컴퓨터구조 프로젝트 보고서

Project 0

Implementation of MU0 Processor

수업 명: 컴퓨터구조

담당 교수: 이성원

학과: 컴퓨터정보공학부

학번: 2023202070

이름: 최현진

제출일: 2025.04.06

1. Introduction

MU0는 매우 간단한 16비트 마이크로프로세서이며 프로세서 디자인 입문 학습을 위해 설계되었다. 16비트 길이의 명령어는 4비트의 opcode와 12비트의 주소 공간(S)로 이루어져 있다. 따라서 총 8kilobyte의 데이터를 저장할 수 있다. 해당 프로젝트는 프로그램 카운터(PC), 누산기(ACC, Accumulator), ALU, 명령어 레지스터(IR), control logic과 같은 MU0 프로세서의 구성 요소를 logisim-e 내에서 구현하는 것을 포함한다. 이후 프로그램의 FETCH와 EXECUTE 두 단계의 동작을 확인한다. 또한, logisim-e 내에서 TG, MUX, Latch, D-FF, Counter와 같은 디지털 회로를 설계하고 동작을 이해하는 과정을 포함한다.

2. Assignment

2.1 Practice for digital logics with Logisim-E

|  |
| --- |
| 1. What is the Transmission gate?    Transmission Gate (TG)는 PMOS와 NMOS를 병렬적으로 연결한 스위치와 같은 회로이다.  NMOS와 PMOS는 각각 신호 전달 시에 손실이 발생한다. NMOS는 low 값 전달 시에는 손실이 없지만, week 1을 전달한다. 반대로 PMOS는 high 신호는 잘 통과하지만 week 0를 전달한다.  따라서 PMOS와 NMOS를 병렬로 연결하면, high 신호는 PMOS를 통해 손실 없이 전달되고 low 신호는 NMOS를 통해 손실 없이 전달된다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2. Implement a MUX using Transmission gates    두 개의 TG를 사용해 구성하였다. 선택선 S가 1일 때, A와 연결된 TG가 열리고 B와 연결된 TG는 닫히므로 출력 Y는 A를 전달한다. 반대로 S가 0일 때는 A 쪽 TG가 닫히고 B 쪽 TG가 열리면서 Y는 B값을 출력하게 된다. TG는 S와 S’를 각각 제어선으로 사용한다.  S = 0 -> Y = B, S = 1 -> Y = A  <진리표>   |  |  |  |  | | --- | --- | --- | --- | | A | B | S | Y | | 0 | 0 | 0 | 0 | | 0 | 1 | 0 | 1 | | 1 | 0 | 0 | 0 | | 1 | 1 | 0 | 1 | | 0 | 0 | 1 | 0 | | 0 | 1 | 1 | 0 | | 1 | 0 | 1 | 1 | | 1 | 1 | 1 | 1 | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3. Implement a Latch using MUXs.    En = 0 -> Q에 이전 D값을 유지하고  En = 1 -> Q에 D 값을 저장하는 것을 확인했다.  MUX의 입력 0은 출력 Q의 피드백, 입력 1은 외부 입력 D로 연결했다. 선택선으로는 Enable 신호를 사용한다. logisim 내장 mux는 2번에서 구현한 회로와 다르게 S값이 입력과 같다. (S=0 input 0, S=1 input 1)  <진리표>   |  |  |  |  | | --- | --- | --- | --- | | En | D | 이전 Q | Q | | 1 | 0 | 0 | 0 | | 1 | 0 | 1 | 0 | | 1 | 1 | 0 | 1 | | 1 | 1 | 1 | 1 | | 0 | X | 이전 Q | 이전 Q | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4. Implement D-FF using Latches.      D flip flop은 클럭의 상승 엣지 또는 하강 엣지에서 데이터를 저장하고 유지하는 기능을 하는 회로이다.  3번에서 만든 latch 두 개 사용, master 래치에는 선택선에 CLK 를 연결하고, slave 래치 선택선에는 CLK’를 연결했다. master의 출력이 slave의 입력으로 들어가 동작하게 된다. 따라서 CLK=1이면 마스터가 D값을 저장하고, CLK=0: 슬레이브가 마스터의 출력을 받아 Q에 D 값이 저장되는 것을 확인했다.  <진리표>   |  |  |  | | --- | --- | --- | | CLK | D | Q | | ↑ | 0 | 0 | | ↑ | 1 | 1 | | ↓ | X | 이전 Q | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5. Implement Synchronous Up/Down Counter using D-FFs and adders.  <Components Description>  CLK: 상승 엣지에서만 출력 A가 증가하거나 감소한다.(동기식)  reset: 초기값을 설정할 수 있다.  mux: 선택선(up\_down)을 통해 0일 때 up(0001을 adder의 입력으로 사용), 1일 때 down(1111을 adder의 입력으로 사용) 카운터로 동작하게 한다. A0(lsb)과 연결된 adder의 B 입력은 항상 1이기 때문에 mux는 3개를 사용한다. mux의 입력으로 상수를 사용할 수 있다.  adder: Q인 A 입력과 증감 모드에 따른 B 입력의 가산 결과로 출력은 d-ff의 d로 들어간다.  A3~A0: 카운터 출력값으로 lsb가 A0이다.  <진리표>   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | CLK | reset | up\_down | 이전 Q | 다음 Q | 설명 | | ↑ | 1 | X | XXXX | 0000 | 리셋 | | ↑ | 0 | 0 | 0000 | 0001 | up | | ↑ | 0 | 0 | 1111 | 0000 | up(overflow) | | ↑ | 0 | 1 | 0001 | 0000 | down | | ↑ | 0 | 1 | 0000 | 1111 | Down(underflow) |   <Diagram Example (2-bit)>    Logisim을 이용하여 구성한 4비트 Synchronous Up/Down Counter에서, up\_down 스위치를 토글하며 A의 출력이 0000 → 0001 → 0010으로 증가함을 확인하였다. DOWN으로 전환 시, 반대로 감소하는 것을 관찰할 수 있었고, 1111에서 증가 시 0000으로 되돌아가는 overflow 동작도 정상적으로 확인되었다.      -up\_down: 0 (증가 모드)  0000~1111까지 1비트씩 A가 증가하는 up counter 동작을 수행한다. 1111에서 overflow시 0000으로 돌아오고 다시 0001로 증가하는 것을 확인했다.      -up\_down: 1 (감소 모드)  1111~0000까지 A가 1비트씩 감소하는 down counter 동작을 수행한다. 0000에서 underflow 시 1111으로 출력되고 다시 1110으로 감소하는 것을 확인했다.  현재 회로에서는 메모리 없이 FF로만 상태를 저장하고 있다. 프로그램 가능한 카운터로 확장하기 위해서, ROM이나 RAM 블록을 활용하여 초기 상태를 지정하거나 조건에 따라 상태를 바꿀 수 있도록 개선할 수 있을 것이다. |

