컴퓨터구조 프로젝트 보고서

Project 2

MIPS Multi-Cycle CPU Implementation

수업 명: 컴퓨터구조

담당 교수: 이성원 교수님

학과: 컴퓨터정보공학부

학번: 2023202070

이름: 최현진

제출일: 2025.05.12

1. Introduction

MIPS Multi-Cycle CPU에서, 기본 명령어인 LW, SW, ORI, ADD, SUB, J, LUI, LHI, LLO는 이미 구현되어 있으며, 본 프로젝트에서는 추가적으로 AND, NOR, ADDI, SLTU, SRL, SH, LB, BNE, BGEZ, JALR 명령어를 설계한다.  
설계는 마이크로프로그래밍 기반의 FSM 구조를 따르며, DISP\_ROM과 MICRO\_ROM을 작성하여 명령어 디코딩 및 제어 신호 생성을 수행한다. . FSM은 상태 0(FETCH), 상태 1(DECODE)을 공통으로 사용하며, 이후 명령어 별로 정의된 상태를 통해 실행 단계를 분리하여 구현한다. 실행 결과는 시뮬레이터 상의 testbench waveform과 비교하여 기능을 검증한다.

2. Assignment

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| <Micro-Instruction 의 Field 구분 및 Field 용도>  1. AND  두 레지스터의 비트 단위 AND 연산 결과를 레지스터에 저장한다, R-type  Syntax: f $d, $s, $t  Operation: $d = $s & $t   |  |  | | --- | --- | | undef | raddr | | 0 | 00000010 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | 000 | 000 | 00000 | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | 0x | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_000\_00000\_0x\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x02: AND execution  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  EXTmode: x, imm값을 extend할 필요가 없다.  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 000, ALU 입력으로 register B를 사용한다.**  **ALUop: 00000, bitwise AND**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | 01 | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | 000 | 1 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 00 |  |   x\_x\_0\_xxx\_0\_01\_000\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_00 // 0x03: AND completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  **RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.**  **RegDatSel: 000, 레지스터 파일에 ALU 결과값을 쓴다.**  **RegWrite: 1, 레지스터 파일에 쓴다.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 00, Next State = 0**  $s, $t 레지스터를 읽고 ALU에서 bitwise AND연산한 결과가 $d 레지스터에 저장된다. PC=PC+4이다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2. NOR  두 레지스터의 비트 단위 NOR 연산 결과를 레지스터에 저장한다, R-type  Syntax: f $d, $s, $t  Operation: $d = ~($s | $t)   |  |  | | --- | --- | | undef | raddr | | 0 | 00000100 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | 000 | 000 | 00010 | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | 0x | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_000\_00010\_0x\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x04: NOR execution  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  EXTmode: x, imm값을 extend할 필요가 없다.  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 000, ALU 입력으로 register B를 사용한다.**  **ALUop: 00010, bitwise NOR**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | 01 | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | 000 | 1 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 00 |  |   x\_x\_0\_xxx\_0\_01\_000\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_00 // 0x05: NOR completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  **RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.**  **RegDatSel: 000, 레지스터 파일에 ALU 결과값을 쓴다.**  **RegWrite: 1, 레지스터 파일에 쓴다.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 00, Next State = 0**  $s, $t 레지스터를 읽고 ALU에서 bitwise NOR 연산한 결과가 $d 레지스터에 저장된다. PC=PC+4이다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3. ADDI  imm값을 sign extend하여 $s와 더한 결과를 $t에 저장한다, I-type  Syntax: o $t, $s, i  Operation: $t = $s + SE(i)   |  |  | | --- | --- | | undef | raddr | | 0 | 00000110 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | 1 | 000 | 011 | 00100 | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | 0x | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_1\_000\_011\_00100\_0x\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x06: ADDI execution  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  **EXTmode: 1, imm값을 sign extend한다.**  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 011, ALU 입력으로 sign extend unit 결과값(imm값 se)을 사용한다.