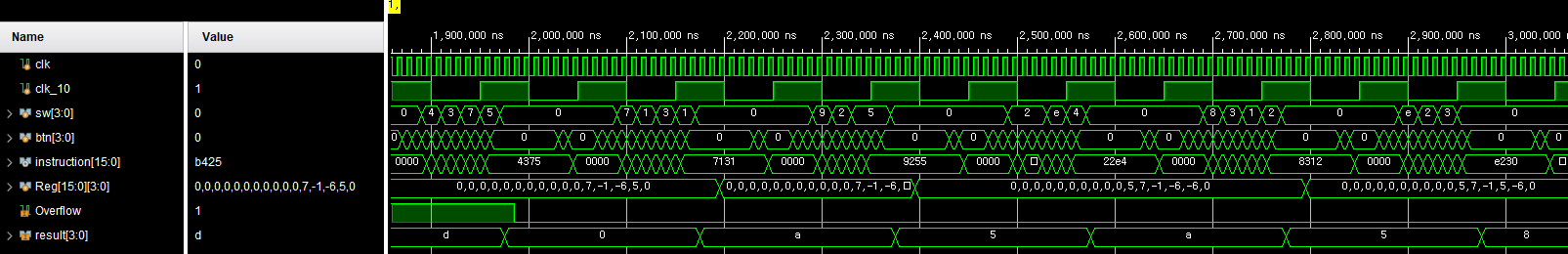


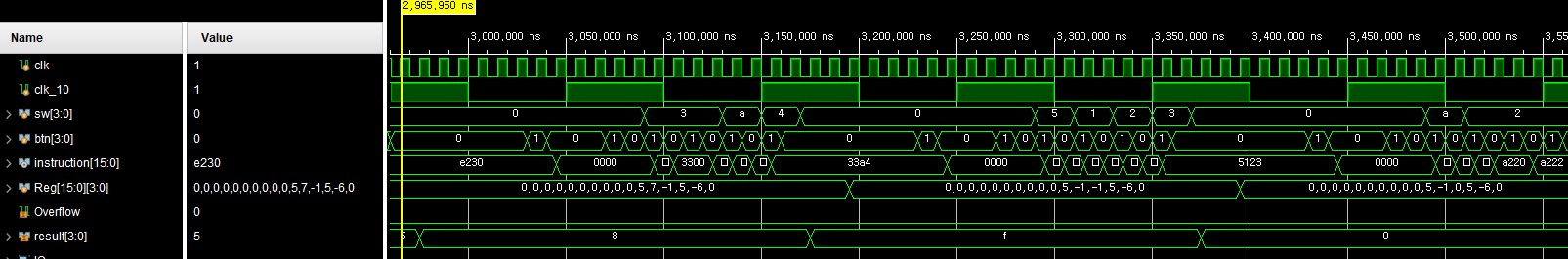


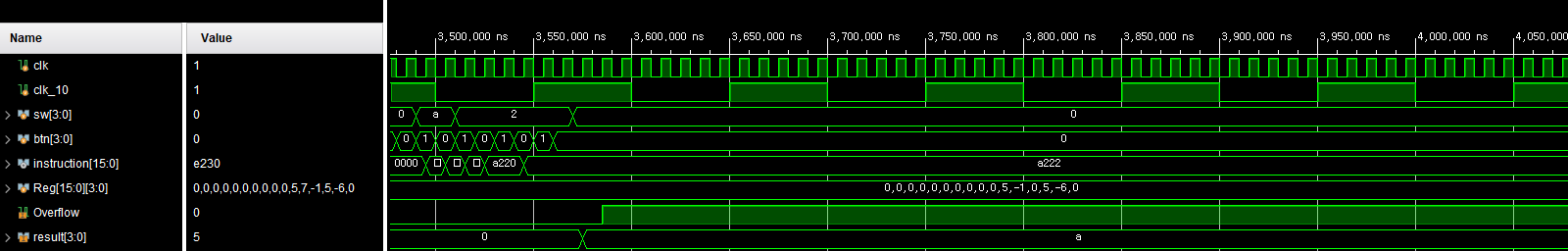


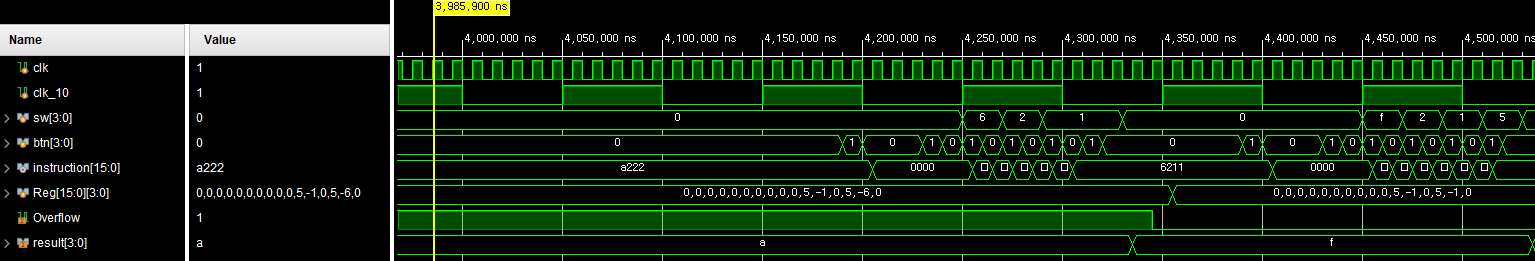
**Simulation Result**

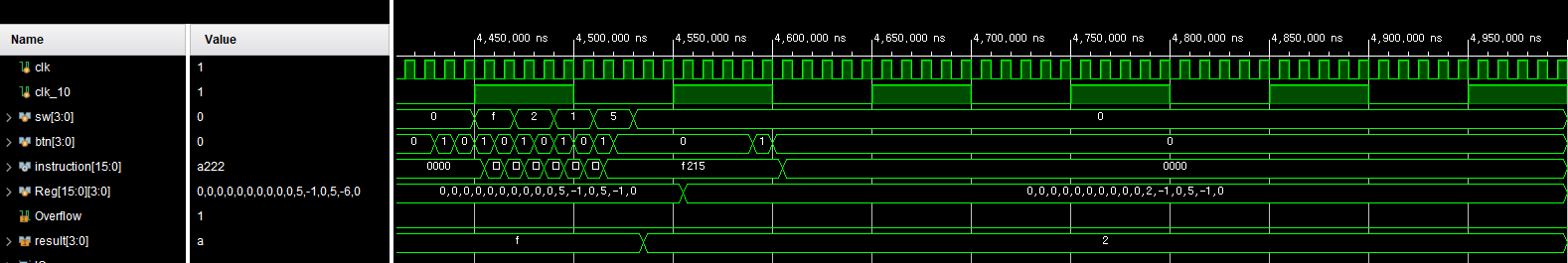












|  |
| --- |
| **Discussion** |
| **. - Verilog Coding을 시작하기 전 작성한 Block Diagram**    -Block Diagram -SM Chart    -State Machine  **- 작성한 Verilog Module에 대한 설명**    <clk\_wiz0>  먼저 clk\_wiz0으로 FPGA 보드 내부 clock인 125MHz를 clk\_100M 즉 100MHz의 clock을 얻어 각 module의 내부 clock으로 사용한다.  IO의 경우 sevensegment와 LED를 clock에 따라 번갈아 on 시키기 때문에 frequency divider를 사용하여 100MHz의 clock과 1MHz, 20Hz의 clock을 만들어 IO module에 입력한다.  FPGA 보드에서 입력하는 신호인 btn은 switch처럼 계속 해서 low거나 high이지 않고 누르는 동안만 High가 되기 때문에 외부에 대한 입력인 btn을 내부 clock인 100MHz에 맞추어 synchronizer module로 동기화 시킨다.  또한 noise를 제거하기 위해 Debouncer module로 debounce 시킨다.  <IO>    IO module에서는 입력된 switch 값과 btn 값을 바탕으로 명령어 instruction과 각 모듈의 시작신호를 생성하고 이를 각 module에 입력한다.  또한 control, ALU, Register에서 수행하는 동작의 시행 시점을 IO에서 관리, 결정한다.  현재 입력된 명령어를 Seven segment를 통해 보여주고 현재 state를 LED를 통해 출력한다. 또한 계산이 완료된 단계에서는 계산 결과를 Seven segment를 통해 출력하고 계산이 끝난 상태일 때 0번 버튼을 누르면 해당 계산을 수행한 명령어를 보여준다.  계산 결과가 Overflow라면 LED를 0.2s마다 0번, 1번 LED와, 2번, 3번 lED를 번갈아가며 on 시키도록 한다.    <Control>    control module에서는 IO에서 생성된 instruction중 상위 4bit opcode를 입력 받아 해당 명령어에 맞는 연산 동작을 하도록 ALU 연산 동작을 정한다.  또한 Register에 값을 저장할 수 있게 하는 enable 신호와 Register에서 연산의 입력을 register 안에 저장된 값과 상수를 선택하도록 하는 MUX select 신호를 출력하여 Register에 입력하도록 한다.  <Reg>    Register module에서는 IO에서 생성한 명령어 중 하위 12bit를 입력받아 ALU의 연산을 하도록 하는 연산 입력을 결정하고 연산 결과를 register file에 저장하도록 한다.  연산의 입력은 instruction의 11~8bit에 있는 주소에 저장된 레지스터 값인 연산 입력1(Reg\_out1)과 control에서 Register로 연결된 연산 입력 2 선택 신호(ALU\_Src)를 통해 instruction의 7~4bit에 있는 주소에 저장된 레지스터 값과 상수 instruction의 7~4bit 중 하나를 선택하여 연산 입력2 (Reg\_out2)로 출력한다.  IO에서 레지스터 저장 시작 신호(Reg\_Store)가 High로 입력되면 Control에서 입력된 Register 쓰기 enable 신호(Reg\_Write)를 판단하여 Register에 ALU 연산 결과를 저장한다. Register 결과가 Overflow라면 Register에 저장하지 않는다.  <ALU>    ALU module에서는 IO에서 연산 시작 신호(Execute\_St)가 입력되면 Register에서 입력된 연산 입력1과 연산 입력2를 Control에서 입력된 ALU 연산 동작에 따라 해당 연산을 수행해 연산 결과를 IO와 Register로 출력한다.  