# **FINAL PROJECT. micro processor**

**Micro processor 및 설계하는 Micro processor에 대한 소개**

**[Microprocessor]**

: 마이크로프로세서는 복잡한 디지털 시스템의 예이다. 이는 컴퓨터의 가장 핵심적인 부분으로 Memory로부터 명령어와 data를 Fetch하고 Decode하여 특정 동작을 Execute한다. 정해진 명령어에 따라서 레지스터 연산, 산술 연산, 논리 연산 등을 수행하며, 명령어 조합에 따라서 원하는 계산 결과를 얻을 수 있다. 마이크로프로세서는 RISC로 알려진 유사한 철학으로 설계되었다. RISC 프로세서 출현 후 CISC라는 복잡 명령어 집합 컴퓨터가 있었지만, 주로 쓰이는 MIPS가 RISC 기반으로 설계되면서, 설계 단순화를 위한 RISC 기반의 설계가 dominant하게 되었다. MIPS는 파이프라인 구조를 사용하여 명령어의 처리를 최적화한다. 파이프라인은 여러 단계로 명령어를 분할하여 동시에 여러 명령어를 처리하는 방식이다. RISC 프로세서의 특징은 다음과 같다.

- **균일한 명령어 길이** : 32비트의 동일한 명령어 길이

- **적은 수의 명령어 형식** : 가능한 적은 수의 명령어 형식을 갖고 명령어 필드를 균일하게 인코딩하여 명령어 디코딩을 단순화

- **적은 수의 주소지정 방식** : 하나 또는 2개의 메모리 주소지정 방식을 지원한다. 이는 명령어가 접근해야 할 메모리 주소를 나타낼 수 있는 다른 방법을 제공한다.

- **많은 수의 레지스터** : RISC 철학은 메모리에 자주 접근하여 발생하는 성능저하 방지를 위해 많은 레지스터를 포함한다. 32bit 레지스터 32개를 사용하며, 이를 통한 레지스터-레지스터 구조를 실현한다.

- **로드/저장 구조** : 직접적으로 메모리 피연산자를 사용하는 산술 명령어가 없다. 이는 레지스터-레지스터 구조임을 의미한다.

- **암묵 피연산자 또는 부작용** : RISC 구조의 원리는 최소한의 암묵 피연산자/연산과 부작용을 가진다.

텍스트, 도표, 평면도, 개략도이(가) 표시된 사진

자동 생성된 설명

<32bit MIPS Microprocessor>

**[설계하는 Microprocessor 소개]**

도표, 텍스트, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

<Simple Microprocessor>

: 설계하는 마이크로프로세서는 MIPS 마이크로프로세서를 단순화한 모습이다. 각 블록의 역할은 다음과 같다.

- **IO Block** : 사용자 입력으로 명령어를 구성하여 전달한다. 명령이 입력은 스위치와 버튼을 통해 값을 입력 받아 명령어를 생성해 전달하며, 하나의 상태당 4bit씩의 명령어를 받는다. 현재의 상태는 LED를 통해 나타내며, 현재 상태 및 결과는 SSD를 통해 0~F(hex)를 출력한다.

- **Control Block** : 명령어의 Opcode를 디코딩해 ALU\_OP, ALU\_Src, Reg\_Write 신호를 출력한다. ALU\_OP는 ALU의 동작을 결정하고, ALU\_Src는 ALU의 입력을 결정하며, Reg\_Write는 레지스터에 쓸지에 대한 여부를 결정한다.

- **Register Block** : Data를 저장하고 Read, Write 동작을 수행한다.

- **ALU Block** : 제어 신호로 인해 결정된 두 개의 입력과 동작을 동해 결과값을 도출한다. add, sub 연산시에는 4bit data를 2의 보수로 취급한다.

MIPS의 PC 관련 block을 제외하고 생각하더라도, block의 수가 많이 줄었다. 메모리를 사용하지 않고 레지스터만을 사용하기 때문에 메모리에 대한 control signal과 memory block, mux가 없어져 단순화되었다. 위의 RISC 철학과 연결 지어 본다면, 균일한 명령어의 길이가 32->16bit로 줄었고 이에 따라 동작의 폭 또한 감소하였다. 기존 32bit MIPS에서의 명령어 포맷은 R, I, J로 총 3개의 포맷이 있었지만, Simple Microprocessor에서는 1개의 포맷으로 통일되었다. 명령어의 포맷은 아래와 같다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

<Simple Microprocessor Instruction Format>

메모리가 없기 때문에 레지스터에 저장하는 것만 수행하며, 레지스터는 32bit MIPS와 다르게 4bit 레지스터 16개로 구성 되어있다. 하나의 명령어만 받아 실행하며, 이로 인해 파이프라인 구조와는 거리가 있다.

