

학과 전자공학과, 물리학과

학번 12171457, 12181457, 12181473, 12202073

이름 강종희, 김범수, 김현수, 김혜림

제출일 23.06.18

**VLSI Microprocessor Project**

**목차**

**Micro-Processor**

**SRAM**

**SRAM Cell**

**Pre-Charge**

**Sense Amplifier**

**Write Driver**

***SRAM Block***

**SRAM Controller**

**Decoder**

**D F/F Using SR-Latch**

**D-MUX**

**ALU**

**간단한 이론**

**Micro-Processor**

**Micro-Processor**는 Computer System에서 **CPU** (Central Processing Unit)의 핵심 부분으로 작동하는 집적 회로 칩입니다. 이 장치는 Computer의 Instruction Set을 실행하고 데이터를 처리하는 역할을 담당합니다. 위의 프로젝트를 빌려 설명해보면 산술 및 논리 연산을 수행하는 중앙 장치이다. 이제 마이크로프로세서에 대해 자세히 설명해보겠습니다.

**Components**

**· Control Unit**: Instruction Decode and Execute, Data Transfer, 연산을 제어하는 부분

**· ALU**(Arithmetic Logic Unit): 연산과 논리 연산을 수행하는 부분

**· Register**: Data를 일시적으로 저장하는 기억 장치

**· Clock**: Micro-Processor의 모든 작업을 동기화하는 신호를 발생시키는 타이밍 장치

**RISC** (Reduced Instruction Set Computer): RISC Architecture는 단순화되고 간소화된 Instruction Set으로 구성하여 Hardware 구조를 간단하게 만드는 방식이다. 이들은 일반적으로 Single Clock Cycle로 실행할 수 있는 소수의 간단한 Instruction을 가지고 있다. RISC Processor는 각 Instruction의 실행 시간을 최적화하여 전반적인 성능을 향상시키는 것을 목표로 한다.

**CISC** (Complex Instruction Set Computer): CISC Architecture는 복잡한 Instruction Set으로 구성됩니다. 이러한 Processor는 일반적으로 Single Instruction으로 복잡한 작업을 수행할 수 있는 다단계 작업을 포함하여 다양한 Instruction을 지원합니다. CISC 아키텍처는 종종 이전 명령어 집합과의 역호환성과 관련이 있습니다.

최근에는 더 높은 수준의 Programming Language를 사용할 수 있기에 Assembly Language의 활용도가 떨어졌고, Mobile 및 Wearable의 확대는 전력 효율성과 소형 Hardware에 더욱 중점을 두고 있습니다. 따라서 RISC (Reduced Instruction Set Computer) Architecture에 대한 의존도가 증가하고 있습니다. RISC Architecture의 대표적인 예인 ARM은 널리 채택되어 Apple의 M1 칩과 Samsung의 Snapdragon 프로세서를 포함한 다양한 애플리케이션에 사용되고 있다. 이러한 RISC 기반 설계로의 전환은 현대 컴퓨팅 장치에서 에너지 효율과 소형화에 대한 업계의 우선순위를 반영한다.

도표, 라인, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

**Fig1.** SRAM Microprocessor

**SRAM Block**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **6T-SRAM**  6T-SRAM Cell은 Cross-Coupled Inverter에 1 Bit Data를 저장하는 Unit Structure로, 6개의 Transistor로 이루어진다. P1, D1, P2, D2가 Inverter Latch를 구성하고, Word Line에 연결된 A1, A2 Pass Transistor로 Read/Write Operation을 제어한다. 현재 Layout에서 Q와 Q­\_B에 각각 반대의 Data를 저장하며 1Bit를 표현할 수 있고, Static Storage이기에 시간이 지나도 Refresh가 필요하지 않습니다.  집적도 향상을 위해 Layout은 상하 반전된 형태의 2개를 위아래로 배치하여 배열합니다. (b)와 같은 구조의 반복으로 VDD와 VSS Line을 공유할 수 있고, D1과 A1을 Merge할 수 있다. Layout에서 가로 방향과 세로 방향은 서로 다른 Metal을 사용하여 중첩되어 연결되지 않도록 설계하였다. 왼쪽과 같은 배치 방식을 반사 또는 중첩 구조라고 하며 설계를 진행하는 Memory는 8x4 이기에, 총 16개의 중첩 구조를 사용하면 설계 가능하다.  **Operation**  **Write Operation**  1. Target Word Line에 VDD를 인가한다.  2. Write Driver에 Data를 인가하여 Bit, Bit B를 “VDD, GND” 혹은 “GND, VDD”로 원하는 Data에 맞게 인가한다.  3. Inverter Latch에 Data가 저장된다. 이후 Word Line을 끄면 Pass Transistor가 꺼져 Data Update가 완료된다.  **Read Operation**  1. Read할 Cell을 선택하기 위해 Word Line에 VDD를 인가한다.  2. Pre-Charge를 통해 Bit, Bit B Line를 VDD로 만들어준다.  3. Word Line으로 Pass Transistor가 켜져 있는 상태이기에, Bit, Bit B Line에 전압 변화가 발생한다.  4. Sense Amplifier가 위 전압 변화를 읽고, 증폭하여 Cell에 저장된 Data를 출력한다.  도표, 개략도이(가) 표시된 사진  자동 생성된 설명  (a) (b)  **Fig2.** 6T-SRAM  (a) 6T-SRAM Cell Schematic (b) Reflection Structure with 6T-SRAM  **SRAM Sizing**  SRAM 내부 PUN (Pull-Up Network), PDN (Pull-Down Network), Pass Transistor의 Sizing한 근거는 다음과 같다. (b)와 같이 Read Operation에서 A에 저장된 “Data-0”가 “Data-1”로 Flip되는 상황이 발생한다면, 이는 Read Failure이 발생했다 할 수 있다. 그렇기에 Pass Transistor보다 PDN을 Strong하게 설계해야 Failure를 줄일 수 있다.  다음으로 Write Operation에 대해 살펴보겠다. “Data-1”이 저장된 A b를 “Data-0”으로 Flip해야 하는 상황에 PUN가 Strong하여 Data Flip이 발생하지 않고 1이 유지된다면 Write Failure이 발생하게 된다. 이를 해결하기 위해 Pass Transistor를 PUN보다 Strong하게 설계한다.  정리하면 PUN를 Weak하게 설계하고, PDN를 Strong하게 설계해야 Operation Failure이 발생하지 않게 된다. 또한 Pass Transistor는 PUN와 PDN의 중간 정도로 설계하면 된다.    (a) (b) (c)  **Fig3.** SRAM Sizing for Read and Write Operation  (a) SRAM Cell Sizing (b) Read Operation (c) Write Operation  위 구조를 적용하여 Cadence Tool을 활용하여 Schematic과 Layout을 설계했고, 이를 바탕으로 SRAM Block을 설계한다. 이때 PUN은 220nm, PDN는 이의 2배인 440nm, Pass Transistor는 중간 정도의 330nm로 설계를 진행했다. 또한 (b)는 중첩 구조를 활용하여 2개의 SRAM Cell이 대칭 구조로 구성된 것을 확인할 수 있다.   |  |  | | --- | --- | |  | 스크린샷, 텍스트, 디스플레이, 다채로움이(가) 표시된 사진  자동 생성된 설명 | | (a) | (b) | |
| **Fig4.** Cadence 6T-SRAM Schematic and Layout  (a) 6T-SRAM Cell Schematic (b) 6T-SRAM Cell layout |