**  **ALUop: 00100, a + b**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | 00 | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | 000 | 1 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 00 |  |   x\_x\_0\_xxx\_0\_00\_000\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_00 // 0x07: ADDI completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  **RegDst: 00, 결과값을 rt에 쓸 것이므로 목적지 레지스터로 rt를 선택한다.**  **RegDatSel: 000, 레지스터 파일에 ALU 결과값을 쓴다.**  **RegWrite: 1, 레지스터 파일에 쓴다.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 00, Next State = 0**  $s를 읽고, imm값을 sign extend하여 ALU에서 $s와 더한 결과를 $t에 저장한다 PC=PC+4이다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4. SLTU  Set Less Than Unsigned, unsigned 기준으로 $s < $t 비교 결과를 $d에 저장한다, R-type  Syntax: f $d, $s, $t  Operation: $d = ($s < $t)   |  |  | | --- | --- | | undef | raddr | | 0 | 00001000 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | 000 | 000 | 10001 | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | 0x | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_000\_10001\_0x\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x08: SLTU execution  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  EXTmode: x, imm값을 extend할 필요가 없다.  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 000, ALU 입력으로 register B를 사용한다.**  **ALUop: 10001, Unsigned SLT**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | 01 | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | 000 | 1 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 00 |  |   x\_x\_0\_xxx\_0\_01\_000\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_00 // 0x09: SLTU completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  **RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.**  **RegDatSel: 000, 레지스터 파일에 ALU 결과값을 쓴다.**  **RegWrite: 1, 레지스터 파일에 쓴다.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 00, Next State = 0**  $s, $t 레지스터를 읽고 ALU에서 Unsigned SLT 연산한 결과가 $d 레지스터에 저장된다. PC=PC+4이다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5. SRL  Shift Right Logical: shamt만큼 logical shift한 값을 레지스터에 저장한다. R-type  Syntax: f $d, $t, sa  Operation: $d = $t >> a   |  |  | | --- | --- | | undef | raddr | | 0 | 00001010 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | 000 | 000 | 01110 | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | 00 | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_000\_01110\_00\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x0a: SRL execution  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  EXTmode: x, imm값을 extend할 필요가 없다.  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 000, ALU 입력으로 register B를 사용한다.**  **ALUop: 01110, b >> a**  **ALUctrl: 00, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= 0 (shift = shift amount)**  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | 01 | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | 000 | 1 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 00 |  |   x\_x\_0\_xxx\_0\_01\_000\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_00 // 0x0b: SRL completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  **RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.**  **RegDatSel: 000, 레지스터 파일에 ALU 결과값을 쓴다.**  **RegWrite: 1, 레지스터 파일에 쓴다.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 00, Next State = 0**  $t를 읽고, ALU control을 통해 shamt만큼 ALU에서 logical shift한다. ALU 결과값을 $d에 저장한다. PC=PC+4이다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6. SH  Store Halfword: MEM [$s + i]에 레지스터 $t의 하위 2바이트 값을 쓴다, I-type  Syntax: o $t, i ($s)  Operation: MEM [$s + i]:2 = LH ($t)   |  |  | | --- | --- | | undef | raddr | | 0 | 00001100 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | 1 | 000 | 011 | 00100 | | ALUctrl | Branch | PCsource | PCwrite | StateSel |  | | 0x | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_1\_000\_011\_00100\_0x\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x0c: SH execution  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  **EXTmode: 1, imm값을 sign extend한다.