|  |
| --- |
| 6. Compare Synchronous counter and Asynchronous counter such as a ripple counter.  공통점:   * 상향(up), 하향(down) 카운터가 있다.   Synchronous counter(동기식 카운터):   * 모든 ff에 동일한 클럭 신호를 사용하기 때문에 고속 처리가 가능하다. * 프로세서, 메모리 주소 생성 등 정밀한 제어가 필요할 때 응용된다.   Asynchronous counter(비동기식 카운터, 리플 카운터)   * 회로가 간단하다. * 첫번째 ff에만 클럭 신호가 입력된다. * 앞쪽의 ff 출력이 뒤쪽의 ff의 클럭으로 사용된다. * ff가 많아짐에 따라 delay가 커진다. * 오차 가능성이 높다. * 간단한 타이머, 빈도 측정에 사용된다. |

|  |
| --- |
| 7. What makes that Asynchronous counter is rarely used?  비동기식 카운터는 6번에서 설명했듯, 이전 ff의 출력이 다음 ff의 클럭 신호로 연결되는 방식이기 때문에 전파 지연(propagation delay)이 누적된다. 이로 인해 카운터의 속도가 느려지고, 최종 출력까지 도달하는 과정에서 오차가 생긴다. 따라서 빠르고 정밀한 동작이 요구되는 실제 시스템에서는 비동기식 카운터가 거의 사용되지 않는다. |