**Pre-Charge**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pre-Charge Circuit은 Read Operation에서 Bit, Bit B Line을 VDD로 Pre-Charge하기 위한 회로이다. Bit, Bit B line이 Pre-Charge 신호에 따라 VDD에 연결될 수 있도록 PMOS로 Switch를 만들어 주고, Bit와 Bit B line 사이의 전압 균형을 맞춰주기 위해 두 Line 사이에 PMOS를 추가하여 구성한다. 기존에는 2개의 PMOS만으로 Pre-Charge Circuit을 구현할 수 있으나, Bit, Bit B를 S/D로 하는 PMOS를 1개 추가하면 더 빠른 시간내에 각 Line을 Pre-Charge할 수 있다.  Pre-Charge Signal가 GND이면 Pre-Charge Circuit이 동작하여 Bit, Bit B Line이 VDD로 Pre-Charge되고, VDD가 인가되면 Pre-Charge Circuit은 동작하지 않습니다. 현재 SRAM Block은 8x4 Structure이다. 즉, 4개의 Bit Line과 Bit B Line이 존재하기에 Pre-Charge Circuit은 4개를 사용하게 됩니다. 이는 Read Operation에서 사용하기 위해 만들어진 회로이다.    (a) (b)  **Fig5.** Pre-Charge Circuit Schematic  (a) Without Equivalent PMOS (b) With Equivalent PMOS  위 구조를 적용하여 Cadence Tool을 활용하여 Schematic과 Layout을 설계했고, 이를 바탕으로 SRAM Block을 설계한다. Pre-Charge Circuit을 구성하는 모든 PMOS Transistor는 동일한 Size로 설계했고, Pre-Charge Signal이 인가될 때 Bit, Bit B Line을 빠르게 VDD로 Pre-Charge하기 위해 Size를 기본 Size인 180nm에서 530nm로 높여 설계를 진행했다.   |  |  | | --- | --- | |  | 스크린샷, 텍스트, 다채로움, 라인이(가) 표시된 사진  자동 생성된 설명 | | (a) | (b) | |
| **Fig6.** Cadence Pre-Charge Schematic and Layout  (a) Pre-Charge Schematic (b) Pre-Charge layout |

**Sense Amplifier**

Sense Amplifier는 Write Operation에서 Bit, Bit B Line의 전압 차이를 감지하여 증폭하는 기능을 한다. Sense\_Clk 입력이 들어오면 PMOS가 SRAM cell로부터 Sense Amp를 Isolation 시키고 Bit line과 Bit B Line을 Regeneration 시켜서 Sense 신호를 Digital level로 변환한다.

(a)는 기본적인 Sense Amplifier를 나타내고, (b)는 Isolation Transistor를 사용해서 Power Consumption을 줄여주는 회로입니다. 추가적으로 몇 개의 Transistor를 사용하지만, 전력 관점에서는 이득이기에 오른쪽 구조를 적용해서 SRAM Block을 설계했습니다.

도표, 라인, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명 도표, 텍스트, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

(a) (b)

**Fig7.** SRAM Pre-Charge Circuit

(a) Without Isolation Transistor (b) With Isolation Transistor

위 구조를 적용하여 Cadence Tool을 활용하여 Schematic과 Layout을 설계했고, 이를 바탕으로 SRAM Block을 설계한다. 미세한 전압 차이를 인지하여 Voltage Amplifier의 동작을 수행해야 하기에, 전체적인 Transistor Size를 크게 가져갔다. 만약 VDD의 전압이 더 높은 Level이거나 SRAM Cell의 설계를 Tight하게 진행한다면 Sense Amplifier Circuit의 Size를 줄일 수 있는 여지가 존재한다.

|  |  |
| --- | --- |
|  |  |
| (a) | (b) |

**Fig8.** Cadence Sense Amplifier Schematic and Layout

(a) Sense Amplifier Schematic (b) Sense Amplifier layout

**Write Driver**

Write driver는 SRAM cell에 Write할 때, Bit, Bit B Line을 구동시키기 위한 회로이다. NMOS는 Switch의 역할로, Write Signal 입력을 통해 write driver를 sram cell과 연결되게 하고 Write-Data 신호에 연결된 인버터를 통해 Bit와 Bit B Line에 차동 데이터를 인가한다.

(a)는 강의노트에 제안된 Write Driver Schematic이고, (b)는 구글링을 통해 찾은 Write Driver Schematic이다. 이는 Inverter의 사용을 1개 줄일 수 있고, 이는 Power Consumption과 집적도 모두 높일 수 있기에 오른쪽 구조를 적용해서 SRAM Block을 설계했다.

공통적으로 Write Signal이 VDD인 경우에 NMOS Transistor가 켜저 Bit & Bit B Line과 연결된다. 이를 통해 입력받은 Data를 SRAM Cell에 저장할 수 있고, 위 Data는 2:1 MUX에 의해 새로운 Data를 입력 받거나, ALU Result를 입력받게 된다.

도표, 라인, 폰트, 그래프이(가) 표시된 사진

자동 생성된 설명 도표, 라인, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명

(a) (b)

**Fig9.** Write Driver Schematic

(a) With 2 Inverter Write Driver (b) With 1 Inverter Write Driver

위 구조를 적용하여 Cadence Tool을 활용하여 Schematic과 Layout을 설계했고, 이를 바탕으로 SRAM Block을 설계한다. Write Driver는 Bit Line 혹은 Bit B Line의 Terminal을 Float 혹은 GND와 연결되게 설계를 진행하면 되기에 기본적인 Transistor Size로 설계를 진행했다. 또한 위에서 살펴봤듯이 Inverter를 1개 덜 사용하는 구조를 채택하여 전체적인 Microprocessor의 Size를 줄이기 위해 노력했다.

|  |  |
| --- | --- |
|  |  |
| (a) | (b) |

**Fig10.** Cadence Write Driver Schematic and Layout

(a) Write Driver Schematic (b) Write Driver layout

**SRAM Block**

SRAM Block을 구성하기 위해 Pre-Charge, Sense Amplifier, Write Driver, SRAM Cell을 연결하여 Layout을 완성했다. 위에서 언급했듯이 SRAM Cell의 중첩 구조를 활용하여 Layout의 Size를 최소화 했으며, Word Line과 Data Out을 고려하여 최적화를 진행했고, 31.3 x 88.7 Size로 Layout을 최적화 할 수 있었다.

Word Line은 총 8개 (W0 ~ W7)로 구성되고, SRAM Decoder에서 출력되어 들어간 SRAM Controller의 입력을 통해 각 Word Line의 입력으로 인가된다. 이를 통해 SRAM Block을 Control할 수 있다.

Bit & Bit B Line은 차례로 Pre-Charge, SRAM Cell, Sense Amplifier, Write Driver로 연결된다. 이때 Sense Amplifier의 하단 출력부가 Data Out을 의미하는데, 증폭된 값을 출력해야 하기에 위와 같이 설계를 진행했고 “8x4 Structure”이기에 4개의 출력이 존재한다. 또한 위에서 살펴봤듯 Data Input은 Write Driver를 통해 인가되고 “8x4 Structure”이기에 4개의 Input이 존재한다.

위 Block을 설계함에 있어서 가장 중요하게 생각한 부분은 면적이었다. 전체 Layout을 통해 Micro-Processor를 만드는 과정에서 SRAM Block 자체의 세로 크기가 매우 크면, 다른 Block들의 세로 크기가 작더라도 면적을 낭비할 수 있게 된다. 그렇기에 중첩 구조와 최적의 Sizing을 찾기 위해 노력했다.