**Verilog Code / 주석**

**[MPU]**

**텍스트, 스크린샷, 문서, 번호이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 폰트, 스크린샷, 문서이(가) 표시된 사진

자동 생성된 설명**

**[clk\_100M]**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[freq\_div\_100]**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[btn\_filter]**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[synchronizer]**

**텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

자동 생성된 설명**

**[debouncer]**

**텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

자동 생성된 설명**

**[IO]**

**텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 스크린샷, 문서이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[SSD]**

**텍스트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명**

**[CTRL]**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[Reg4x16]**

**텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명**

**[Mux\_4x2]**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[ALUU]**

**텍스트, 번호, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

****

**[Write]**

**텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[Read]**

**텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[Copy]**

**텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[NOT]**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[ANDD]**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[ORR]**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[XORR]**

**텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

자동 생성된 설명**

**[NANDD]**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[NORR]**

**텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[add]**

**텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

**[sCLA4]**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[CLL4]**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[GPFA]**

**텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명**

**[sub]**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**[comp\_2s]**

**텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명**

**[LSF]**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[RSF]**

텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

자동 생성된 설명

**Pin Mapping**

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

**Synthesis Report**

<timing>

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

<utilization>

텍스트, 스크린샷, 번호, 도표이(가) 표시된 사진

자동 생성된 설명