**  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 011, ALU 입력으로 sign extend unit 결과값(imm값 se)을 사용한다.**  **ALUop: 00100, a + b**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**     |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | 1 | x | 1 | 010 | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsource | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 00 |  |   1\_x\_1\_010\_0\_xx\_xxx\_0\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_00 // 0x0d: SH memory access  IorD: 1, DM에 접근함  MemRead: x, 메모리를 읽지 않음  MemWrite: 1, 메모리에 쓴다.  **DataWidth: 010, 16-bit Halfword를 DM에 쓴다.**  IRwrite: 0, IR에 쓰지 않음  **RegDst: xx, 레지스터 파일에 쓰지 않음.**  **RegDatSel: xxx, 레지스터 파일에 쓰지 않음.**  **RegWrite: 0, 레지스터 파일에 쓰지 않음.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 00, Next State = 0**  $t, $s를 읽고, 메모리 [$s + i]에 $t의 하위 2바이트 값을 쓴다. imm값을 sign extend하여 $s와 ALU에서 주소 연산한다. PC=PC+4이다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7. LB  Load Byte: MEM [$s + i]의 1바이트 데이터를 가져와서 sign extend 후 $t에 저장한다, I-type  Syntax: o $t, i ($s)  Operation: $t = SE (MEM [$s + i]:1)   |  |  | | --- | --- | | undef | raddr | | 0 | 00001110 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | 1 | 000 | 011 | 00100 | | ALUctrl | Branch | PCsource | PCwrite | StateSel |  | | 0x | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_1\_000\_011\_00100\_0x\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x0e: LB execution  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  **EXTmode: 1, imm값을 sign extend한다.**  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 011, ALU 입력으로 sign extend unit 결과값(imm값 se)을 사용한다.**  **ALUop: 00100, a + b**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | 1 | 1 | 0 | 111 | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsource | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 11 |  |   1\_1\_0\_011\_0\_xx\_xxx\_0\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x0f: LB memory access  **IorD: 1, DM에 접근함**  **MemRead: 1, 메모리를 읽는다**  **MemWrite: 0, 메모리에 쓰지 않음.**  **DataWidth: 111, 8-bit Byte with Sign Ext를 DM에서 읽는다.**  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음.  RegDatSel: xxx, 레지스터 파일에 쓰지 않음.  **RegWrite: 0, 레지스터 파일에 쓰지 않음.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | 00 | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | 001 | 1 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsource | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 00 |  |   x\_x\_0\_xxx\_0\_00\_001\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_00 // 0x10: LB write-back  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  **RegDst: 00, 결과값을 rt에 쓸 것이므로 목적지 레지스터로 rt를 선택한다.**  **RegDatSel: 001, 레지스터 파일에 MDR 값을 쓴다.**  **RegWrite: 1, 레지스터 파일에 쓴다.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 00, Next State = 0**  $s를 읽고 MEM [$s + i]의 1바이트 데이터를 가져와서 sign extend 후 $t에 저장한다. imm값을 sign extend하여 $s와 ALU에서 주소 연산한다. PC=PC+4이다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8. BNE  Branch Not Equal: $s != $t이면 branch한다, I-type  Syntax: o $s, $t, label  Operation: l if ($s != $t) pc += i << 2   |  |  | | --- | --- | | undef | raddr | | 0 | 00010001 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | 000 | 000 | 00110 | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | 0x | 101 | 01 | 1 | 00 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_000\_00110\_0x\_101\_01\_1\_xxxxxxxx\_00 // 0x11: BNE completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  EXTmode: x, imm값을 extend할 필요가 없다. (이전 state에서 branch addr 미리 계산하기 때문)  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 000, ALU 입력으로 register B를 사용한다.**  **ALUop: 00110, a – b (두 레지스터 값 비교)**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  **Branch: 101, Branch if not equal**  **PCsrc: 01, PC source From ALUOut Register**  **PCwrite: 1: pc에 쓴다**.  **StateSel: 00, Next State = 0**  $s, $t값을 읽고, $s != $t 조건이 참이면 PC = PC + i << 2로 branch한다. imm값은 먼저 sign extend를 수행한다. ALU에서 뺄셈 연산을 통해 두 레지스터 값을 비교한 결과를 사용한다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9. BGEZ  I-type, RegImm-type: opcode=000001 고정하고, rt(=regimm)로 명령어를 구분한다. $s >=0이면 branch한다.  Syntax: r $s, label  Operation: if ($s >= 0) pc += i << 2   |  |  | | --- | --- | | undef | raddr | | 0 | 00010010 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | 000 | 010 | 10000 | | ALUctrl | Branch | PCsrc | PCwrite | StateSel |  | | 0x | 010 | 01 | 1 | 00 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_010\_10000\_0x\_010\_01\_1\_xxxxxxxx\_00 // 0x12: BGEZ completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  EXTmode: x, imm값을 extend할 필요가 없다. (이전 state에서 branch addr 미리 계산하기 때문)  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다.**  **ALUsrcB: 010, ALU 입력으로 0x0을 사용한다.**  **ALUop: 10000, set less than**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  **Branch: 010, Branch if not negative**  **PCsrc: 01, PC source From ALUOut Register**  **PCwrite: 1: pc에 쓴다**.  **StateSel: 00,** Next State = 0  $s를 읽고, $s >= 0 조건이 참이면 PC = PC + i << 2로 branch한다. imm값은 먼저 sign extend한다. ALU 입력으로 0을 사용하고 $s과의 set less than 연산을 통해 조건 검사를 수행한다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10. JALR  Jump And Link Register: PC를 $31에 저장하고 $s 레지스터가 가리키는 주소로 점프한다, R-type  Syntax: f labelR  Operation: $31 = pc; pc = $s   |  |  | | --- | --- | | undef | raddr | | 0 | 00010011 |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | 11 | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | 100 | 1 | x | xxx | xxx | xxxxx | | ALUctrl | Branch | PCsource | PCwrite | StateSel |  | | xx | xxx | xx | 0 | 11 |  |   x\_x\_0\_xxx\_0\_11\_100\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x13: JALR register write  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  **RegDst: 11, PC를 $31에 쓸 것이므로 목적지 레지스터로 $31를 선택한다.**  **RegDatSel: 100, PC를 register file에 쓴다.**  **RegWrite: 1, 레지스터 파일에 쓴다.**  EXTmode: x, imm값을 extend할 필요가 없다.  ALUsrcA: xxx, ALU 연산하지 않음  ALUsrcB: xxx, ALU 연산하지 않음  ALUop: xxxxx, ALU 연산하지 않음  ALUctrl: xx, ALU 연산하지 않음  Branch: xxx, branch 수행하지 않음  PCsrc: xx, pc에 쓰지 않음  PCwrite: 0: pc에 쓰지 않음  **StateSel: 11, Next State = Current State + 1**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | IorD | MemRead | MemWrite | DataWidth | IRwrite | RegDst | | x | x | 0 | xxx | 0 | xx | | RegDatSel | RegWrite | ExtMode | ALUsrcA | ALUsrcB | ALUop | | xxx | 0 | x | 000 | 010 | 00100 | | ALUctrl | Branch | PCsource | PCwrite | StateSel |  | | 0x | 000 | 00 | 1 | 00 |  |   x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_010\_00100\_0x\_000\_00\_1\_xxxxxxxx\_00 // 0x14: JALR completion  IorD: x, IM 또는 DM에 접근하지 않음  MemRead: x, 메모리를 읽지 않음  MemWrite: 0, 메모리에 쓰지 않음  DataWidth: xxx, IM 또는 DM에 접근하지 않음  IRwrite: 0, IR에 쓰지 않음  RegDst: xx, 레지스터 파일에 쓰지 않음  RegDatSel: xxx, 레지스터 파일에 쓰지 않음  RegWrite: 0, 레지스터 파일에 쓰지 않음  EXTmode: x, imm값을 extend할 필요가 없다.  **ALUsrcA: 000, ALU 입력으로 register A를 사용한다. ($rs)**  **ALUsrcB: 010, ALU 입력으로 0x0을 사용한다.**  **ALUop: 00100, a+b (PC로 ALUout을 사용하기 위해 $rs + 0 -> $rs)**  ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)  **Branch: 000, jump**  **PCsrc: 00, PC source From ALU output**  **PCwrite: 1: pc에 쓴다**.  **StateSel: 00, Next State = 0**  PC를 $31에 저장하고 $s가 가리키는 주소로 점프한다 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| <구현한 Multi Cycle CPU FSM Diagram>     |  | | --- | | State 0: Instruction Fetch  메모리에서 명령어를 가져오고, PC = PC + 4 수행  IorD = 0 (Instruction Memory)  MemRead = 1  IRwrite = 1  ALUsrcA = 011 (PC)  ALUsrcB = 001 (4)  ALUop = 00100 (PC + 4)  PCsrc = 00 (ALU output)  PCwrite = 1 |  |  | | --- | | State 1: Instruction Decode / Register Read / Branch Address 계산  $rs, $rt 읽기, Branch 명령어일 경우 분기 주소 계산 수행  ALUsrcA = 011 (PC)  ALUsrcB = 100 (sign-extend(IR[15:0]) << 2)  ALUop = 00100 (PC + offset) |  |  | | --- | | State 2: Memory Address