스크린샷이(가) 표시된 사진

자동 생성된 설명 다채로움, 스크린샷, 패턴, 마조렐 블루이(가) 표시된 사진

자동 생성된 설명

(a) (b)

**Fig11.** Cadence SRAM Block Schematic and Layout

(a) SRAM Block Schematic (b) SRAM Block layout

**SRAM Controller**

|  |
| --- |
| **SRAM Controller**  **Pre Charge:** WR = 1 일 때는 1로 유지되고, WR = 0 인 Read Operation의 반주기 동안, CLK’와 함께 실행되어야 한다. 따라서 Pre Charge의 논리식은 다음과 같다.  **Pre-Charge** = WR + CLK’ = (WR’ CLK)’  **Sense Clock**: WR = 1 일 때는 0으로 유지되고, WR=0 인 Read Operation의반주기 동안, 즉 CLK’와 함께 실행되어야 Read Operation시 Pre-Charge 이후 Sense Amp의 증폭이 일어난다. 이때 Sense Clock의 실행은 Pre-Charge의 실행보다 늦도록 Delay 조절을 해주어야 한다. Sense Clock의 논리식은 다음과 같다.  **Sense Clock** = WR’ CLK’ = (WR + CLK)’  **Word[n]:** SRAM Cell에 연결되는 Word Line과 연결되는 NMOS의 Gate에 연결되어 Word[n] = 1일 때 해당 Word의 SRAM Cell들이 동작할 수 있게 된다. 따라서 Word[n]은 W [n] = 1일 때 CLK’와 함께 동작하므로 Word[n]의 논리식은 다음과 같다.  **Word[n]** = W[n] CLK’ = (Word[n]’ + CLK)’    (a) (b)  **Fig12.** SRAM Controller  (a) SRAM Controller Block (b) Input and Output Signal  스크린샷, 우주이(가) 표시된 사진  자동 생성된 설명스크린샷, 디스플레이, 텍스트, 회로이(가) 표시된 사진  자동 생성된 설명  (a) (b) |
| **Fig13.** Cadence SRAM Controller Schematic and Layout  (a) SRAM Controller Schematic (b) SRAM Controller layout |

**SRAM Decoder**

**SRAM Decoder**

Decoder는 n개의 Input Line을 2nOutput Line으로 변환하는 회로이다. SRAM Block은 8x4 Structure로 8개의 Word Line이 필요하다. 그렇기에 3개의 Input Line으로 8개의 Output Line을 만들어낼 수 있는 3Bit Decoder를 사용할 것이다. 실제 Schematic에서는 SRAM Decoder의 3Bit Input Data로 ADDR<2:0>를 사용하여 “W[7] ~ W[0]”을 만들어낸다.

또한 아래의 (a), (b)를 생각해보면, Static CMOS 또는 Pseudo-NMOS로 Decoder를 구성할 수 있다. 동작 속도는 Pseudo-NMOS가 빠를 수 있으나 거의 차이가 나지 않을 것이라 생각했고, 전체 Size 관점으로 생각해봐도 Static CMOS가 유리해서 (a) Structure로 SRAM Decoder 설계를 진행했다.

아래의 Schematic을 살펴보면 Inverter 2개를 사용하여 AND Gate의 입력 조합을 만들어 줬고, 이유는 Fanout 문제를 해결하기 위함이다. 하지만 실제 동작에 있어 Clock의 주기가 상대적으로 길어 Fanout 문제가 크게 발생하지 않았고 Inverter 1개를 제거하면 Layout의 전체적인 Size를 줄일 수 있기에 1개의 Inverter를 사용하여 A와 A-B 조합을 만들었다.

도표, 텍스트, 라인, 기술 도면이(가) 표시된 사진

자동 생성된 설명도표, 텍스트, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명스크린샷이(가) 표시된 사진

자동 생성된 설명

(a) (b) (c)

**Fig14.** SRAM Decoder

(a) SRAM Decoder Static CMOS (b) SRAM Decoder Pseudo-NMOS (c) Cadence SRAM Decoder Schematic

스크린샷, 다채로움, 회로, 예술이(가) 표시된 사진

자동 생성된 설명

**Fig15.** SRAM Decoder Layout

**D F/F Using SR-Latch**

**SR Flip Flop**

Flip Flop은 Level Trigger Latch 2개를 연결하여 구성하게 된다. 이때 같은 Level로 설계하게 되면 Flip Flop의 역할을 수행하지 못하기에, High Level – Low Level Latch 구조 또는 Low Level - High Level Latch 구조로 설계해야 한다. 이는 차례로 Falling Edge Flip Flop, Rising Edge Flip Flop을 나타낸다.

위의 프로젝트에서는 Rising Edge Trigger Flip Flop을 사용해야 하기에 Low Level - High Level Latch Structure로 설계를 진행했고, “프로젝트 강의노트”를 참고하여 100MHz 이하의 Clock에서도 사용할 수 있는 SR-Latch를 활용하여 Flip Flop을 구현했다. 이때 각 NAND Gate, Inverter의 Size는 기본 Size로 진행하되 병렬과 직렬을 고려하여 Sizing을 진행했다. 또한 실제 동작에 있어 Q\_B를 사용하지 않기 때문에 Slave Latch에서 QN의 출력을 삭제했다.

**※ Setup and Hold Time**

Flip Flop을 Setup Time과 Hold Time을 만족시켜야 정확한 동작을 수행할 수 있다. 이를 만족하지 않는다면 Data의 변형 및 소실을 초래할 수 있다. 이는 Clock의 문제에 의해 주로 발생할 수 있다. 이를 해결하기 위해서 PLL을 활용하여 Clock을 정교하게 만드는 것에서부터 시작을 해야 할 것이다.

하지만 Cadence 상황에서 Clock은 입력만 하면 가능하기 때문에 이의 문제는 발생하지 않고, 결국 문제는 Wire Delay 혹은 Logic Delay에 의해 발생하게 된다. 그렇기에 Clock의 주기를 조금 넉넉하게 잡아 Flip Flop의 정상적인 동작을 기대해야 한다.

또한 아래의 “SR Flip Flop”에서 AND Gate와 NOR Gate를 사용하는데, 이를 Bubble Method를 활용하여 고려하면 모든 Gate를 NAND Gate로 구현할 수 있고, 이는 아래의 “Fig17.”에 표현했다. 이의 장점으로는 AND Gate는 NAND Gate에서 Inverter를 추가해서 설계를 진행해야 하기에 Size와 Speed 관점에서는 손해이지만, 이를 모두 NAND Gate로 설계를 진행한다면 Speed 와 Size 관점에서 모두 이득이기 때문이다.

도표, 라인, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

**Fig16.** SRAM SR Flip Flop

스크린샷, 우주이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| **Fig17.** 1 Bit D Flip-Flop Schematic |

전자제품, 스크린샷, 다채로움, 회로이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| **Fig18.** 1 Bit D Flip-Flop Layout |

스크린샷, 우주, 도표, 천문학이(가) 표시된 사진

자동 생성된 설명스크린샷, 다채로움, 패턴, 예술이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| (a) (b) |

**Fig19.** 4 Bit D Flip-Flop Schematic and Layout

(a) 4 Bit D Flip-Flop Schematic (b) 4 Bit D Flip-Flop Layout

**8 to 4 MUX**

**8 to 4 MUX**

8 to 4 MUX는 SRAM Block의 Input Data, 즉 Write Driver에 입력되는 Data를 결정하기 위해 설계한 Block이다. 이의 목적은 ALU Result를 다시 SRAM Block으로 입력하여 SRAM Cell에 저장할 것인가 혹은 Data를 입력 받아 SRAM Cell에 저장할 것인가 결정하기 위함이다. 이를 위한 Select Signal은 DIR\_SRAM으로 선택하고, 이는 Main Controller에서 출력하는 Signal이다.

도표, 평면도, 기술 도면, 라인이(가) 표시된 사진

자동 생성된 설명

**Fig20.** 8 to 4 MUX Block

|  |
| --- |
| 회로, 스크린샷, 다채로움, 라인이(가) 표시된 사진  자동 생성된 설명 |
| **Fig21.** Cadence 2 to 1 MUX Schematic and Layout |

