MIPS Single Cycle Microprocessor Implementation Report

차 호 현(#32224560), outcider112@dankook.ac.kr  
Undergraduate Student in Mobile System Engineering, Dankook University

Code Repository: https://github.com /HOchacha/Computer-Architecture

**1. Introduction**

컴퓨터 구조는 ‘컴퓨터 시스템이 어떻게 구성되었는가’를 다루는 가장 근본적인 이론이다. 현대의 컴퓨터는 폰 노이만 구조로부터 발전시켜, CPU 내부에 CU와 ALU, 레지스터 등 연산을 수행하기 위한 장치들이 동작한다. 하지만, 복잡한 장치를 다루기 위해 SW에 부품들을 외우게 하는 것은 SW 개발자에게 복잡한 작업이다. ISA는 이러한 복잡함을 감출 수 있는 추상화를 가능하게 하여 시스템을 다루기 쉽게 한다. HW에서는 ISA에서 정의한 동작을 수행할 수 있도록 마이크로 아키텍처를 구현하고, SW에서는 HW가 어떻게 구현되었는지 상관없이 ISA로 정의된 명령어를 사용함에 따라서 SW를 작성할 수 있게 되었다.

텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명

그림 1. ISA의 인터페이스 역할

ISA에 따라서 HW를 구현하기 위해, 마이크로 아키텍처를 구현할 때, ISA에서 지시한 동작과 명령어 구조 등에 따라, 필요한 컴포넌트가 달라질 수 있다. 또한, ISA에서 지시한 동작에만 충실하게 따른다면, 실제 동작은 같은 동작을 수행함에도 상이할 수 있다. 이에 주의하여 구현이 필요하다.

구현한 프로젝트는 MIPS Greet Sheet에 정의된 동작과 실제 컴포넌트를 모듈 별로 분리하여 객체 지향적인 접근으로 프로그램을 작성하였다. 가령, ISA가 입력되고 나서, 분리되어 있는 CU와 ALU가 동작한다. CU는 읽어들인 명령어에서, opcode에 따라 ALU가 의도된 동작을 할 수 있도록 신호를 준다.

MIPS ISA에 따라서 구현된 HW 시스템은 MIPS가 제공하는 명령어를 기반으로 프로그래밍을 할 수 있다. SW 개발자는 C언어 혹은 파이썬 등의 고급 언어를 컴파일러 혹은 인터프리터를 통해서, 기계어로 변환을 한다. 해당 프로젝트에서는 MIPS 아키텍처를 따르므로 컴파일러 내에는 MIPS 어셈블러가 포함되며, 1차적으로 컴파일러가 어셈블리 언어로 변환하면 최종적으로 어셈블러가 변환된 어셈블리 코드를 기계어로 변환한다.

**2. Design Structure of a Single Cycle Program**

싱글 사이클 머신을 구현하기 위해서 따르는 ISA는 MIPS이다. MIPS(Micoroprocessor without Interlocked Pipeline Stages)는 RISC로 분류되는 아키텍처로, 32비트의 고정 길이의 명령어와 명령어 타입마다 지정되어 있는 필드에 따라서 각 컴포넌트를 동작시킨다. 모든 레지스터는 32비트 크기를 가지고 있다.

**R 타입 명령어**

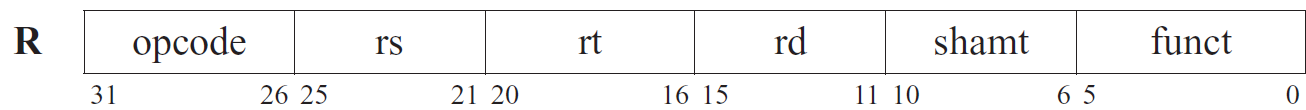


그림 2. R 타입 명령어 구조

opcode : 명령어 구분 (R타입에서는 전부 0으로 지정)