2.2 Complete the implementation of MU0 (2 cycle CPU)

|  |
| --- |
| 0. 전체 회로도 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. Complete the design by connecting all wires labeled as w1 to w8    w1~w8 control signals을 디코더에 연결했다. 디코더의 선택선은 Opcode로 설정했고, 각각의 관련 명령어 실행 시 w1~w8 제어 신호가 출력되도록 구성하였다. 다음은 w1~w8 제어선들의 기능과 동작을 정리한 표이다.   |  |  |  |  | | --- | --- | --- | --- | | 제어선 | 관련 명령어 | 기능 | 설명 | | w1 | LDA | MUX 0000 선택, ACC WE | Adder 입력을 0000으로 하여 메모리에서 로드한 값만 ACC에 반영 | | w2 | SUB | Adder 뺄셈 수행, ACC WE | Adder가 2의 보수(invert, +1) 적용하여 뺄셈 수행하고 ACC에 저장 | | w3 | ADD | MUX 이전 ACC 값 선택, WE | Adder 입력을 기존 ACC 값으로 하여메모리에서 불러온 값과 덧셈 수행하고 ACC에 저장 | | w4 | JGE | 비교기 출력과 함께 AND 입력, PC WE | 비교기 조건 검사 이후 메모리에서 불러온 값과 새로 계산한 주소로 JUMP | | w5 | JNE | 비교기 출력과 함께 AND 입력, PC WE | 비교기 조건 검사 이후 메모리에서 불러온 값과 새로 계산한 주소로 JUMP | | w6 | JMP | 무조건 PC WE 활성화 | 무조건 JUMP | | w7 | STP | PC WE 비활성화 | 디코더에 연결된 NOR 게이트의 출력이 비활성화되어 PC WE 비활성화, 프로그램 종료 | | w8 | STO | 메모리 WE | 메모리 WE와 연결된 AND 게이트의 입력으로 들어가서 WE 활성화, 메모리 쓰기 작업 가능 | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2. With the given binary file which will be loaded into the RAM, write the MU0 assembly program and explain how it operates  <MU0 명령어 표>     |  |  |  |  | | --- | --- | --- | --- | | OpCode (hex) | OpCode | 명령어 | 설명 | | 0 | 0000 | LDA S | ACC <- M[S] | | 1 | 0001 | STO S | M[S] <- ACC | | 2 | 0010 | ADD S | ACC <- ACC + M[S] | | 3 | 0011 | SUB S | ACC <- ACC – M[S] | | 4 | 0100 | JMP S | PC <- S | | 5 | 0101 | JGE S | If ACC >= 0, PC <- S | | 6 | 0110 | JNE S | If ACC =/ 0, PC <- S | | 7 | 0111 | STP | 프로그램 종료 |   먼저 위에서 정리한 MU0 명령어 표와 RAM 파일 내용, 구현한 회로를 토대로 예상 프로그램 흐름을 설명하겠다. 현재 구현한 MU0에서 모든 명령어는 두 개의 Cycle, FETCH와 EXECUTE로 수행된다.   * FETCH:   명령어를 읽고 해석하는 단계이다. 먼저 현재 PC가 가리키는 주소에서 16bit 명령어를 읽고 IR(Instruction Register)에 저장한다. IR의 명령어는 디코더의 입력으로 들어가 w1~w8 중 해당하는 제어선이 활성화되어 해석된다. PC는 1 증가한다.   * EXECUTE:   명령어 종류에 따라 수행된다:  LDA, ADD, SUB: IR의 주소 필드로 메모리 접근, ALU 연산, ACC에 저장  STO: ACC값을 메모리 주소에 저장 (메모리 WE 활성화)  JMP, JGE, JNE: IR 주소를 PC에 무조건 또는 조건에 따라 저장  STP: PC WE가 비활성화되어 FETCH가 멈추고 정지 상태 유지  다음으로 실제 동작 결과를 마찬가지로 FETCH, EXECUTE 두 단계와 클럭 번호에 따라 분석하겠다. 각 입력 또한 클럭에 따라 동작을 확인했다. 예상한 구조대로 명령어 종류에 따라 제어선이 알맞게 활성화되고 동작함을 확인했다. JNE 000 명령에서 0400 명령어 점프하여 무한 루프가 시작된다. JGE 008에서 조건 검사에 실패하고 7000 명령어(STP 000)로 점프하지 않기 때문에 구현한 프로그램은 종료되지 않을 것으로 예상했고 확인한 결과는 아래와 같다.     |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 0 | Fetch | 0 | - | 000 | - | Fetch | | 1 | Execute | 0 | 0 | 400 | LDA 400 | ACC←0020 (M[400]) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 2 | Fetch | 1 | - | 001 | - | Fetch | | 3 | Execute | 1 | 2 | 400 | ADD 400 | ACC ← 0040 (0020 + M[400]) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 4 | Fetch | 2 | - | 002 | - | Fetch | | 5 | Execute | 2 | 1 | 800 | STO 800 | M[800] ← 0040(ACC) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 6 | Fetch | 3 | - | 003 | - | Fetch | | 7 | Execute | 3 | 0 | 401 | LDA 401 | ACC ← 5555 (M[401]) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 