스크린샷, 도표, 천문학이(가) 표시된 사진

자동 생성된 설명스크린샷, 다채로움이(가) 표시된 사진

자동 생성된 설명

**Fig22.** Cadence 8 to 4 MUX Schematic and Layout

**D-MUX**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **D-MUX**    **Fig23.** D-MUX  MUX가 2n개의 입력과 n개의 Select Signal을 이용해여 1개의 출력으로 선택하는 논리회로라면, DMUX는 1개의 입력선과 n개의 Select Signal을 이용하여 2n개를 출력하는 논리회로이다. DEMUX는 “Fig23”과 같이 SRAM으로부터 4Bit의 Data를 받고, Main Controller로부터 DIR\_EXE (S0, S1)의 2Bit Select Signal을 받아 4Bit씩 총 12Bit의 3가지 가능한 경우를 출력하는 역할을 한다. DMUX는 신호를 출력할 것인지 (output), SRC1으로 보낼 것인지, SRC2로 보낼 것인지를 결정하며, 이 동작을 위한 S0 S1은 아래 표와 같이 되도록 설계하였다.   |  |  | | --- | --- | | S0 S1 |  | | 00 | Output | | 01 | SRC1 | | 10 | SRC2 |   DMUX 설계 시 Fig24 (a)와 같이 할당하고, Fig24 (b)와 같이 뒤에 D-F/F을 추가하여 설계할 수도 있다. 이렇게 하면 S0, S1 Signal에 따라 3개의 D-FF 중 하나의 Line에만 Data를 출력하고 나머지 Line에서는 전부 “Data-0”으로 만든다. 그 결과 ALU를 하기 위해 SRC1, SRC2가 저장이 되어 있어야 하는데, 저장되지 못하고 0으로 떨어지는 것을 방지할 수 있다. 다만, 본 프로젝트에서는 이와 같은 방법은 사용하지 않고 설계를 진행하였다.     |  | | --- | | (a) (b) |   **Fig24.** D-FF을 추가한 D-MUX  (a) S1 S0 할당 (b) MUX와 D-FF을 추가한 모습  위의 표와 같이 동작하는 Pre-DMUX를 설계하고, 전체 4Bit의 출력을 받기 위해서 Pre-DMUX 4개를 병렬 연결하여 최종 D-MUX를 완성했다. 아래의 Schematic을 확인해보면 Select Signal의 조합이 “11”일 때 동작을 하지 않는 것을 확인할 수 있다. 이를 통해 DMUX의 출력이 존재하지 않을 때 위 조합을 사용해 출력을 하지 않도록 하는 방법이다. | |
| 스크린샷, 다채로움, 도표이(가) 표시된 사진  자동 생성된 설명 |
| **Fig25.** Pre-DMUX Schematic and Layout |
|  |

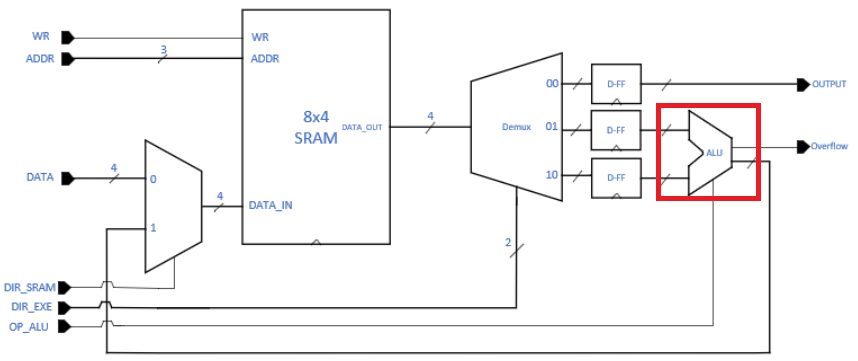
스크린샷, 우주, 도표, 천문학이(가) 표시된 사진

자동 생성된 설명다채로움, 스크린샷, 마조렐 블루, 라인이(가) 표시된 사진

자동 생성된 설명

**Fig26.** DMUX4 Schematic and Layout

**ALU**



**Fig27.** ALU

ALU는 SRAM에서 정보를 받아와서 이를 이용해 연산을 진행하는 Bloc이다. 이번 Project에서 ALU는 Adder와 Subtractor로만 구성하였다. ALU는 1 Bit Full Adder와 XOR을 이용하여 구성하였고 ALUOP를 이용하여 Adder와 Subtractor를 하나의 HW로 구성할 수 있도록 하였다. 먼저 1Bit Full Adder를 구현하였다. 기본적인 1 Bit Full Adder의 진리표는 아래와 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | CIn | Cout | S |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

이를 논리식으로 나타내면 다음과 같다.

S=A ⊕ B ⊕ CIn,

Cout=AB+(A ⊕B)CIn

가감산기를 하나의 HW로 설계하기 위하여 위 Adder을 수행하면서 Subtractor로서 동작하는 방법을 고안하였다. 먼저 입력단에 XOR을 추가하여 ALUOP가 1일 때 B와 XOR 연산을 진행하고 1을 더해주어 2의 보수법을 사용해 B를 -B로 만들어주었다. 이 동작에 따라 A+B가 A+(-B)로 변화하게 되어 Subtractor의 동작을 수행할 수 있게 된다. 이때 ALUOP가 0인 경우 B 값이 변화하지 않고 0을 더해주기 때문에 동작에 아무런 변화 없이 Adder 동작을 수행할 수 있다.

따라서 위 결과 ALUOP=0 : Adder, ALUOP=1 : Subtractor로 동작할 수 있었다.

|  |
| --- |
| **Fig28.** 1 Bit Full Adder Schematic  위 그림은 앞선 내용을 바탕으로 설계한 1 Bit Full Adder의 Schematic이다. XOR을 사용하여 B를 ALUOPB 값에 따라 Adder와 Subtractor의 연산을 수행하도록 설정하였다. 위 Schematic에서 Cin과 ALUOPB가 서로 연결되어 있지 않아 정상적인 Subtractor의 동작을 수행할 순 없지만 4 Bit Full Adder의 동작을 위해 연결하지 않은 상태로 설계하였다. 이때 면적을 최대한 줄이기 위하여 Bubble을 이용하여 OR Gate를 NAND Gate로 변경하였다. |

스크린샷, 다채로움, 회로이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| **Fig29.** 1Bit Full Adder Layout  위 그림은 Schematic을 바탕으로 설계한 Layout이다. Metal 3를 사용하여 면적을 최대한 줄이려고 하였고 NAND와 XOR의 위치를 조정하여 최대한 빈자리 없이 Gate가 연결되도록 설계하였다. |

|  |
| --- |
|  |
| **Fig30.** 4Bit Full Adder Schematic |

위 그림은 4Bit Full Adder를 Schematic으로 나타낸 것이다. 맨 위쪽이 LSB, 맨 아래쪽이 MSB로 입력되도록 설정하였다. 정상적인 Subtractor 동작을 위하여 ALUOP와 CIn을 연결하여 ALUOP가 1이 되어 Subtractor로 회로가 동작하는 경우 최하위 Bit에 1을 더해주는 동작을 수행할 수 있도록 하였다. 또한 XOR 연산을 통해 모든 B의 Bit가 반전되어 결과적으로 B를 2의 보수로 변화시킬 수 있도록 하여 연산을 수행할 수 있도록 하였다.

스크린샷, 다채로움이(가) 표시된 사진

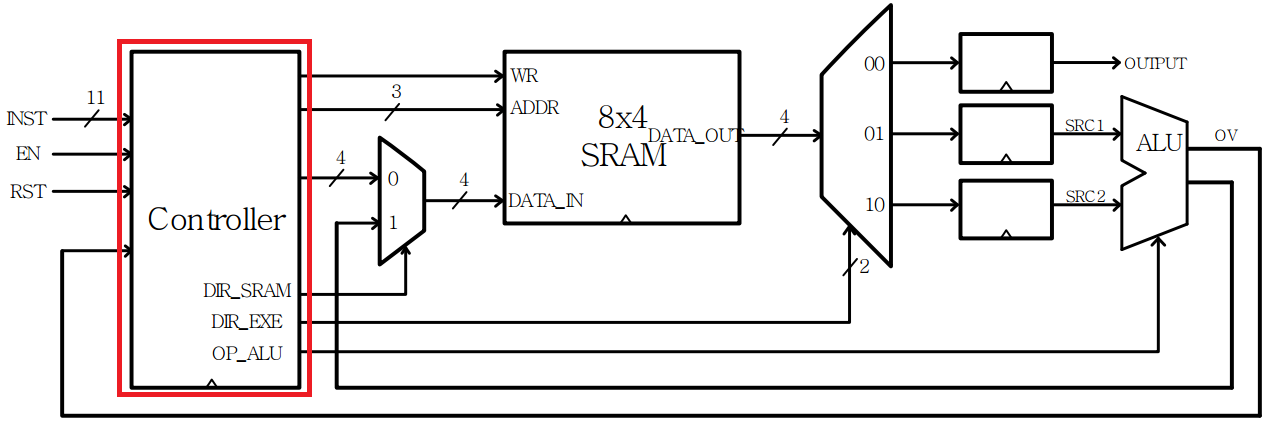
자동 생성된 설명

|  |
| --- |
| **Fig30.** 4Bit Full Adder Layout |

위 그림은 4Bit Full Adder의 Layout으로 마지막으로 XOR가 추가되어 ALU가 완성된 모습이다.