Computation (LB, SH, ADDI)  Load/Store/ADDI 명령어에서 주소 계산 (rs + imm)  EXTmode = 1 (Sign-extend)  ALUsrcA = 000 (A 레지스터 = rs)  ALUsrcB = 011 (sign-extended imm)  ALUop = 00100 (a + b) |  |  | | --- | | State 3: Memory Access (LB)  메모리에서 byte 단위 읽기:  IorD = 1 (Data Memory)  MemRead = 1  DataWidth = 111 (8-bit, Sign-extend) |  |  | | --- | | State 4: Memory Write-Back (LB)  MDR 값을 $rt에 저장  RegDst = 00 (write to $rt)  RegDatSel = 001 (MDR)  RegWrite = 1 |  |  | | --- | | State 5: Memory Access (SH)  메모리에 halfword 저장  IorD = 1 (Data Memory)  MemWrite = 1  DataWidth = 010 (16-bit halfword) |      |  | | --- | | State 6: ADDI Write-Back  ALU 결과를 $rt에 저장  RegDst = 00  RegDatSel = 000 (ALUOut)  RegWrite = 1 |  |  | | --- | | State 7: Execution (AND, NOR, SLTU, SRL)  R-type 명령어의 실제 연산 수행  ALUsrcA = 000 (A 레지스터 = rs)  ALUsrcB = 000 (B 레지스터 = rt)  ALUop = 각 명령어에 따라 다름 | | State 8: ALU Write-Back (AND, NOR, SLTU, SRL)  ALU 결과를 $rd에 저장  RegDst = 01  RegDatSel = 000  RegWrite = 1 |  |  | | --- | | State 9: BNE Completion  $rs ≠ $rt 조건 판단 → PC update  ALUsrcA = 000 (rs)  ALUsrcB = 000 (rt)  ALUop = 00110 (a - b)  Branch = 101 (bne)  PCsrc = 01 (ALUOut), PCwrite = 1 |  |  | | --- | | State 10: BGEZ Completion  $rs ≥ 0 조건 판단 → PC update  ALUsrcA = 000 (rs)  ALUsrcB = 010 (0)  ALUop = 00110 (a - b)  Branch = 111 (bgez)  PCsrc = 01, PCwrite = 1 |  |  | | --- | | State 11: JALR Register Write  $31 ← PC + 4 수행  RegDst = 11 (write to $31)  RegDatSel = 100 (write PC)  RegWrite = 1 |  |  | | --- | | State 12: JALR Completion  PC ← $rs 수행  ALUsrcA = 000 (rs)  ALUsrcB = 010 (0)  ALUop = 00100 (a + b)  PCsrc = 00 (ALU output)  PCwrite = 1 | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| <시뮬레이션 결과와 예상 결과 비교 분석>   |  | | --- | | <초기 M\_TEXT\_SEG.txt>  001111\_00000\_00010\_0001001000110100 // lui $2, 0x1234  001101\_00010\_00011\_0101011001111000 // ori $3, $2, 0x5678  001111\_00000\_00100\_0001000100100010 // lui $4, 0x1122  001101\_00100\_00101\_0011001101000100 // ori $5, $4, 0x3344  <실행 결과>  $2 = 0x12340000  $3 = $2 | 0x5678 = 0x12340000 | 0x00005678 = 0x12345678  $4 = 0x11220000  $5 = 0x11220000 | 0x00003344 = 0x11223344 |   1. and, nor, sltu, srl, addi   |  | | --- | | 000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5  001000\_00011\_01000\_0000000000000001 // addi $8, $3, 1 000000\_00010\_00101\_01001\_00000\_101011 // sltu $9, $2, $5  000000\_00000\_00011\_01010\_00100\_000010 // srl $10, $3, 4 |   먼저 기본 ALU 연산 동작을 하는 명령어들의 시뮬레이션을 진행했다. 레지스터 연산을 하는 R-type 명령어 and, nor, sltu, srl과 imm값을 사용하여 연산하는 I-type 명령어 addi의 검증을 수행했다. 다음은 사용한 명령어와 그 동작이다.      먼저 add 명령어를 읽고 수행 결과, $3, $5 두 레지스터를 읽고 $6에 $3 & $5 연산 결과인 0x12345678 & 0x11223344 = 0x10201240의 값이 저장되었다. PC = PC + 4이다.  <o\_state>  002: add execution  003: add completion, register file에 쓴다    다음으로 nor 명령어를 읽고 수행 결과, $3, $5 두 레지스터를 읽고 $7에 ~($3 | $5) 연산 결과인 ~(0x1336777C) = 0xECC98883의 값이 저장되었다. PC = PC + 4이다.  <o\_state>  004: nor execution  005: nor completion, register file에 쓴다    addi 명령어를 읽고 수행한 결과, $3과 imm값인 1을 읽은 후 $8에 0x12345678 + 1 연산 결과인 0x12345679가 저장되었다. PC = PC + 4이다.  <o\_state>  006: addi execution  007: addi completion, register file에 쓴다    sltu 명령어를 읽고 수행 결과, $2와 $5를 읽고 비교하여 0x12340000 > 0x11223344이므로 $9에 unsigned 기준 연산 결과인 0이 저장되었다. PC = PC + 4이다.  <o\_state>  008: sltu execution  009: sltu completion, register file에 쓴다    srl 명령어를 읽고 연산 결과, $3와 shamt 값을 읽은 후 $10에 $3를 shamt만큼 shift한 $3 >> 4 연산 결과인 0x01234567이 저장되었다. PC = PC + 4이다.  <o\_state>  00A: srl execution  00B: srl completion, register file에 쓴다  명령어 수행 후 저장된 reg\_dump.txt 파일이다.    2. sh, lb  다음으로 메모리 접근을 수행하는 두 명령어 sh, lb의 시뮬레이션을 수행하였다.   |  |  | | --- | --- | | 101001\_00000\_00011\_0000000000000100 // sh $3, 4($0)  100000\_00000\_01011\_0000000000000100 // lb $11, 4($0) |  |       sh 명령어를 읽고 수행한 결과, MEM[$0 + 4]에 $3의 하위 2바이트 값을 쓸 것이다. ALU에서0x0000에 imm 값인 4이 더해져 메모리 주소를 계산한다. $3=0x12345678 중 리틀 앤디언 방식에 따라 메모리 주소 0x0801에 0x5678이 저장되었다. sh는 10개 명령어 중 유일하게 데이터 메모리에 쓰는 명령어이기에 RegWrite는 0, MemWrite는 1이다. PC = PC + 4이다.  <o\_state>  00C: sh execution  00D: sh memory access    다음으로 lb 명령어를 읽고 수행 결과, MEM[$0 + 4]의 하위 1바이트 데이터를 가져와 sign extend 후 $12에 저장할 것이다. ALU에서 0x0000에 imm 값인 1이 더해진 결과인 0x0004에는 이전 sh 명령어 수행 결과 0x78이 저장되어 있다. write register인 $11에 0x78이 저장되었다. 따라서 RegWrite는 1, MemWrite는 0이다. PC = PC + 4이다.  <o\_state>  00E: lb execution  00F: lb memory access  010: lb write-back  명령어 수행 후 저장된 reg\_dump.txt 파일이다.    3. bne, bgez, jalr  다음으로 branch와 jump 명령어들인 bne, bgez, jalr을 각각 검증했다.   |  | | --- | | 000101\_00010\_00101\_0000000000000001 // bne $2, $5, 1  000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5 -> 실행x  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5 -> 실행o |     먼저 bne 명령어를 읽고 수행한 결과, $2와 $5를 읽었고 ALU에서 a-b 연산을 수행한 결과 0이아니므로 branch조건인 0x12340000 =/ 0x11223344를 만족한다. 계산한 pc 주소는 pc = pc+4+(imm<<2)이므로, o\_pc 값이 0x002C + 0x0004 + 0x0004 = 0x0034인 것을 확인했다.  따라서 다음 클럭에 bne 바로 다음 명령어인 and(pc 30)가 아닌 그 다음 명령어인 nor(state 0,1,4,5, oc 34)이 실행되는 것을 확인했다.  <o\_state>  011: BNE completion   |  | | --- | | 001111\_00000\_01100\_0000000000000000 // lui $12, 0x0000  001101\_01100\_01101\_0000000000000001 // ori $13, $12, 0x0001  000001\_01101\_00001\_0000000000000001 // bgez $13, 1  000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5 -> 실행x  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5 -> 실행o |     다음으로 bgez 명령어 검증을 위해 일단 lui와 ori 명령어를 통해 $13에 0x0001을 저장했다. bgez 명령어를 읽고 수행한 결과, $13를 읽고 0과 set less than ALU 연산한 결과 조건인 0x00000001 >= 0을 만족하므로 branch한다. imm값은 1이고, 계산한 pc 주소는 pc = pc+4+(imm<<2)이므로, o\_pc 값이 decimal로 표현했을 때 40 + 4 + 4 = 48인 것을 확인했다.  따라서 다음 클럭에 bgez 바로 다음 명령어인 and가 아닌 그 다음 명령어인 nor(state 0,1,4,5)이 실행되는 것을 확인했다.  <o\_state>  012: BGEZ completion   |  | | --- | | 000000\_01111\_00000\_11111\_00000\_001001 // jalr $31, $15  000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5 -> 실행x  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5 -> 실행o |     jalr 명령어를 읽고 수행 결과, 13 state에서 $31에는 PC+4=0x0058가 저장된다. 14 state에서 PCwrite가 활성화되고 PC에는 레지스터에서 읽은 $15= 0x0000005C가 저장된다. 따라서 다음 클럭에 bgez(pc 54) 다음 명령어인 and(pc 58)가 아닌 그 다음 명령어인 nor(state 0,1,4,5, pc 5C)이 실행되는 것을 확인했다.  <o\_state>  013: JALR register write  014: JALR completion  jalr 명령어 수행 후 저장된 reg\_dump.txt 파일이다.    이렇게 10개의 명령어 시뮬레이션을 마쳤고, M\_TEXT\_SEG.txt의 각 명령어 필드를 고려하여 동작 예상을 주석과 함께 작성하였다. 시뮬레이션 결과는 예상 결과와 같았다. 다음은 모든 명령어 검증에 사용할 수 있는 전체 M\_TEXT\_SEG.txt이다.   |  | | --- | | 001111\_00000\_00010\_0001001000110100 // lui $2, 0x1234  001101\_00010\_00011\_0101011001111000 // ori $3, $2, 0x5678  001111\_00000\_00100\_0001000100100010 // lui $4, 0x1122  001101\_00100\_00101\_0011001101000100 // ori $5, $4, 0x3344  000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5  001000\_00011\_01000\_0000000000000001 // addi $8, $3, 1  000000\_00010\_00101\_01001\_00000\_101011 // sltu $9, $2, $5  000000\_00000\_00011\_01010\_00100\_000010 // srl $10, $3, 4  101001\_00000\_00011\_0000000000000100 // sh $3, 4($0)  100000\_00000\_01011\_0000000000000100 // lb $11, 4($0)  000101\_00010\_00101\_0000000000000001 // bne $2, $5, 1  000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5 -> 실행x  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5 -> 실행o  001111\_00000\_01100\_0000000000000000 // lui $12, 0x0000  001101\_01100\_01101\_0000000000000001 // ori $13, $12, 0x0001  000001\_01101\_00001\_0000000000000001 // bgez $13, 1  000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5 -> 실행x  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5 -> 실행o  001111\_00000\_01110\_0000000000000000 // lui $14, 0x0000  001101\_01110\_01111\_0000000001011100 // ori $15, $14, 0x005C = 92  000000\_01111\_00000\_11111\_00000\_001001 // jalr $31, $15  000000\_00011\_00101\_00110\_00000\_100100 // and $6, $3, $5 -> 실행x  000000\_00011\_00101\_00111\_00000\_100111 // nor $7, $3, $5 -> 실행o  000000\_00000\_00000\_00000\_00000\_001101 // break |   위 시뮬레이션 시 얻은 전체 reg\_dump.txt이다. |