Overflow가 발생할 수 있는 ADD(ADDI)와 SUB(SUBI) 연산일 경우 adder\_subtractor에 연결하여 연산 결과와 Overflow가 발생하였는지를 확인한다.  Overflow가 발생하였는지 확인하는 신호인 Overflow\_St가 입력되면 Overflow를 확인하고 overflow 결과를 IO에 전달한다.  adder\_subtractor에서는 ALU 연산이 ADD(ADDI)와 SUB(SUBI) 연산일 경우 해당 연산을 수행한다.  ADD의 경우 Full Adder를 통해 두 수를 더하고 SUB의 경우 피감수(연산 입력 2)를 2의 보수를 취한 다음 두 수를 더한다. 결국 양수+양수, 양수+음수, 음수+양수, 음수+음수의 연산이 된다.  여기서 Overflow는 같은 부호의 숫자를 더했을 경우 발생할 수 있다.  같은 부호의 수를 더한 뒤 그 결과가 부호가 다르다면 Overflow가 발생한 것을 확인할 수 있고 이 경우 Overflow가 발생하였다고 판단해 ALU에 Overflow 결과를 출력한다.    Overflow를 감지하는 방법은 Carry와 Cout을 비교하는 방법을 사용하였다.  4bit의 adder에서 최상단의 bit는 부호를 나타내는 bit이므로 2의 보수법에 따르면 4bit에서 표현할 수 있는 수의 범위는 -8~7이다.  즉 이 수를 벗어나는 4bit의 sum이 발생하게 된다면 overflow로 판별할 수 있다.  이 수를 벗어나는 경우를 생각해보면 두가지로 나눌 수 있다.   1. 양수와 양수의 합이 8이상이 되는 경우 (1xxx인 경우 8 이상이라고 했을 때)   0~7의 양수는 네번째 bit가 0이므로 4번째 bit에서 1+1인 경우가 없으므로 어떠한 경우에도 cout은 발생하지 않는다.  반면 overflow인 경우에 네번째 bit가 1이 되어 음수를 표현하게 된다.  이 경우 overflow로 detect된다.  네번째 bit가 1이 되기 위해서는 c[2]이 발생하여 부호 bit를 1로 채워야 한다.   1. 음수와 음수의 합이 양수가 되는 경우   0~-8의 음수는 네번째 bit가 1이므로 cout이 항상 발생하게 된다.  Overflow인 경우 네번째 bit가 0으로 양수가 되어야 한다.  이를 만족할 수 있는 방법은 c[2]가 발생하지 않아 부호 bit가 0이 되는 경우이다.  위 경우를 모두 정리하면 cout과 c[2]이 같은 값을 가지지 않는 경우 overflow가 발생한다.  예외의 경우를 생각해보면 양수+양수에서 cout은 항상 발생하지 않고 overflow가 아니라면 c[3]가 발생하지 않아 결과값이 항상 양수가 되도록 하므로 양수+양수에서는 예외의 경우가 없다.  음수+음수에서 cout은 항상 발생하고 overflow가 아니라면 c[3]는 항상 발생해 결과값이 항상 음수가 되도록 하므로 음수+음수의 경우에서 예외의 경우는 없다.  <synchronizer>  synchronizer module에서는 입력된 버튼 신호가 클럭에 동기화 되도록 하였다.  <debouncer>  debouncer module에서는 노이즈를 제거하고 한 클럭에서만 클럭이 생성되도록 하였다.  btn[3], btn[2], btn[0]은 한 클럭만 사용하는 버튼이고 btn[1]은 계속해서 결과를 출력해야하는 버튼이기 때문에 btn[3], btn[2], btn[0]는 debouncer를 사용하였고 btn[1]는 debouncer를 사용하지 않았다.  **- 작성한 Code가 어떤 동작을 하는지? 왜 그런 동작을 하는지?**  모듈별 입**·**출력신호, 동작 설명  <Microprocessor>  sw : 스위치 신호  btn : 버튼 신호  led : LED 신호  seg\_en : seven segment enable 신호  seg\_ab : seven segment A, B data 신호  seg\_cd : seven segment C, D data 신호  Microprocessor에서는 FPGA 보드를 통해 입력 신호가 들어오고 이를 synchronizer, debouncer와 연결하여 내부 clock에 동기화, noise 제거를 실행한다.  또한 내부 clock은 frequency divider를 통해 clk가 변환되어 각 module의 제어 clock으로 사용된다.  <synchronizer>  D flip flop으로 작동하여 입력 신호를 내부 clock과 동기화 시켜주는 역할을 하도록 한다.  <debouncer>  내부 clock의 edge마다 high일 경우 count를 하여 미리 저장된 값을 넘길 경우 noise가 아닌 real input으로 판별해 출력하여 noise를 제거한다.  또한 assign 문을 사용해 조건 연산자로 count가 N일 경우에만 출력을 1로 설정하기 때문에 연속적으로 버튼을 눌러 1을 입력하게 되어도 1 clock의 신호만 출력되어 단일 신호가 된다.  <IO>  clk\_10 : overflow 시 LED 점멸을 위한 clock  Overflow : 연산 결과가 overflow인지 나타내는 신호  sw : 스위치 신호  btn : 버튼 신호  result : 연산 결과  seg\_en : seven segment enable 신호  seg\_ab : seven segment A, B data 신호  seg\_cd : seven segment C, D data 신호  Execute\_St : ALU 연산 시작 신호  Overflow\_St : Overflow 확인 시작 신호  Reg\_Store : Register에 ALU 연산 결과 저장 시작 신호  instruction : 입력받은 명령어를 출력 해주는 신호  IO state 설명  Idle : LED, Segment 표시 값, 명령어 초기화  Instruction1 : 명령어의 상위 4비트 설정, LED 변경  Instruction2 : 명령어 [11:8] index에 값 설정, LED 변경  Instruction3 : 명령어 [7:4] index에 값 설정, LED 변경  Instruction4 : 명령어의 하위 4비트 설정, LED 변경  Execute1 : ALU 연산 시작 신호 출력  Execute2 : Overflow 확인 신호 출력  Execute3 : Register에 ALU 연산 결과 저장 신호 출력  Done : 결과 출력  IDLE 상태에서는 LED를 모두 off 시키고 seven segment를 0000으로 초기화 한 뒤 명령어를 0으로 초기화를 한다. 현재 명령어를 출력하지 않은 상태를 IDLE로 설정하였다.  0번 버튼을 누르면 clk\_100M에 맞춰 다음 상태가 instruction1로 설정된다.  나머지 경우 (1번, 2번,3번 버튼)에는 현재 상태인 IDLE을 계속 반복하도록 설계하였다.    Instruction1 상태에서는 총 16bit의 명령어 중 상위 4 bit, 즉, 15~12 bit를 설정하는 단계이다.  3번 LED를 on시켜 현재의 상태를 나타낸다.  4 bit의 switch 신호를 High 상태 = 1, Low 상태 = 0으로 표시하여 명령어 15~12 bit를 입력받는다.  seven segment는 계속해서 피드백하여 1번 seven segment에 표시한다.  0번 버튼을 누르면 다음 단계인 Instruction2 상태로 이동하고 3번 버튼을 입력하면 초기 상태인 IDLE 상태로 돌아가도록 한다.  나머지의 경우에는 현재 상태를 반복한다.    Instruction2 상태에서는 총 16bit의 명령어 중 11~8 bit를 설정하는 단계이다.  2번 LED를 on시켜 현재의 상태를 나타낸다.  4 bit의 switch 신호를 High 상태 = 1, Low 상태 = 0으로 표시하여 명령어 11~8 bit를 입력받는다.  seven segment는 계속해서 피드백하여 2번 seven segment에 표시한다.  0번 버튼을 누르면 다음 단계인 Instruction3 상태로 이동하고 3번 버튼을 입력하면 초기 상태인 IDLE 상태로 돌아가도록 한다.  나머지의 경우에는 현재 상태를 반복한다.    Instruction3 상태에서는 총 16bit의 명령어 중 7~4 bit를 설정하는 단계이다.  1번 LED를 on시켜 현재의 상태를 나타낸다.  4 bit의 switch 신호를 High 상태 = 1, Low 상태 = 0으로 표시하여 명령어 7~4 bit를 입력받는다.  seven segment는 계속해서 피드백하여 3번 seven segment에 표시한다.  0번 버튼을 누르면 다음 단계인 Instruction4 상태로 이동하고 3번 버튼을 입력하면 초기 상태인 IDLE 상태로 돌아가도록 한다.  나머지의 경우에는 현재 상태를 반복한다.    Instruction4 상태에서는 총 16bit의 명령어 중 3~0 bit를 설정하는 단계이다.  0번 LED를 on시켜 현재의 상태를 나타낸다.  4 bit의 switch 신호를 High 상태 = 1, Low 상태 = 0으로 표시하여 명령어 3~0 bit를 입력받는다.  