4Bit Full Adder를 서로 연결하고 마지막 단에 XOR를 통하여 Overflow를 Detection할 수 있도록 회로를 구성하였다. 모든 Output은 3시 방향을 향하게 하여 연결성을 높여 전체 회로를 합쳐 구성할 경우 면적을 줄일 수 있도록 구성하였다.

**Main Controller**

****

**Fig31.** Main Controller

Controller는 Instruction을 통하여 Microprocessor의 동작을 제어하는 Module이다. Instruction에 따라 State를 변경하고 그에 맞는 출력을 만들도록 Code를 작성하였다. 먼저 이 Controller의 입출력 신호는 아래와 같다.

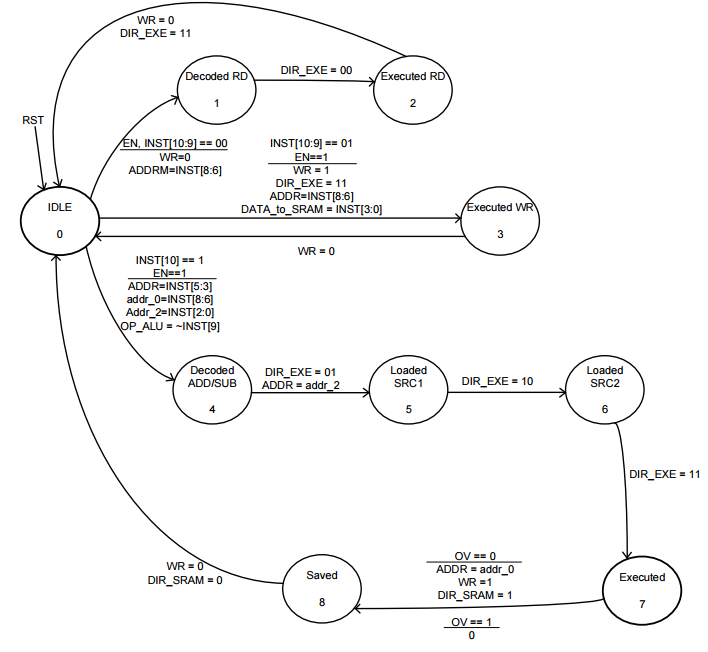
|  |  |  |
| --- | --- | --- |
| 입력 | INST | 11 Bits의 Instruction |
| EN | High일 때 Instruction을 Fetch |
| RST | High일 때 Controller Reset |
| OV | High일 때 ALU의 Result를 저장하지 않음 |
| 출력 | DIR\_SRAM | SRAM의 Data 입력 방향 결정 (0이면 Instruction, 1이면 ALU 결과에서) |
| DIR-EXE | DATA\_OUT을 출력 Register 혹은 ALU의 Source로 들어갈지 결정 |
| OP\_ALU | ALU 제어 신호 |

위 입력을 통해 원하는 출력을 만들어내기 위한 Instruction Set은 아래와 같다.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Inst[10] | Inst[9] | Inst[8] | Inst[7] | Inst[6] | Inst[5] | Inst[4] | Inst[3] | Inst[2] | Inst[1] | Inst[0] |
|  | OP Code | |  | | | | | | | | |
| Read | 0 | 0 | ADDR | | | Don’t Care | | | | | |
| Write | 0 | 1 | ADDR | | | Don’t Care | | Value | | | |
| Add | 1 | 0 | DEST | | | SRC1 | | | SRC2 | | |
| Sub | 1 | 1 | DEST | | | SRC1 | | | SRC2 | | |

**Fig32.** Main Controller Instruction

먼저 위 Instruction을 분석하면 MSB쪽의 2 Bit는 OP Code로 어떤 동작을 수행할지 나타내는 코드이다. Read, Write, Add, Sub 네가지 Instruction으로 구분할 수 있도록 한다. Read와 Write 동작에서 8~6 Bits는 Address로 읽거나 쓸 주소를 가리키게 된다. 이때 Write는 미리 저장할 값을 설정해야 한다. 따라서 3~0 Bits를 Value 값으로 쓸 값을 설정할 수 있도록 하였다. ADD와 SUB는 8~6 Bits에 Destination Address를 설정하여 결과값을 어디에 저장할지 결정할 수 있도록 하였다. 5~3 Bits는 SRC1, 2~0 Bits는 SRC2로 어떤 주소의 값을 연산할지 선택할 수 있도록 하였다. 위와 같이 Instruction에 따라 원하는 출력을 만들어내기 위한 FSM는 아래와 같다.

****

**Fig33.** Main Controller FSM

Main Controller는 위 FSM을 사용하여 구현하였다.

* **Read**

EN=1이고 OP Code인 INST[10:9]이 00이면 Read동작을 수행하게 된다. 이 때 읽어올 주소인 ADDR에 INST[8:6]이 들어가고 Write 신호인 WR은 0이 된다. 이 제어 신호를 통하여서 SRAM에 해당하는 주소의 cell이 켜져 저장된 값들을 읽어오도록 동작한다. 이후 다음 클럭에서 DIR\_EXE=00으로 설정MUX로 출력되어 최종 OUTPUT을 출력한다.

* **Write**

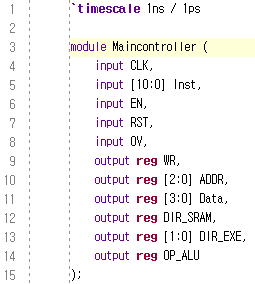
Write 동작은 EN=1이고 OP Code인 INST[10:9]이 01일 때 실행된다. WR=1로 설정하여 Write 동작을 수행한다. 이때 DIR\_EXE을 11로 설정하여 MUX에 해당하는 값이 없도록 하여 출력이나 ALU의 입력으로 들어가지 않게 된다. Write 동작을 마치면 WR=0으로 바뀌어 더 이상의 Write 동작을 수행하지 않도록 하고 IDLE로 State를 초기화 한다.

* **ADD/SUB**

OP Code를 OP\_ALU 신호로 사용하여 ADD 동작이나 SUB 동작을 결정한다. 만약 명령어의 10번째 비트가 1이라면 ADD/SUB 동작을 수행하게 된다. 이때, 먼저 Address(ADDR)를 통해 SRC1에 해당하는 주소를 할당한다. 그리고 addr\_0과 addr\_2 레지스터에는 각각 DEST와 SRC2 값을 저장한다. OP\_ALU 신호는 OP\_code의 9번째 비트를 반전시킨 값으로 설정되며, OP\_ALU가 1일 경우 ADD를, 0일 경우 SUB 동작을 수행하도록 한다. 또한 S4에서 DIR\_EXE가 01일 때, INST[5:3]에 해당하는 SRAM 값을 ALU의 첫번째 Input인 SRC1으로 들어가는 레지스터에 저장한다. Address(ADDR)에는 Instruction의 SRC2인 addr\_2 값을 할당한다. 만약 DIR\_EXE가 10이라 면, INST[2:0]에 해당하는 SRAM 값을 SRC2에 들어가는 레지스터에 저장하게 된다.

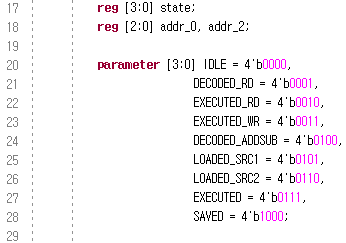
위의 동작을 통해 ALU는 SRC1과 SRC2 신호를 받아들여 연산을 수행할 수 있게 된다. Overflow가 발생하지 않을 경우, ADDR에는 DEST에 해당하는 addr\_0 값을 할당하고, WR=1, DIR\_SRAM=1을 설정하여 연산 결과 값을 DEST 주소에 기록한다. 동작이 완료된 후에는 IDLE 상태로 초기화한다.