|  |
| --- |
| <Micro-Instruction 반복된 사용에 대한 분석>  이번 프로젝트에서 여러 명령어를 Microprogram 구조로 구현하면서, 제어 신호가 동일한 state를 여러 명령어에서 공통적으로 재사용할 수 있는 경우가 있었다.  1. R-type 명령어들의 공통 Write-Back  and, nor, sltu, srl 등은 모두 R-type 명령어로서, 연산 결과를 레지스터 파일의 $rd에 저장해야 한다.  이 명령어들은 ALU를 통해 연산한 후 공통적으로 동일한 write-back 동작을 수행하므로, Completion State에서 다음과 같은 제어 신호 조합을 공유한다.  RegDst = 01 // 목적지 레지스터 rd  RegDatSel = 000 // ALU 결과 사용  RegWrite = 1 // 레지스터에 씀  반면, addi는 I-type 명령어이므로 결과를 $rd가 아닌 $rt에 저장해야 하며, 이 때문에 RegDst = 00이 되어야 한다. 따라서 레지스터에 쓴다는 같은 동작을 하더라도 목적지 레지스터 필드 값이 다르기 때문에 별도의 write-back 상태를 필요로 한다.  2. I-type 명령어들의 공통 Execution  addi, sh, lb 세 명령어는 모두 ALU에서 동일한 연산을 수행한다.  ALUsrcA = A // rs  ALUsrcB = 011 // sign-extended immediate  ALUop = 00100 // a + b, 주소 계산  이라는 구성으로 rs + imm 주소 계산을 수행한다.  addi는 레지스터에 써넣을 값을 얻을 목적으로, sh와 lb는 메모리 주소를 얻을 목적으로 계산을 수행한다. 단, 이후 단계인 reg write 또는 memory access는 서로 다르므로 각각 별도의 상태로 분리된다.  이처럼 기능적으로 동일한 명령어의 일부 단계를 공통화함으로써,  ROM\_MICRO의 중복을 줄이고, 유지보수 시 같은 상태를 재활용할 수 있으며, 새로운 명령어 추가 시 참고할 수 있는 명확한 템플릿을 제공한다.  지금 구조에서는 같은 동작이어도 state 위치가 다르면 micro-instruction를 복사할 수밖에 없기 때문에 반복은 불가피하다. |