seven segment는 계속해서 피드백하여 4번 seven segment에 표시한다.  0번 버튼을 누르면 다음 단계인 Excute1 상태로 이동하고 3번 버튼을 입력하면 초기 상태인 IDLE 상태로 돌아가도록 한다.  나머지의 경우에는 현재 상태를 반복한다.    Execute1 상태에서는 ALU 연산 시작 신호인 Execute\_St를 출력한다. 이 상태에서는 한 클럭 뒤에 다음 상태는 Excute2가 되게 된다.    Excute2 상태에서는 Execute\_St를 0으로 만들고 Overflow가 발생하였는지를 확인하기 위해 Overflow\_St 신호를 1로 만든다.  이 상태에서는 한 클럭 뒤에 다음 상태는 Excute3이 되게 된다.    Excute3 상태에서는 Overflow\_St를 0으로 초기화하고 ALU 연산 결과를 저장하기 위해 Reg\_Store을 1로 만든다.  이 상태에서는 한 클럭 뒤에 다음 상태는 Excute3이 되게 된다.    Done 상태에서는 모든 연산이 끝난 상태를 의미한다.  seven segment로 연산 결과를 출력한다.  만약 over flow가 발생한다면 clk\_20 에 맞춰 0,1번 LED와 2,3번 LED가 번갈아가며 on이 되게 된다.  Done 상태에서 1번 버튼을 누르게 되면 seven segment에 연산 결과가 아닌 해당 결과를 내는 명령어 instruction을 4bit씩 나누어 16진수로 표현한다.  0번, 3번 버튼을 누르게 되면 IDLE을 다음 상태로 설정하고 나머지의 경우 현재 상태 Done을 유지한다.      <Control>  opcode : IO에서 출력된 instruction 중 15~12 bit  ALU\_Src : Register에서 ALU 연산 입력2 선택 신호  Reg\_Write : Register에 연산 결과 저장 enable 신호  ALU\_op : 연산 종류 선택 신호  control에서는 IO에서 출력된 명령어 instruction 중 상위 4bit를 입력으로 받아 명령어를 해석해 ALU 연산 동작, Register 쓰기 enable, Register의 연산 입력 결정 신호를 내보낸다.  opcode를 parameter로 각각 의미하는 연산을 설정하였다.  clk\_100M에 동기화 되어 opcode를 해석한 뒤 opcode에 따라 해당 ALU 동작(ALU\_op), Register 쓰기 enable(Reg\_Write), 연산 입력 결정 신호(ALU\_Src)를 출력한다.  <Register>  ALU\_Src : Register에서 ALU 연산 입력2 선택 신호  Reg\_Write : Register에 연산 결과 저장 enable 신호  Reg\_Store : Register에 ALU 연산 결과 저장 시작 신호  Overflow : 연산 결과가 overflow인지 나타내는 신호  Write\_data : Register에 저장할 ALU 결과  instruction : IO에서 입력 받은 명령어의 11~0 bit  Reg\_out1 : 연산 입력 1  Reg\_out2 : MUX에 의해 선택된 연산 입력 2  초기 설정으로 16개의 4 bit register를 모두 0으로 초기화한다.  clk\_100M에 동기화되어 연산 입력1인 Reg\_out 1에는 instruction의 11~8bit의 주소에 해당하는 register에 저장된 값을 출력하도록 한다.  연산 입력2인 Reg\_out2는 연산 입력2 결정 신호(ALU\_Src)에 따라 0일 경우 instruction의 7~4 bit의 주소에 해당하는 register에 저장된 값으로, 1일 경우 instruction의 7~4 bit의 값으로 설정한다.  또한 Register 저장 시작 신호(Reg\_Store)와 Register 쓰기 enable 신호(Reg\_Write)를 and 연산하여 둘 다 1일 경우에만 연산 결과를 저장한다.  연산 결과를 저장할 때에도 Overflow가 발생하지 않은 경우와 instruction 3~0 bit가 0의 값을 가지고 있지 않을 경우 연산 결과를 저장하고 Overflow가 발생하였거나 0번 레지스터에 값을 저장하라는 명령어라면 값을 저장하지 않는다.  <ALU>  Execute\_St : ALU 연산 시작 신호  Overflow\_St : Overflow 확인 시작 신호  ALU\_op : 연산 종류 선택 신호  Reg\_out1 : 연산 입력 1  Reg\_out2 : MUX에 의해 선택된 연산 입력 2  result : 연산 결과  Overflow : 연산 결과가 overflow인지 나타내는 신호  Control에서 출력한 ALU 연산 종류 신호(ALU\_op)를 입력받아 연산 시작 신호(Execute\_St)가 1이 되면 연산을 수행한다.  NOP, Write, Read, AND, OR, XOR, NAND, NOR의 경우 연산 입력1과 연산 입력2를 적절히 조작해 해당하는 연산을 수행한다.  Overflow가 발생할 수 있는 ADD, ADDI, SUB, SUBI의 경우 adder\_subtractor에 연산 입력을 연결한다.  ADD, ADDI의 경우 연산 입력 1과 연산 입력2를 그대로 연결하고 SUB, SUBI의 경우 연산 입력2에 반전을 취하여 입력한다.  SUB,SUBI의 경우 Add\_Sub가 1이 되고 연산 입력2와 1bit씩 XOR 연산을 수행해 반전을 취하게 된다.  또한 Adder\_Subtractor에서 Cin에 1이 입력되어 최종적으로 뺄셈은 연산 입력2에 2의 보수를 취한 후 덧셈을 수행하도록 설계하였다.  앞서 설명했듯이 덧셈 연산 중 carry와 Cout을 비교하여 overflow를 감지하는 것을 코드로 구현하여 계산된 Overflow를 IO로 연결한다.  Shift 연산의 경우 연산 입력 2에 1이 입력된 경우, 2가 입력된 경우, 3이 입력된 경우, 나머지의 경우로 나누어 shift를 진행하였다.  Shift left의 경우 1이 입력되었을 때 연산 입력1의 2~0 bit를 3~1 bit 로 만들고 0 bit 에는 0을 채우고, 2가 입력되었을 때는 연산 입력 1의 1~0 bit 를 3~2 bit 로 만들고 1~0 bit 에는 0을 채우고, 3이 입력되었을 때는 연산 입력 1의 0 bit를 3 bit로 만들고 2~0 bit에는 0을 채우고 나머지 경우 연산 결과를 모두 0으로 만듬으로써 Shirt Left를 구현하였다.  위와 같은 방법으로 Shift Right 또한 구현하였다.  - 작성한 Test Bench Code가 어떤 동작을 하는지?  Testbench에서는 clk\_Wiz0의 clock 생성 시점을 원하는 시간에 할 수 없기 때문에 5ns마다 clk을 반전시키도록 하여 100MHz의 clk을 생성하였고 50ns마다 clock을 반전시키도록하여 10MHz를 생성하였다.  overflow를 위한 clk\_10의 경우 testbench의 time scale을 생각해 보았을 때 원활하게 동작하는지 관측이 쉽지 않아 관측을 위해 clock의 주파수를 변경하였다.  초기 switch값과 btn값을 0으로 초기화 시켜주었다.  이후 입력한 명령어와 예상된 결과는 다음과 같다.   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 명령어 | 기계어 | 출력 값 | Register(결과) | | | | | | | $0 | $1 | $2 | $3 | $4 | $5 | | ADDI $1, 5, $1 | 1100\_0001\_0101\_0001 | 0101(5) | 0 | 5 | 0 | 0 | 0 | 0 | | SUBI $2, 6, $2 | 1101\_0010\_0110\_0010 | 1010(-6) | 0 | 5 | -6 | 0 | 0 | 0 | | ADD $1, $2, $3 | 1010\_0001\_0010\_0011 | 1111(-1) | 0 | 5 | -6 | -1 | 0 | 0 | | NOP x x x | 0000\_xxxx\_xxxx\_xxxx | 0000(0) | 0 | 5 | -6 | -1 | 0 | 0 | | Write x, 7, $4 | 0001\_xxxx\_0111\_0100 | 0111(7) | 0 | 5 | -6 | -1 | 7 | 0 | | SUB $4, $2, $5 | 1011\_0100\_0010\_0101 | Overflow | 0 | 5 | -6 | -1 | 7 | 0 | | NOT $3, x, $5 | 0100\_0011\_xxxx\_0101 | 0000(0) | 0 | 5 | -6 | -1 | 7 | 0 | | XOR $1, $3, $1 | 0111\_0001\_0011\_0001 | 1010(-6) | 0 | -6 | -6 | -1 | 7 | 0 | | NOR $2, $5, $5 | 1001\_0010\_0101\_0101 | 0101(5) | 0 | -6 | -6 | -1 | 7 | 5 | | Read $2, x, x | 0010\_0010\_xxxx\_xxxx | 1010(-6) | 0 | -6 | -6 | -1 | 7 | 5 | | NAND $3, $1, $2 | 1000\_0011\_0001\_0010 | 0101(5) | 0 | -6 | 5 | -1 | 7 | 5 | | Left Shift $2, 3, $0 | 1110\_0010\_0011\_0000 | 1000(-8) | 0 | -6 | 5 | -1 | 7 | 5 | | Copy $3, x, $4 | 0011\_0011\_xxxx\_0100 | 1111(-1) | 0 | -6 | 5 | -1 | -1 | 5 | | AND $1, $2, $3 | 0101\_0001\_0010\_0011 | 0000(0) | 0 | -6 | 5 | 0 | -1 | 5 | | ADD $2, $2, $2 | 1010\_0010\_0010\_0010 | overflow | 0 | -6 | 5 | 0 | -1 | 5 | | OR $2, $1, $1 | 0110\_0010\_0001\_0001 | 1111(-1) | 0 | -1 | 5 | 0 | -1 | 5 | | Shift Right $2, 1, $5 | 1111\_0010\_0001\_0101 | 0010(2) | 0 | -1 | 5 | 0 | -1 | 2 |   **- Simulation 파형이 왜 그렇게 나온것인지?**  위 표를 이용하여 시뮬레이션이 정상적으로 동작하는지 확인하였다.  앞서 Simulation Result에서 작성한 그림을 비교해보면 모두 정상적으로 동작하는 것을 확인할 수 있다.  또한 Reg 모듈의 Reg에도 동일한 값이 저장되어있는 것을 확인할 수 있었다.  하지만 시뮬레이션 결과는 delay에 의해 실제 implementation을 진행한 시뮬레이션에서 일치하지 않을 수 있다.  실제 딜레이에 의한 분석은 이후 한번에 진행하였다.  **- 작성한 Code가 잘 동작하지 않는다면 어디서 문제가 발생한 것인지?**  처음 코드를 작성한 방식은 각각의 모듈을 clk과 상관 없이 동작하도록 Data path로만 구성하는 것이었다.  RISC 구조의 MIPs에서 clk는 각 module이 동작하는 시간을 이용하여 모든 동작이 정상적으로 동작할 수 있는 clk를 설정하면서 구성한다.  이러한 방식을 동일하게 코드에 구성하였다.  이런 방식을 사용하면서 RTL Schematic에서는 입력한 코드와 동일하게 module이 구성된 반면에 Synthesis와 implementation을 진행한 Schematic은 clk가 들어있지 않은 모듈이 모두 사라졌다.  일부 모듈이 다른 모듈과 통합되어 나타났지만 이러한 결과는 원하는 결과가 아니었다.  또다른 문제점은 clk를 내 마음대로 설정할 수 없어 장점을 극대화할 수 없다는 것이었다.  clk는 최대 100MHz로 제한되어 있어 더 빠른 Hz를 사용할 수 없었다.  따라서 이러한 장점을 사용할 수 없다면 clk를 모든 module과 연결하여 안정성을 높이는 것이 더 낫다고 판단하였다.  clk를 모든 module과 연결하면 각 module에서의 동작 타이밍을 쉽게 분석할 수 있었으며 각종 Control 신호의 timing에서 발생하는 문제에 대해서도 자유로웠다.  **- 딜레이에 대한 분석**  delay는 실제 보드위에 코드를 올렸을 때 단순히 실행이 늦어지는 것 뿐만 아니라 제대로된 동작을 할 수 없도록 한다.  따라서 delay에 대한 분석을 통해 실제 보드위에서 정상적으로 동작하고 있는지 확인하였다.  implementation은 실제 보드를 바탕으로 delay가 생성된다.  따라서 implementation의 timing simulation을 통해 delay를 확인해볼 수 있었다.    먼저 report timing summary 메뉴를 통해 분석한 delay는 위와 같다.  이 report는 전체적으로 최악의 경우의 delay에 의한 timing을 분석해주는 지표이다.  setup time의 slack은 5.981ns로 나타났으며 Hold time의 slack은 0.038ns Pulse Width slack은 2ns로 나타났다.    위 그림은 setup time의 slack을 나타낸 것이다.  slack이란 쉽게 말해 여유 시간을 의미한다.  