위 내용을 기반으로 아래 코드를 구현하였다.



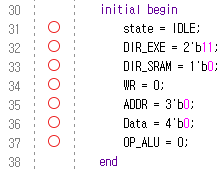
**Fig34.** Main Controller Verilog Code

먼저 입력을 정상적으로 받고 출력을 내기 위하여 위와 같은 코드를 작성하였다. CLK, Inst, EN, RST, OV를 Input으로 설정하였고 output으로 WR, ADDR, Data, DIR\_SRAM, DIR\_EXE, OP\_ALU를 reg 형식으로 선언하였다.



**Fig35.** Main Controller Verilog Code

중간 변수인 상태를 나타내는 state와 addr\_0, addr\_2를 선언하였다. 또한 각 State의 이름을 할당하여 parameter로 선언하였다.



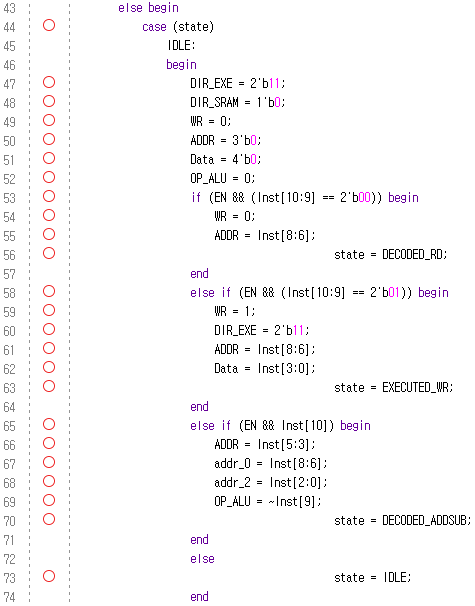
**Fig36.** Main Controller Verilog Code

먼저 모든 값을 초기화해 주었다.



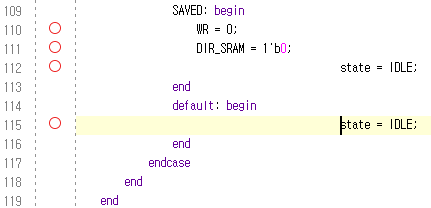
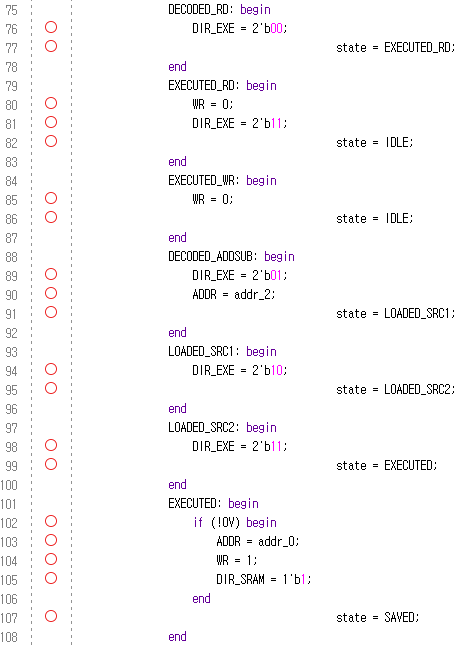
**Fig37.** Main Controller Verilog Code

다음으로는 주요 동작을 하는 always문을 작성하였다. 주어진 코드는 CLK의 positive edge에서 동작하도록 구현하였으며 RST 신호가 들어오면 IDLE 상태로 돌아가기 위하여 위와 같은 코드를 작성하였다.



**Fig38.** Main Controller Verilog Code

먼저 초기 상태인 IDLE 상태에 대해 정의하였다. 모든 변수를 초기화 하고 EN과 OP Code인 Instruction의 10~9 Bits를 통해 Read, Write, Add, Sub를 구분할 수 있도록 하였다. Add와 Sub는 동일한 State로 이동하기 때문에 Instruction의 10 Bit만 비교해 다음 State로 이동, 변수 설정, output을 출력하도록 하였다. EN 신호가 0인 경우 IDLE 상태를 유지하도록 설정하였다. Read 동작을 하는 경우 WR을 0으로 설정하고 ADDR 신호를 출력하여 SRAM으로 DATA를 읽어내 DMUX로 출력하도록 하였다. Write 동작을 할 때 data를 SRAM에 자성하여 WR, ADDR, Data를 전달할 수 있도록 하였다. Add, Sub 동작은 DEST, SRC1, SRC2, 주소를 저장하고 OP\_ALU를 출력하도록 하였다.



**Fig39.** Main Controller Verilog Code

DECODED\_RD는 Read 동작을 진행하기 위해 DMUX의 Input을 00으로 설정하였으며 다음 State인 EXECUTED\_RD로 이동하도록 하였다.

EXECUTED\_RD에서는 WR을 0으로 초기화해주고 DIR\_EXE도 11로 초기화 해 주어 더 이상 값이 읽히지 않도록 하였다. 이후 IDLE 상태로 돌아가도록 하였다.

EXECUTED\_WR은 Write를 위한 State로 WR을 0으로 초기화하고 IDLE 상태로 돌아간다.

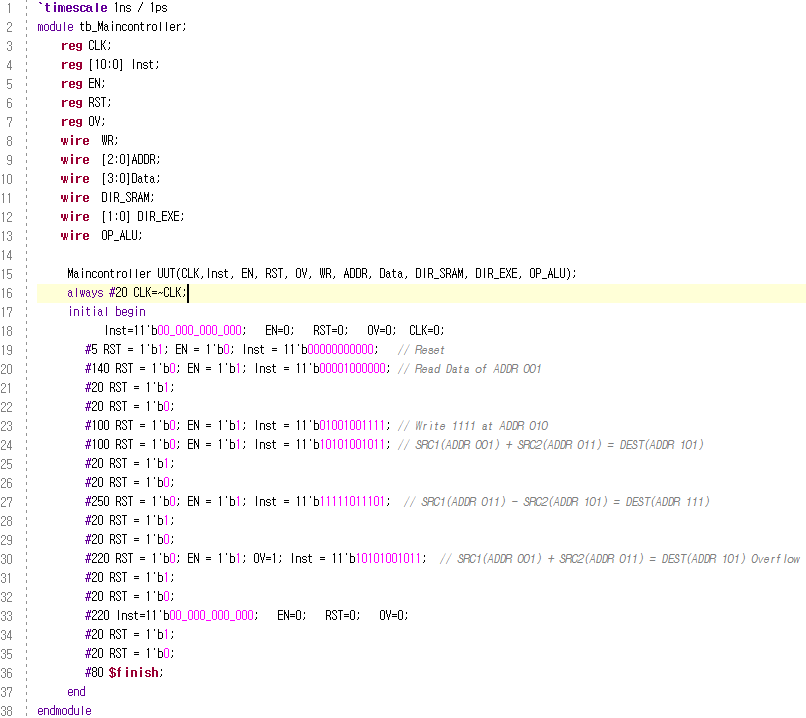
DECODED\_ADDSUB은 DIR을 1로 설정하여 SRC1에 값이 들어가도록 하고 ADDR에 다음 address인 addr\_2를 넣는다. 이후 다음 State인 LOADED\_SRC1로 이동한다.

LOADED\_SRC1에서는 SRC2를 대입하기 위하여 DIR\_EXE를 10으로 설정하였다. 이후 State를 LOADED\_SRC2로 설정하였다.

LOADED\_SRC2에서는 DIR\_EXE를 11로 설정하여 값이 들어가지 않도록 변경하고 State를 EXCUTED로 변경해주었다.