|  |
| --- |
| <문제점 및 해결 방향>  1.  이번 프로젝트에서 JALR 명령어를 구현하는 과정에서 순차적으로 문제를 마주하였다.  처음에는 JALR이 점프 명령이기 때문에 IF, DE, JUMP의 세 개 state로 구성해야 하는지 고민하였다.  하지만 PCsrc에는 $rs를 직접 PC로 설정하는 필드가 없었고, 대신 ALUOut 값을 PC로 사용하는 필드만 존재하였다.  따라서 $rs 값을 가져와서 PC에 저장하려면, decode 단계에서 $rs를 읽은 후 ALU에서 $rs + 0 연산을 수행하여 ALUOut에 저장하고, 이를 통해 점프 주소로 사용해야겠다는 결론을 내렸다.  초기에는 따라서 다음과 같이 하나의 state에서 ALU 연산, 레지스터 파일 쓰기, PC 쓰기를 모두 처리하는 방식으로 마이크로 명령을 구성하였다:  x\_x\_0\_xxx\_0\_11\_100\_1\_x\_000\_010\_00100\_0x\_000\_00\_1\_xxxxxxxx\_00 // 0x13: jalr completion  하지만 이는 ALU 연산과 reg, pc 두 개의 write 동작이 동시에 수행되어 구조상 무리가 생길 수 있었고, 무엇보다 JALR은 R-type 명령이므로 기존 R-type 명령어와 동일하게 두 개의 상태로 나누는 것이 타당하다는 점에서 재고하게 되었다. 최종적으로는 ‘$31 ← PC + 4’를 수행하는 상태와 ‘PC ← $rs’를 수행하는 상태로 나누어 총 두 개의 state로 구현하였다.  ALU 연산은 단순히 `$rs + 0`을 수행하므로 PC write와 동일 사이클에 배치해도 성능에 영향이 없다고 판단하였다. 이렇게 명령어 유형에 따라 일관성 있게 FSM을 구성함으로써, 유지보수 및 디버깅에도 용이한 구조를 갖추게 되었다.  최종 마이크로 명령은 다음과 같다:  x\_x\_0\_xxx\_0\_11\_100\_1\_x\_xxx\_xxx\_xxxxx\_xx\_xxx\_xx\_0\_xxxxxxxx\_11 // 0x13: JALR register write  x\_x\_0\_xxx\_0\_xx\_xxx\_0\_x\_000\_010\_00100\_0x\_000\_00\_1\_xxxxxxxx\_00 // 0x14: JALR completion  2.  jalr 명령어를 구현할 때 PCsrc를 어떤 값으로 설정해야 할지 혼란스러웠다. bne나 bgez 같은 분기 명령어는 PCsrc = 01을 사용하는데, jalr에서는 왜 00을 써야 하는지 명확하지 않았다. Branch 명령어들은 분기 주소를 미리 ALUOut register에 저장해두고 이후 단계에서 꺼내 쓰는 방식이지만, jalr는 ALU에서 계산한 $rs + 0 값을 바로 PC에 써야 한다. 따라서 PCsrc = 00(ALU 결과)을 사용해야 하며, 이는 명령어 동작 방식에 따른 필연적인 구조 차이라는 점을 인식하고 설계에 반영하여 해결했다.  3.  bne, bgez 명령어를 구현하면서 처음에는 EXTmode를 어떻게 설정해야 할지 혼란스러웠다. 싱글 사이클 CPU에서는 항상 imm값을 확장하는 방식이었기 때문에, 멀티 사이클에서도 동일하게 확장해야 한다고 생각했다. 멀티 사이클 구조에서는 DECODE 단계(state 1)에서 이미 branch 주소 계산을 수행한다. 따라서 이후 completion 단계에서는 imm값 확장이 불필요하므로, 해당 단계에서는 EXTmode = x로 설정해도 문제가 없다는 것을 이해하고 적용하여 해결하였다. |

3. 고찰

이번 프로젝트는 이전 프로젝트 1에서 구현했던 10개 명령어를 그대로 사용하는 과제였기 때문에, 명령어에 대한 이해가 이미 충분히 되어 있어서 훨씬 더 재미있게 수행할 수 있었다. 달라진 점은 싱글 사이클 CPU에서 멀티 사이클 CPU로 넘어갔다는 점인데, 처음에는 각각의 사이클들이 어떤 역할을 하는지 헷갈리기도 했지만, 하나하나 단계를 나눠서 다시 이해하면서 구현을 진행하니 점점 더 확실하게 구조가 잡혀갔다.

싱글 사이클 구조에서는 존재하지 않았던 MDR, ALUOut 같은 레지스터가 멀티 사이클 구조에서 추가되면서 그 흐름을 놓치지 않고 따라가는 것이 중요했다. 특히 MUX 선택선이 많아지고 control signal이 다양해진 만큼, FSM의 동작을 정확히 그려가며 설계해야 했다.