<power>

텍스트, 스크린샷, 소프트웨어, 번호이(가) 표시된 사진

자동 생성된 설명

<schematic>

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

**Implementation Report**

<timing>

텍스트, 폰트, 라인, 번호이(가) 표시된 사진

자동 생성된 설명

<utilization>

텍스트, 스크린샷, 번호, 도표이(가) 표시된 사진

자동 생성된 설명

<power>

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

<schematic>

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

**Elaborated Design**

도표, 텍스트, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

**Test Bench Code**

텍스트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

**Simulation Result**

스크린샷, 전자제품, 회로이(가) 표시된 사진

자동 생성된 설명

**HW Debugging**

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷, 라인, 다채로움이(가) 표시된 사진

자동 생성된 설명

<1. 10A5를 입력한 후 btn[0]을 눌렀을 때에 대한 디버깅 (명령어 입력4 -> Done)>

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷, 라인, 다채로움, 우주이(가) 표시된 사진

자동 생성된 설명

<2. 1에서 btn[0]을 눌렀을 때에 대한 디버깅 (Done -> Idle)>

|  |
| --- |
| **Discussion** |
| **- Verilog Coding을 시작하기 전 작성한 Block Diagram**    **- 작성한 Verilog Module에 대한 설명**  : IO block은 입출력 , Control block은 제어 신호 형성, Register는 데이터 저장, ALU는 연산하여 값 도출을 담당하여 동작한다. IO block의 입력으로는 스위치와 버튼, clk이 있고 출력으로는 LED와 SSD가 있다. btn은 주로 상태 천이에 사용되며, sw는 명령어 입력에 사용된다. 총 16비트의 명령어를 받아서 동작을 하게 되며, 명령어는 4비트씩 각각의 의미를 가진다. 명령어는 상태에 따라서 SSD에 출력된다. 명령어 [15:12]에서 이 명령어의 동작을 결정하여 control block에서 다른 block들의 제어 신호를 내보낸다. [11:8]은 Register에서의 주소를 나타내며 이는 ALU의 input이 된다. [7:4]는 명령어에 따라서 Register의 주소 혹은 데이터 값을 나타내며 이는 control block에서 생성된 제어 신호에 따른 Mux 출력에 의해 달라지게 된다. ALU를 통해 나온 값은 명령어에 따라서 Register에 Write 할지가 결정되고, 이 또한 명령어 [15:12]에 따라 생성된 제어 신호에 따라서 달라지게 된다. IO block의 출력을 통해 현재 상태를 led를 통해 출력하며, SSD를 통해 ALU의 결과값이 출력된다. SSD는 btn[1] 입력 여부에 따라서 현재 입력되어 있는 명령어도 출력으로 나온다.  위 Block Diagram을 보면 Register에 CTRL에서의 Reg\_Write 신호와 I/O단에서의 Reg\_Write 신호를 AND 연산한 값을 Register에 넣는 것을 볼 수 있다. 이는 불필요한 Write 동작을 막기 위한 연산으로, 자세한 내용은 아래 ‘고찰’에 기재하였다.  **- 작성한 Code가 어떤 동작을 하는지? 왜 그런 동작을 하는지?**  [MPU]  : 최상위 top 모듈이다. 125MHz 시스템 CLK을 100MHz CLK으로 변환하고, 모든 하위 모듈을 적절히 이어 전체 시스템을 구성하는 역할을 한다. 시스템 통합과 함께 I/O모듈에서 나온 4개의 segment 데이터를 2개씩 엮어 CLK 기반으로 segment 모듈에 출력하도록 하는 역할도 한다.  [clk\_100M]  : clk generator를 통해 기본 내장 clk인 125MHz를 이용하여 100MHz 주파수를 가지는 clk을 생성하였다.  [freq\_div\_100]  : clk의 주파수를 100배만큼 낮추는 역할을 한다. 코드를 보면, 원래의 clk이 100번 clocking할 때마다 출력 clk이 1번 clocking하도록 구성 되어있다. 이를 통해 clk의 주파수를 100배 낮춘다.  [btn\_filter]  : 버튼 입력에 대해 올바른 값을 얻기 위해서 synchronizer, debouncer를 통과하고 CLK을 쪼개는 코드를 포함한 버튼 필터 모듈이다.  [synchronizer, debouncer]  : 입력 버튼이 한 번 눌렸음에도 여러 번 눌린 것처럼 동작하는 현상 및 버튼으로 인한 노이즈를 감소하기 위한 모듈이다. synchronizer를 통해 버튼으로 인한 노이즈와 같은 입력 신호가 clk과 같은 형태로 변화하게 되며, debouncer를 통해 이 입력 신호가 안정화되었을 때만을 입력으로 보아, 이 때의 신호를 연결된 모듈의 입력으로 들어가게 한다.  [IO]  : 사용자 입력과 출력을 관리하고, 전체 FSM의 state를 관리한다. btn[0]의 입력에 따라 다음 state로 전환하며, btn[3]이 입력되면 state를 IDLE로 초기화한다. 각 상태는 led를 통해 표시되며, INS1~4 상태에서 사용자로부터 스위치 입력을 받고 Instruction의 각 부분을 구성한다. 구성중인 명령어는 실시간으로 segment를 통해 사용자에게 표시된다. DONE state에서 일반적인 경우 모든 led가 on되지만, overflow가 발생한 경우에는 1,3번째 led만 on되어 표시한다. 또한 DONE state에서 reg\_write 신호를 High로 출력하여 Register에 값이 쓰일 수 있도록 한다.  [SSD]  : 4비트 데이터를 입력하면 7-segment에 출력할 7비트 신호로 변환하는 변환기이다. 입력 값에 대해 정해진 출력 값을 반환한다.  [CTRL]  : 명령어에 따른 ALU의 동작과 ALU의 input, Register에의 Write 여부를 결정하는 신호를 생성한다. 이는 아래의 표를 참조하였으며, ALU\_OP 즉, 명령어에 맞는 ALU의 각 동작의 enable 신호가 1이 되도록 코드를 구성하였다. ALU\_Src의 경우, Reg[Rd2] 값을 사용해야 할 때 0을 출력하도록 하였고, Rd2 값을 사용해야 할 때에는 1을 출력하도록 하여 이를 아래 MUX의 select signal로써 활용되도록 하였다.    [Reg4x16]  : 12비트의 instruction 데이터로부터 Rd1, Rd2, Wr을 구성하고, Reg\_Write 신호와 Data\_Write 값을 받아 동작하는 16열의 4비트 레지스터다. Rd1, Rd2 주소의 레지스터 값을 Data\_Read 1, 2로 출력하고, Reg\_Write 신호와 함께 Wr 주소에 Data\_Write 값을 저장한다. 또한 항상 0번 주소의 레지스터에 4’b0000 값을 유지하도록 한다.  [Mux\_4x2]  : 4비트 데이터 src1과 src2를 select 신호에 따라 선택하여 출력하는 단순한 MUX이다.  [ALUU]  : 이 마이크로프로세서의 모든 연산 동작에 대한 집합이다. 각 동작들은 모두 각각의 모듈로 구성하였고, 이들의 입력은 MUX를 통해 결정된다(Reg[Rd2] or Rd2). CTRL에서 결정된 동작에 맞는 출력만을 ALU의 출력으로 사용하고, 해당 동작에 따른 enable 신호가 인가되지 않는다면 0을 출력한다. overflow의 경우, 덧셈과 뺄셈의 overflow신호를 구분하여 각각의 동작에서의 overflow만을 출력으로 내보내게 하였다. 모든 모듈은 가시성을 위해 입력과 출력을 통일하였으며, 각각의 ALU 동작 모듈에서 en은 해당 동작의 enable 신호, 첫 번째 입력의 Rd1는 Reg[Rd1]의 값을 고정적으로 나타낸다. 두 번째 입력 Rd2의 경우, mux에 의하여 Reg[Rd2]와 Rd2값이 달라지며, 하위 모듈 설명 간에는 이를 두 번째 입력이라고 하겠다. en이 인가되었을 때의 결과 출력은 &&와 \*을 이용하여 구현하였다.  [Write]  : write 동작의 경우, Rd2의 데이터를 출력으로 하여 이를 Wr address에 쓴다. 따라서 write enable 신호가 인가되었을 때, 두 번째 입력의 값을 출력하여 이 값을 write하는 데에 사용하도록 하였다.  [Read]  : read 동작의 경우, enable 신호가 인가되었을 때 Reg[Rd1]의 데이터를 읽어야 한다. 이를 위해서, 첫 번째 입력의 값을 출력하여 읽을 수 있도록 하였다.  [Copy]  : copy 동작의 경우, enable 신호가 인가되었을 때 Reg[Rd1]의 데이터를 복사하여 쓴다. 이를 위해 첫 번째 입력의 값이 출력되도록 하였다.  [NOT]  : not 동작의 경우, enable 신호가 인가되었을 때 첫 번째 입력에 대한 not operation 값이 출력으로 나오도록 하였다.  [ANDD]  : and 동작의 경우, enable 신호 인가 시, 두 입력에 대한 and operation 값이 출력으로 나오도록 하였다.  [ORR]  : or 동작의 경우, enable 신호 인가 시, 두 입력에 대한 or operation 값이 출력으로 나오도록 하였다.  [XORR]  : xor 동작의 경우, enable 신호 인가 시, 두 입력에 대한 xor operation 값이 출력으로 나오도록 하였다.  [NANDD]  : nand 동작의 경우, enable 신호 인가 시, 두 입력에 대한 nand operation 값이 출력으로 나오도록 하였다.  [NORR]  : nor 동작의 경우, enable 신호 인가 시, 두 입력에 대한 nor operation 값이 출력으로 나오도록 하였다.  [add]  : add 동작의 경우, enable 신호 인가 시, 두 입력에 대한 덧셈 값이 출력으로 나오도록 하였다. 이때, 두 입력은 signed binary라고 생각하여 덧셈을 하며, 덧셈 동작 구현은 Carry lookahead adder를 통하여 하였다. overflow의 경우, en 신호를 통한 add 동작이 수행될 때이고, 두 입력 간의 부호 비트가 같지만 이 비트가 덧셈 결과의 부호 비트와 다를 때 overflow라고 간주하였다.  [sCLA4]  : carry lookahead logic이며, 하위 모듈인 CLL과 GPFA를 통해 최상단의 carry와 sum을 계산한다.  [CLL4]  : carry lookahead logic이며, 최상단의 carry와 G, P를 계산한다. 마이크로프로세서의 경우, 덧셈 동작의 sum만을 필요로 하기 때문에 G와 P는 사용하지 않는다.  [GPFA]  : 각 비트의 G와 P, 그리고 sum을 계산하여 출력한다. 이때의 sum을 통하여 최종 덧셈 결과를 출력하게 된다.  [sub]  : 뺄셈 표현을 위한 2의 보수화 모듈과 이를 계산하기 위한 CLA를 이용하여 뺄셈을 구현하였다. 뺄셈은 (첫째 입력) - (둘째 입력) 순서로 진행되며, 뺄셈에 대한 enable 신호가 인가되었을 때만 이때의 계산 결과가 출력되도록 하였다. overflow의 경우, 두 입력의 부호가 다르고 첫 입력의 부호 비트와 sum의 부호 비트가 다를 경우 overflow로 간주하도록 하였다.  [comp\_2s]  : 뺄셈을 위한 2의 보수화 모듈이며, 반전을 통한 1의 보수화를 한 후 +1을 하여 2의 보수화가 되도록 하였다.  [LSF]  : enable 신호 인가 시, Reg[Rd1]의 값을 Rd2만큼 shift left한 값을 출력으로 가지도록 하였다.  [RSF]  : enable 신호 인가 시, Reg[Rd1]의 값을 Rd2만큼 shift left한 값을 출력으로 가지도록 하였다.  **- 작성한 Test Bench Code가 어떤 동작을 하는지?**  : 최상단 모듈인 MPU에 대하여 테스트 벤치를 제작하여 테스트하였다. 기본 clk의 주기는 8ns 즉, 125MHz의 주파수로 하였고 이는 보드에 대한 기본 내장 clk이다. 우선 rst 버튼인 btn[2]를 통하여 clk에 대한 rst를 진행하였고, 동작의 시작은 버튼들의 입력과 관련된 clk\_100M이 생성되기 시작하는 약 300ns부터 시작하도록 하였다.  : 최상단 모듈인 MPU에 대하여 테스트 벤치를 제작하여 테스트하였다. 기본 clk의 주기는 8ns 즉, 125MHz의 주파수로 하였고 이는 보드에 대한 기본 내장 clk이다. 우선 rst 버튼인 btn[2]를 통하여 clk에 대한 rst를 진행하였고, 동작의 시작은 버튼들의 입력과 관련된 clk\_100M이 생성되기 시작하는 약 300ns부터 시작하도록 하였다. btn[0]을 눌러 상태를 변화시키면서 명령어 **10A5**, **10C3**, **A358**을 차례로 입력하였다. 이후 마지막 명령어인 **A358**의 입력이 마치고 결과가 출력되는 ‘done’ 상태에서 출력을 확인한 뒤, 20ns 후 시뮬레이션을 종료하도록 하였다.  이 과정에서 btn[0]의 입력은 btn[0]=1이 입력되고 20ns 후 btn[0]=0으로 하여 버튼이 한번 테스트할 수 있도록 하였다. 명령어의 각각의 동작을 보겠다. **10A5**는 Reg[5]에 A를 Write 하도록 한다. **10C3**은 Reg[3]에 C를 Write하도록 한다. 마지막 명령어인 **A358**은 Reg[3]+Reg[5]의 값을 Reg[8]에 Write하도록 한다.  **- Simulation 파형이 왜 그렇게 나온 것인지?**  : 우선, clk generator를 통해 125MHz의 clk으로 100MHz의 clk을 만들었음을 clk\_100MHz에서 확인할 수 있다. 또한 freq\_div\_100을 통해 clk\_1M을 만들었기에 1MHz의 clk 또한 만들어 졌음을 파형을 통해 확인할 수 있다. 초반에 btn의 파형이 4에서 0으로 변한 것을 통하여 btn[2]=1->0으로 하여 clk의 rst이 이루어졌음을 확인할 수 있다. 