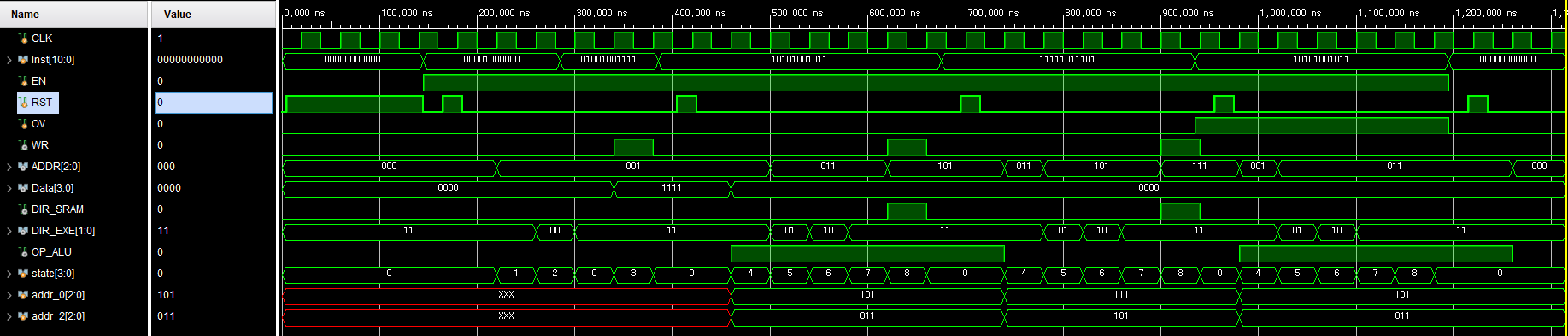
EXCUTED에서는 Overflow가 있었는지 확인하고 ALU에서 나온 값을 SRAM에 저장할지 판단한다. Overflow가 아닌 경우 ADDR에 addr\_0을 넣고 WR=1, DIR\_SRAM을 1로 설정하여 원하는 위치에 값을 저장한다.

SAVED에서는 WR과 DIR\_SRAM을 0으로 초기화 하고 IDLE로 돌아가도록 하였다.

****

**Fig40.** Main Controller Test Bench Verilog Code

위 코드는 이 동작을 적절하게 수행하는지 확인하는 Test Bench Code이다. Read, Write, ADD, SUB, Overflow가 발생하였을 때 Add가 제대로 동작하는지 확인해보았다.

**Fig41.** Simulation Result

위 Simulation 결과를 분석해보면 다음과 같다.

Read : Instruction (00/001/000000)을 대입하였을 때 정상적으로 ADDR이 1로 나오는 것을 확인할 수 있다.

Write : Instruction (01/001/00/1111)을 대입하였을 때 정상적으로 ADDR에 1이 나오고 Data에 1111이 나오는 것을 확인할 수 있다.

Add : Instruction(10/101/001/011)을 대입하였을 때 정상적으로 OP\_ALU가 1이 나오고 (~OP\_ALU 신호를 ALU에서 사용하기 때문에) ADDR이 순서대로 변화하는 것을 확인할 수 있다. 001(SRC1) -> 011(SRC2) -> 101(DEST)

Sub : Instruction(11/111/011/101)을 대입하였을 때 정상적으로 OP\_ALU가 0이 나오고 (~OP\_ALU 신호를 ALU에서 사용하기 때문에) ADDR이 순서대로 변화하는 것을 확인할 수 있다. 011(SRC1) -> 101(SRC2) -> 111(DEST)

Add Overflow : Instruction(10/101/001/011)을 대입하고 OV를 1로 설정하였을 때 WR 신호가 생성되지 않는 것을 확인할 수 있다. 위 동작을 통해 SRAM에 연산 된 결과를 저장하지 못하게 한다.

**Micro-Processor**

**Micro-Processor 설계 방안**

Main Controller에서 출력되는 Control Signal은 Cadence Simulation 환경에서 구현할 수 없기 때문에 “Setup Analog Stimuli”에서 목적에 맞게 인가한다. Layout 관점에서는 최적의 Layout을 설계하기 위해 Metal1, 2, 3을 활용하여 Micro-Processor를 구현했다. 이를 통해 “130 x 100 Size”의 Micro-Processor를 Layout으로 구현했다.

또한 강의에서는 D-F/F 뒤에 MUX를 추가하여 Data를 Hold하고, 이를 통해 ALU의 연산을 동일하게 진행할 수 있다. 하지만 늦게 출력되는 SRC2의 Data를 2개의 D F/F을 활용하여 설계했을 때 동일한 동작을 수행할 수 있고 Layout의 Size 관점에서 이득이 되기 때문에 위와 같이 진행했다.

아래에서 살펴볼 동작들은 다음과 같고, Operation을 수행하기 위해 각각의 Test Set들을 만들어 검증을 진행했다. 또한 위 Simulation은 “Clock = 1MHz”에서 진행한다.

**도표, 평면도, 기술 도면, 라인이(가) 표시된 사진

자동 생성된 설명**

**Fig42.** Given Micro-Processor

**도표, 평면도, 라인, 기술 도면이(가) 표시된 사진

자동 생성된 설명**

**Fig43.** Re-Interpreted Micro-Processor

**Add/Sub and Check the Result**

Overflow를 포함하는 ALU 연산 결과를 설명하기 위해 Simulation을 진행했다. 앞서 원래 Main Controller에서 제어해야 하는 Control Signal들을 만들어 SRAM의 Read/Write Operation을 수행할 수 있는 환경으로 설정해보겠다.

스크린샷, 소프트웨어, 텍스트, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**스크린샷, 블랙이(가) 표시된 사진

자동 생성된 설명**

**Fig44.** Control Signal Graph

**0~1us:** ALU 연산을 위한 SCR1을 쓰는 구간이다. Write Operation이기에 Pre-Charge Signal은 VDD로 고정되어 OFF되고, Sense Amplifier역시 OFF를 위해 GND로 유지된다. Write Operation이 발생하기에 당연하게도 Word Line은 VDD로 위 구간에서는 유지되어야 한다. 위의 그림을 참고하면 0.5us부터 1us까지 Word[7]이 VDD로 유지되는 것을 확인할 수 있다. 이 기간동안 Data에 값을 유지하면 그 Data는 Target SRAM Cell에 저장된다.

**1~2us**: ALU 연산을 위한 SCR2를 쓰는 구간이다. “0~1us”와 Operation은 동일하며, 다른 점은 Word[0]가 1.5us부터 2us까지 VDD로 유지된다는 점이다.

**2~3us**: 실제 ALU Operation을 위해 SCR1과 SCR2를 읽어내는 구간이다. Pre-Charge Signal은 반주기동안 GND를 유지하며 Bit, Bit B Line을 VDD로 Pre Charge한다. 2.5us 지점에서 Pre-Charge는 다시 VDD가 되어 OFF되고, Word[7]이 다시 VDD로 인가된다. 이때 Read Operation처럼 Sense Amplifier가 전압 차이를 읽어 DMUX의 D[4:0] Input으로 전달한다. 이후 DIR\_EXE가 “01”이 인가되어 SCR1이 출력된다.

**3~4us**: “2~3us”와 동일하게 동작하지만, 다른 점은 DIR\_EXE가 “10”이라는 부분과, Word[0]가 VDD로 인가된다는 점이다.

즉, 4us에서 SRC1과 SRC2가 동시에 ALU Input에 인가되며, ALU\_Op에 따라 Add Operation을 진행할 지, Sub Operation을 진행할 지 결정된다. 또한, ALU Result에 따라 Overflow 여부도 확인하며, ALU Result는 “8 to 4 MUX”에서 Data In으로 사용될 지 결정된다.

**Basic Read/Write Operation**

SRAM Block에 “1111”의 Data를 7번 Layer의 SRAM Cell에 저장하고, 읽어오는 과정을 설명하겠다. 이의 동작은 “0~1us”에 SRAM Cell에 Data를 저장하고, “1~2us”에 SRAM Cell에서 정보를 읽어오고 이를 Output으로 출력하는 과정이다. 이 과정은 ALU Operation에 들어가기에 앞서 SRAM Block Operation이 정상인지 확인할 수 있는 과정이다.