8 | Fetch | 4 | - | 004 | - | Fetch | | 9 | Execute | 4 | 2 | 402 | ADD 402 | ACC←55FF (5555 + 00AA(M[402])) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 10 | Fetch | 5 | - | 005 | - | Fetch | | 11 | Execute | 5 | 3 | 403 | SUB 403 | ACC←55AA (55FF – 0055(M[403])) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 12 | Fetch | 6 | - | 006 | - | Fetch | | 13 | Execute | 6 | 1 | 801 | STO 801 | M[801] ← 55AA(ACC) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 14 | Fetch | 7 | - | 007 | - | Fetch | | 15 | Execute | 7 | 4 | 010 | JMP 010 | PC ← 0500 (M[010]) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 16 | Fetch | 10 | - | 010 | - | Fetch | | 17 | Execute | 10 | 0 | 500 | LDA 500 | ACC ← 7FFF (M[500]) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 18 | Fetch | 11 | - | 011 | - | Fetch | | 19 | Execute | 11 | 2 | 501 | ADD 501 | ACC←8000 (7FFF + 0001(M[501])) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 20 | Fetch | 12 | - | 012 | - | Fetch | | 21 | Execute | 12 | 5 | 008 | JGE 008 | if ACC(8000) >= 0, JMP M[008](7000) -> no JMP |   ACC값인 0x8000가 음수이기 때문에 jump 조건에 실패했다.       |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 22 | Fetch | 13 | - | 013 | - | Fetch | | 23 | Execute | 13 | 3 | 501 | SUB 501 | ACC←7FFF (8000 – 0001(M[501])) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 24 | Fetch | 14 | - | 014 | - | Fetch | | 25 | Execute | 14 | 1 | 802 | STO 802 | M[802] ← 7FFF(ACC) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 26 | Fetch | 15 | - | 015 | - | Fetch | | 27 | Execute | 15 | 5 | 009 | JGE 009 | if ACC >= 0, JMP to M[009](0600) -> yes JMP |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 28 | Fetch | 9 | - | 009 | - | Fetch | | 29 | Execute | 9 | 0 | 600 | LDA 600 | ACC ← 1234 (M[600]) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 30 | Fetch | A | - | 00A | - | Fetch | | 31 | Execute | A | 3 | 600 | SUB 600 | ACC←0000 (1234 – 1234(M[600])) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 32 | Fetch | B | - | 00B | - | Fetch | | 33 | Execute | B | 0 | 616 | LDA 616 | ACC ← 0000 (M[616]) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 34 | Fetch | C | - | 00C | - | Fetch | | 35 | Execute | C | 2 | 600 | ADD 600 | ACC←1234 (0000 + 1234(M[600]) |          |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 36 | Fetch | D | - | 00D | - | Fetch | | 37 | Execute | D | 1 | 803 | STO 803 | M[803] ← 1234(ACC) |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 38 | Fetch | E | - | 00E | - | Fetch | | 39 | Execute | E | 6 | 000 | JNE 000 | if ACC(1234) =/ 0, JMP M[000](0400) -> yes JMP |        |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Clock | Cycle | PC | Opcode | Address | Instruction | Operation | | 40 | Fetch | 0 | - | 000 | - | Fetch | | 41 | Execute | 0 | 0 | 400 | LDA 400 | ACC ← 0020 (M[400]) |   First Clock 상태(PC = 0)로 되돌아왔으며, 예상한 동작과 같이 프로그램이 반복된다.    STP 명령 검증을 위해 RAM 파일을 수정하였다.      PC의 WE가 비활성화되어, 더 이상 PC에 값이 쓰이지 않고 execute 스테이지에서 아무 실행이 일어나지 않는다. |