즉 delay를 감안하였을 때 slack이 존재한다는 것은 setup time에 위배되지 않는다는 것을 의미한다.  또 만약 Total negative slack이 0으로 나타나는 것을 통해 negative slack 즉 setup time이 위배되지 않았다는 것을 의미한다.  아래 그림은 setup time slack의 하위 10개의 path를 보여주는 그림이다.  debouncer에서 IO, IO에서 reg사이에서 가장 큰 delay를 보였다.    Hold time 또한 문제를 일으키지 않을 정도로 발생한 것을 확인할 수 있었다.  0.038ns 의 slack이 있었다.  아래 그림은 hold time slack의 하위 10개의 path를 나타낸 그림이다.    IO에서 reg, ALU에서 Reg 사이에서 가장 작은 slack을 보였다.  pulse width slack 은 2ns로 나타났는데 이는 0.5 duty rate을 가지는 신호를 인가했기 때문이다.  따라서 4ns(125MHz)의 절반인 2ns로 나타난 것이다.  다음으로는 delay를 고려한 simulation result와 timing을 분석해보았다.  아래 그림은 Microprocessor에서 IO까지의 instruction에서 delay값을 나타낸 것이다.      instruction이 전달되는 시간을 기준으로 140.431ns-135ns=5.431ns의 delay가 발생함을 알 수 있었다.  다음은 Microprocessor에서 IO의 Excute\_St까지의 delay를 계산해 보았다.      210.792ns-205ns=5.792ns로 나타났다.  아래 그림은 Overflow\_St의 delay를 나타낸 그림이다.      230.432ns-225ns=5.432ns의 delay가 발생하였다.  Reg\_Store는 아래 그림과 같다.      240.395ns-235ns=5.395ns의 delay가 나타났다.  아래 그림은 Control에서 delay를 나타낸 것이다.        Reg\_Write는150.394ns-145ns=5.394ns의 delay가 나타났고 ALU\_Src는 5.431ns의 delay가 나타났다.  아래 그림은 ALU의 delay를 나타낸 것이다.      230.430ns-225ns=5.430ns의 delay가 나왔다.  아래 그림은 Reg에서의 delay를 확인한 그림이다.      251.319ns-245ns=6.319ns의 delay가 확인되었다.  위 분석을 통해 Reg에서 delay가 가장 많이 발생하는 것을 확인할 수 있었다.  이는 Reg 모듈에서의 동작이 IO에서 instruction을 받아오고 이 값을 이용해 Register에서 값을 받아오는 방식이기 때문이다.  물리적으로 instruction이 IO에서 출발하는 delay인 5.531ns에 Reg에 instruction이 전해진다.  이 값을 이용해 Register 값을 가져오기 때문에 긴 delay 값을 보이는 것으로 생각된다.  위의 전체적인 분석에서 report보다 많은 delay를 보이는 것을 확인할 수 있었다.  이는 clk에서 출력 timing이 얼마나 떨어져 있는지를 확인하는 방식으로 계산하였기 때문으로 생각된다.  구성한 Simulation result에서 clk는 입력 값으로 바로 입력되는 것으로 나타났다.  하지만 실제 회로에서는 reference clk를 만들어주는 모듈에서 구성한 회로까지 clk가 전달되는 과정에서 delay가 발생할 것이다.  따라서 clk가 바로 입력되는 것이 아닌 delay 값을 가질 것이고 이 delay된 clk에서 출력까지의 delay를 계산하는 방식으로 report가 나타나는 것으로 생각된다.  **-HW Debug**  아래 그림은 btn[0]를 trigger하여 얻은 result이다.    HW Debug 특성상 매우 많은 시간이 소요되어 다른 분석은 진행하지 못하였지만 버튼에 의해 instruction이 제대로 변경되는 것은 확인할 수 있었다.  또한 led도 상태에 따라 동일하게 출력되는 것을 확인할 수 있었다.  **-HW 분석**    전체 HW    위 그림은 HW를 자세히 살펴본 그림이다.  왼쪽 큰 네모는 clk를 생성해내는 IP에 의해 나타나는 HW로 정상적으로 신호를 출력하고 있는 것을 확인할 수 있다.  오른쪽 그림은 나머지 회로를 나타내는 그림이다.  Zybo z7-10보드는 다양한 LUT와 FDRE, MUX, XOR로 구성되어 있음을 확인할 수 있었다.  