아래의 Figure를 살펴보면, Input Data (1111)이1us까지 유지되고, 이후 (0000)으로 Flip되는 것을 확인할 수 있다. 이는 설계자의 의도이다. 다음 위에서 살펴본 듯 Word[7]은 0.5us에서 Rising Edge를 가지고 이는 1us까지 유지된다. 아래의 “Q7 Cell”을 살펴보면 0.5us에서 Data가 (0000)에서 (1111)로 바뀌는 것을 확인할 수 있다.

DIR EXE를 초기에는 (11)로 유지하여 DMUX4의 동작을 방지한 이후, 1us에서 (00)을 인가하여 SRAM Block의 Output Data를 Output으로 연결할 수 있도록 했다. 아래의 Figure에서 확인해보면 1us에서 Output이 (0000)에서 (1111)로 Flip하며 SRAM Block이 정상적으로 동작한다는 것을 확인할 수 있다.

**텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

**Fig45.** Basic Read Write Operation without ALU

이제 SRAM Block의 Read & Write Operation이 정상적으로 작동하는 것을 확인했기에, ALU Operation이 정상적인지 확인해보겠다. Overflow가 발생하는 Add, 발생하지 않는 Add, 발생하는 Sub, 발생하지 않는 Sub, 총 4가지의 Case로 설계 초반에 의도했던 대로 동작하는지 확인해보겠다.

추가로 실제 Layout과 Schematic의 LVS를 맞춘 Model은 SRAM Cell 내부를 측정할 수 있는 Q 단자가 존재하지 않는다. 하지만 SRAM Operation 도중 Cell 내부의 정보가 바뀌는 것을 확인하기 위해서 Simulation을 진행하는 Schematic에서는 SRAM Cell과 연결된 Q Node를 통해 Cell 내부의 정보의 변화를 측정했다.

**Case1) SCR1 = 0101, SCR2 = 1010 ADD Case Without Overflow**

**스크린샷, 멀티미디어 소프트웨어, 소프트웨어, 전자제품이(가) 표시된 사진

자동 생성된 설명**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**Fig46.** SRC1, SRC2, Data Restoring

위 결과를 보면 SCR1과 SCR2에 원하던 Data가 입력되었고, 연산 결과는 “01012 + 10102 = 11112”로 잘 연산 된 것을 확인할 수 있고, Overflow는 발생하지 않은 것을 확인할 수 있다. 또한 Data의 Restoring은 “4us~5us”에 발생하는 것을 확인할 수 있다.

이의 Flow는 다음과 같다. ALU Result가 4us에 출력되고, DIR\_SRAM은 3us부터 VDD를 유지하기에 “8 to 4 MUX”의 출력은 ALU Output<3:0>이다. 이가 다시 Input Data가 되어 Write Driver에 인가된다. 이때 Word[7]은 4.5us부터 VDD를 유지하기에 Word[7] Line의 SRAM Cell의 Data에 “1111”이 저장된다.

또한 위 수는 Binary로 -8부터 7까지 표현 가능하다. 위 수는 “5 + (-6)”인 상황으로 출력이 “-1”이기에 Overflow가 발생하지 않는다.

**Case2) SCR1 = 0111, SCR2 = 0101 ADD Case With Overflow**

**스크린샷, 멀티미디어 소프트웨어, 그래픽 소프트웨어, 회로이(가) 표시된 사진

자동 생성된 설명**

**Fig47.** SRC1, SRC2, Data Restoring

위 결과를 보면 SCR1과 SCR2에 원하던 Data가 입력되었고, 연산 결과는 “01112 + 01012 = 11002”로 잘 연산 된 것을 확인할 수 있고, Overflow는 발생한 것을 확인할 수 있다. 또한 Data의 Restoring은 “4us~5us”에 발생하는 것을 확인할 수 있다.

원래라면 Overflow가 발생한 상황에서는 ALU Result를 SRAM Block에 저장하지 못한다. 하지만 이의 역할은 Main Controller가 수행하기에 위 상황까지는 구현하지 못했다. 간단하게 Pre-Charge Signal이나 Write Drive Signal을 제어함으로써 위 기능은 구현할 수 있다.

**Case3) SCR1 = 1111, SCR2 = 1010 Sub Case Without Overflow**

**스크린샷, 전자제품이(가) 표시된 사진

자동 생성된 설명**

**스크린샷, 라인, 다채로움이(가) 표시된 사진

자동 생성된 설명**

**Fig48.** SRC1, SRC2, Data Restoring

위 결과를 보면 SCR1과 SCR2에 원하던 Data가 입력되었고, 연산 결과는 “11112(-1) + 10102(-5) = 10102(-6)”로 잘 연산 된 것을 확인할 수 있고, Overflow는 발생하지 않은 것을 확인할 수 있다. 또한 Data의 Restoring은 “4us~5us”에 발생하는 것을 확인할 수 있다.

또한 위 수는 Binary로 -8부터 7까지 표현 가능하다. 위 수는 “(-1) + (-5)”인 상황으로 출력이 “-6”이기에 Overflow가 발생하지 않는다.

**Case4) SCR1 = 1111, SCR2 = 1000 Sub Case With Overflow**

**스크린샷, 전자제품, 멀티미디어 소프트웨어, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

**Fig49.** SRC1, SRC2, Data Restoring

위 결과를 보면 SCR1과 SCR2에 원하던 Data가 입력되었고, 연산 결과는 “11112(-1) + 10002(-8) = 01112”로 연산 된 것을 확인할 수 있고, 4Bit Binary는 “-8~7”까지 표현할 수 있기에 Overflow가 발생한 것을 확인할 수 있다. 또한 Data의 Restoring은 “4us~5us”에 발생하는 것을 확인할 수 있다.

원래라면 Overflow가 발생한 상황에서는 ALU Result를 SRAM Block에 저장하지 못한다. 하지만 이의 역할은 Main Controller가 수행하기에 위 상황까지는 구현하지 못했다. 간단하게 Pre-Charge Signal이나 Write Drive Signal을 제어함으로써 위 기능은 구현할 수 있다.

**Micro-Processor Layout**

아래의 Figure는 위에서 설계한 각 Block의 Layout을 합쳐 전체 Micro-Processor Layout을 만들었다. Layout의 배치는 배치의 최적화로 Area를 최소화하기 위해 Schematic과 상의하게 설계했다. 이를 통해 “128.45 x 102.11”으로 설계를 완성했다.

**다채로움, 스크린샷, 텍스트, 마조렐 블루이(가) 표시된 사진

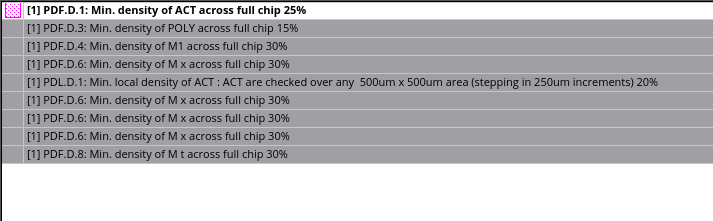
자동 생성된 설명**

**Fig50.** Micro-Processor Layout With SRAM

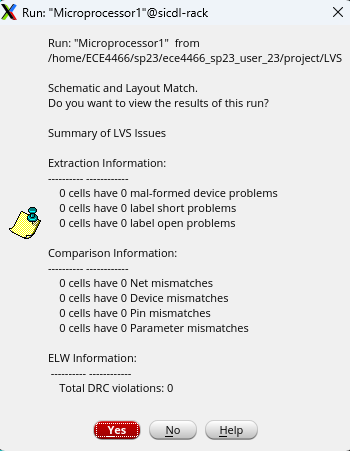
스크린샷, 회로, 밤이(가) 표시된 사진

자동 생성된 설명

**Fig51.** Micro-Processor Schematic With SRAM

****

**Fig52.** Micro-Processor DRC With SRAM

****

**Fig53.** Micro-Processor LVS With SRAM

**TOP Module은 user\_23-> VLSI2 폴더 -> MicroProcessor1 입니다.**