3. IMPORTANT NOTICE for Memory

M0 프로세서는 16bit 명렁어 길이 중 하위 12bit를 메모리 주소로 가지고 있기 때문에 메모리 주소는 2^12 = 4K개이다. 그리고 하나의 주소가 저장하는 메모리 크기는 16bit = 2byte이다. 따라서 총 메모리 용량은 8Kbyte이다.

현재 메모리는 RAM으로 구현되어 있지만, 실제 시스템은 부팅을 위해 일부분 비휘발성이여야 하고, 일부분은 동적으로 메모리에 쓰기 위해 휘발성이여야 한다. 따라서 현재 메모리 중 0x000~0x7FF 는 비휘발성으로 ROM과 같이 동작하고, 0x800~0xFFF는 휘발성으로 RAM과 같이 동작한다고 가정한다.

위 가정과 같이 동작하는 것을 구현하기 위해, 먼저 전체 메모리를 2K씩 ROM과 RAM 영역으로 구분해야 한다. 그리고 DMUX를 사용하여 주소 12bit를 나누고 msb인 addr[11]을 control signal처럼 사용할 수 있다. 해당 비트가 0일 경우 ROM 영역에(Read Only), 1일 경우 RAM 영역에(R/W) 접근하는 방식으로 동작할 수 있다.

예를 들어, 메모리 WE = w8 AND addr[11] 처럼 원래 있던 제어선에 add[11]를 추가하여 논리를 구현할 수 있다.

이러한 설계를 통해 ROM 영역은 쓰기가 차단되어 명령어만 저장되는 공간으로 활용되며, RAM 영역은 실행 중 계산 결과를 저장하는 공간으로 활용된다. MU0 프로세서 구현 상에서 더욱 효율적인 메모리 관리와 효과적인 프로그램 구조 설계가 가능하다.

4. 고찰

먼저 이번 project0을 수행하며, 교육 목적으로 만들어진 MU0이지만, 컴퓨터구조 이론 시간에서 듣기만 했던 마이크로프로세서 구조에 대해 직접 논리 회로를 구현하고 프로그램 동작을 확인할 수 있어, 컴퓨터구조에 대한 전반적인 지식이 많이 늘었다. logisim-evolution을 사용하여 회로 하나하나를 구성하고 시뮬레이션하는 과정이 특히 큰 도움이 되었다.

TG, MUX, Latch, D-FF, Counter를 만들어보는 과제는, 지난 디지털논리회로 시간에서 익혔던 아주중요한 개념들을 다시 떠오르게 해서 굉장히 유익했다. w1~w8 control signal들을 연결하는 과제는 처음에는 어려웠지만, 제어선이 8개이고 명령어도 8개인 것과 디코더의 입력이 IR의 Opcode인 것을 통해 제어선은 각각의 명령어의 제어를 위해 사용되는 것을 제대로 깨닫고 나서 수월하게 진행할 수 있었다.

컴퓨터구조 이론 시간은 듣기 막막하고 어렵게 느껴질 때가 많았는데, 프로젝트를 통해 직접 구조를 설계해보니까 나름의 재미도 있었다. RAM 파일을 로드하고 클럭 신호를 주며 프로그램 동작을 실행하면서, JGE 008 명령에서 점프에 실패하는 것에 회로 문제가 생긴 줄 알았으나, ACC값이 0x8000, 즉 2진수로 1000 0000 0000 0000이고 msb가 1이기 때문에, 2의 보수를 적용하면 음수여서 조건 검사에 실패한다는 것을 이해하게 되었다. 제안서의 마지막 부분에서 RAM과 ROM의 특징과 차이를 이해할 수 있었고, 실제 시스템의 메모리를 그럼 어떻게 설계하면 좋을까라는 궁금증을 떠오르고 해결법을 스스로 찾아가는 과정도 흥미로웠다.