간단하게 설명하면 LUT는 알맞은 정보를 저장하고 있고 FDRE는 Flip-Flop을 나타낸다.  MUX와 XOR은 코드를 연결하며 원하는 동작을 할 수 있도록 연결된다.  이러한 회로의 연결에서 물리적인 거리가 발생하게 되고 이러한 거리는 delay를 발생시키는 원인이 된다.  앞서 delay에 대한 분석을 진행하였는데 이러한 delay는 위와 같은 원인에 의해 발생한 것이다.  HW 분석을 통해 주어진 코드에 따라 이 회로를 최적화하는 것을 확인할 수 있었다.  Zybo z7-20보드로 설정을 바꿔 IP를 재설정하고 implementation을 진행하였을 때 전혀 다른 HW 구성을 나타내는 것을 확인할 수 있었다.  사용한 모듈의 위치가 예상하지 못한 방향으로 바뀌어 있었다.  Vivado에서 보드 설정을 인지하여 최적화된 회로로 바꾸어 주는 것을 확인할 수 있었다.  따라서 보드에 따라서도 다른 delay값을 나타낼 수 있을 것이라고 생각했다.  전체 HW  delay  실제로 implementation을 진행한 결과 zybo z7-20이 더 나은 delay값을 보이는 것을 확인할 수 있었다.  **고찰**  현재 우리 조가 설계한 Microprocessor는 multi cycle로 코드가 동작한다. 명령어의 입력을 모두 받은 즉시 1)연산을 수행하고 2) overflow를 확인하고 3) register에 연산 결과를 저장하는 동작을 수행한다. 연산 속도를 더 빠르게 하기 위해서 개선해야 할 점이 많이 있다고 생각한다.  먼저 하드웨어의 크기를 고려하지 않고 소프트웨어의 크기만 고려한다면 ADD, ADDI, SUB, SUBI일 때만 overflow가 발생하고 16개의 명령어 중 오직 4개의 명령어 많이 overflow가 발생할 가능성이 있다. 다른 명령어들은 overflow가 발생하지 않더라도 overflow 확인 단계에서 명령 수행 종료까지 1 cycle을 낭비하게 된다. 따라서 control에서 명령어가 ADDI, ADD, SUBI, SUB 인 경우에만 추가적으로 overflow가 발생하였는지 확인한다면 평균 명령어당 수행 cycle이 줄 것이라 예상하고 이로 인해 명령어 수행 속도가 더욱 향상될 것이라 기대할 수 있다.  또한 명령어가 시행되는 데 필요한 최소 시간을 모른다는 것이다. 명령어가 수행되는데 필요한 최대한의 시간을 clk의 주기로 설정하면서 clk의 주기가 최소가 되는 방향으로 설계한다면 초당 더 많은 명령어가 수행될 수 있고 이에 따라 명령어 수행 속도가 더 증가할 것으로 기대된다.  현재 명령어가 수행되는 cycle을 생각해 본다면 위에서 설명한 1)에서는 ALU에서만 동작이 이루어지고 2)역시 ALU에서 출력한 overflow 신호를 분석하는 것이다. 3)애서는 Register만 동작을 한다. 이를 pipeline으로 설계를 해 각각의 제어 신호가 모듈과 모듈 사이의 인터페이스에서 Flip Flop을 통해 저장되고 출력될 수 있다면 pipeline으로 설계할 수 있을 것이라 생각한다. pipeline으로 설계를 한다면 한 명령어에 대한 수행속도는 동일하지만 3 cycle동안 3개의 명령어를 각각의 단계에서 수행할 수 있어 초당 수행할 수 있는 명령어 개수가 증가할 것이라 기대된다.  Overflow 발생시 IO module에 입력된 clk\_20을 사용하였는데 Microprocessor에서 frequency divider로 clk를 낮추는 것이 아닌 IO block에서 100MHz를 count 하여 자체적으로 낮은 clk을 만들었다면 Microprocessor에서 사용frequency divider module을 사용하지 않았을 것이다. 전체 module 중 10Hz의 clk를 사용하는 module은 IO block 밖에 없지만 이를 위해서 Microprocessor에서 frequency divider를 4개나 사용할 필요가 없어진다.  또한 overflow역시 연산 결과를 판단하여 overflow가 발생하였는지를 확인하였는데 명령어 입력 단계에서 Register에서 값을 입력 받을 때 ALU에서 연산을 수행하기 전에 입력으로 Overflow를 확인할 수 있다면 명령어 당 2 cycle로 overflow를 확인하는 cycle이 없어지면서 수행 속도가 더욱 증가할 것이라 생각한다.  하지만 이러한 설계를 하게 되면 하드웨어가 무거워 질 것이고 그에 따라 delay가 증가할 것이라 예상된다. |