이제 버튼에 대한 동작을 확인하겠다. btn[0]이 눌렸을 때, state는 0->1 즉, ‘idle’ -> ‘명령어 입력 1’로의 상태 천이가 이루어졌다. ‘명령어 입력 1’을 나타내는 led[3] 또한 led=1000=8을 통해 켜졌음을 확인할 수 있다. ‘명령어 입력 1’ 상태에서 스위치를 통해 1번 입력을 나타내자 seg1\_w 또한 1을 표시하는 06(hex)를 나타내었다. 다시 btn[0]이 눌렸을 때, 상태는 2 즉, ‘명령어 입력 2’로 천이하였고, 이는 led=0100=4를 통해서도 확인할 수 있다. 이 때 스위치를 통해 0을 나타내자 seg2\_w는 계속 3f(hex)를 유지하고 있다. 이때, seg1\_w가 계속 06(hex)를 유지하고 있는 것으로 보아 지금까지 입력된 명령어가 잘 유지되고 있음을 볼 수 있다. 같은 과정으로 btn[0]이 눌렸을 때, 상태는 ‘명령어 입력 3’으로 천이, led=0010=2로 출력이 이루어지고 스위치를 통해 A를 입력하자 seg3\_w에 77(hex) 출력, 이전까지의 명령어 유지 seg1\_w, seg2\_w를 통해 확인하였다. 마지막으로 btn[0]을 눌러 ‘명령어 입력 4’로의 천이, led=0001, 스위치=5를 입력하자 seg4\_w=6d(hex), 이전까지의 명령어 유지를 확인하였다. 명령어의 입력이 끝나고 btn[0]을 누르면 상태가 ‘done’ 상태로 천이하며 led=1111=f 출력이 되고, Register인 regs의 파형을 보면 regs[5]에 A가 잘 Write되었음을 확인할 수 있다. ‘done’ 상태에서 btn[0]을 눌러 ‘idle’ 상태로 가면 seg가 모두 0=3f(hex)로 초기화되며 led=0000이 됨을 확인할 수 있다. 이를 통해 write 동작에 대한 시뮬레이션은 성공적으로 이루어졌음을 확인할 수 있었다. 이후, **10C3** 통해 C를 regs[3]에 Write하였다. 이는 regs 파형을 통해서도 확인할 수 있다. 이제 ALU 동작을 확인하기 위한 **A358** 명령어의 시뮬레이션 파형을 보겠다. 위와 같은 동작을 통해 명령어의 입력과 상태 천이가 이루어졌고, ‘명령어 입력 4’ 상태에서 btn[0]을 눌러 ‘done’ 상태로 가면, 출력은 둘의 덧셈 결과인 3f(0), 06(1), 06(1), 3f(0)을 출력하고 있고, 이는 overflow 값으로 이 또한 overflow led 출력인 led=1010=a에서 확인할 수 있다. regs의 파형으로 regs[8]에 이 결과가 Write된 것 또한 확인할 수 있다. 이 시뮬레이션을 통해 write와 ALU연산이 잘 이루어지고 있음을 확인할 수 있었다.  추가로, **HW debugging**의 파형을 분석해보겠다. **10A5를 입력한 후 btn[0]을 눌렀을 때에 대한 디버깅**에 대해 생각해보겠다. 이는 ‘명령어 입력 4’에서 ‘done’으로의 상태 천이이다. 이때의 파형을 보면, state = 4 (‘명령어 입력 4’), next state = 5 (‘done’)으로 표시되어 있고, led=0001, {seg1, seg2, seg3, seg4} = {06, 3f, 77, 6d} = 10A5가 입력되어 있음을 확인할 수 있다. 이를 통해 명령이의 입력과 상태 천이가 지금까지 잘 이루어졌음을 확인할 수 있다. 이때, flted\_btn 신호의 0->1 변화를 보아, btn[0]의 입력이 잘 들어가고 있음을 확인할 수 있다. 하지만 btn[0]의 입력이 전달되어도 동시에 state와 led의 변화로 이어지지는 않는데, 이는 delay 때문으로 생각되며, simulation 파형에서도 btn 입력과 동시에 상태 천이로 이어지지 않음을 확인할 수 있었고 이와 관련 지어 생각해볼 수 있다. 이는 regs[5]에 아직 a가 Write 되지 않았음을 통해서도 생각할 수 있다. 다음으로, **위의 상태에서 btn[0]을 눌렀을 때에 대한 디버깅**에 대해 생각해보겠다. 파형을 보면, state = 5 (‘done’), next state = 0 (‘idle’)로 표시되어 있고, led=ffff, {seg1, seg2, seg3, seg4} = {06, 3f, 06, 3f} = 1010이 입력되어 있음을 확인할 수 있다. 현재가 ‘done’ 상태이므로 regs[5]에 a가 write되어있어야 했고, 파형을 통해 regs[5]=a가 확인되며 Write 동작이 제대로 수행되었음을 확인할 수 있다. flted\_btn 신호의 0->1 변화를 보아, btn[0]의 입력이 잘 들어가고 있음을 확인할 수 있다. 하지만 위에서와 동일하게 상태천이와 led 변화로 이어지지 않는다. 위에서와 동일한 이유로 생각되며, 이는 segment의 출력이 유지되어 명령어가 0000으로 초기화되지 않음을 통해서도 생각할 수 있다. |