rs : ALU의 operand1로 사용할 레지스터 번호

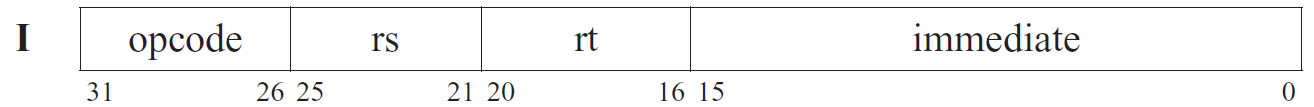
rt : ALU의 operand2로 사용할 레지스터 번호

rd : ALU의 결과 값을 저장할 레지스터 번호

shamt : 비트 시프트를 수행하는 경우 사용하는 필드

funct : ALU에서 수행할 동작을 구분하는 필드

**I 타입 명령어**



opcode : 명령어 구분

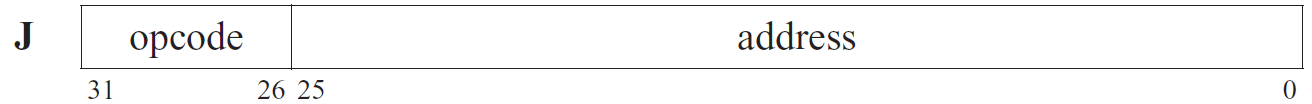
rs : ALU의 operand1로 사용할 레지스터 번호

rt : ALU의 결과 값을 저장하는 레지스터 번호

immediate : ALU의 operand2로 사용할 상수 값, 부호 확장을 통해서 32비트로 변환

단, BEQ, BNE, 등의 명령어는 위의 필드 설명과 상이한 부분이 있다. 따라서 해당 부분은 표 1.을 참고하여 동작을 파악한다.

**J 타입 명령어**



opcode : 명령어 구분 필드

address : 명령어를 나타내는 필드, 상수 값을 명시한다.

이 때, address를 나타내는 경우, MIPS가 4 바이트 길이의 명령어를 가지는 특징에 따라 주소 표현할 때 붙는 마지막 두 비트를 생략하여 표현할 수 있는 주소 범위를 넓게 할 수 있다.

**전체 아키텍처**

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

그림 3. 전체 마이크로 아키텍처 구현, 자세한 그림은 Appendix 참조

(전반적인 구조에 대한 설명) 해당 아키텍처는 하나의 클럭 신호에 따라서, 명령어 인출, 명령어 해독, 레지스터 접근, 메모리 접근, 결과 값 저장 단계가 순차적으로 실행된다. 해당 아키텍처를 구현하기 위해서, 실제 회로가 동작하는 것처럼 구현하였다. 다시 말해 모든 컴포넌트는 항상 클럭에 따라서 동작하되, enable 신호, control 신호가 없으면 의도된 동작을 하지 않도록 구현하였다. 해당 역할을 Control Unit이 opcode와 funct 신호에 따라서 Control Signal을 반환하도록 하였다.

**단계별 동작 방식**

**a. 명령어 인출 단계**

**텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명**

그림 4. 명령어 인출 단계

명령어 인출 단계에서는 PC로부터 실행할 명령어의 주소 값을 가지고 메모리에 접근하여 명령어를 CPU로 가지고 온다.

**b. 명령어 해독 / 레지스터 접근 단계**

**텍스트, 도표, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

그림 5. 명령어 해독 / 레지스터 접근 단계

명령어 해독 단계에서, 인출해온 명령어의 각 필드를 분리하여, 레지스터 혹은 필요한 컴포넌트에 연결한다. 특히, 해당 단계에서는 Control Unit이 opcode와 funct를 받아들여 전체 마이크로 아키텍처의 동작을 관리하게 된다.

**c. 명령어 처리 단계 및 메모리 엑세스 단계**

텍스트, 도표, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

그림 6. 명령어 처리 단계 및 메모리 액세스 단계

레지스터 접근 단계에서 접근할 레지스터의 번호를 지정한 후, 레지스터 컴포넌트는 해당 레지스터의 값을 반환한다. ALU는 두 개의 피연산자를 받아들여서, 연산을 수행한다. 이 때 ALU는 ALU ops 컴포넌트로부터 명령어에 따라서 의도된 동작을 수행할 수 있도록 신호를 받는다.

**d. 메모리 접근 단계**

**텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

그림 8. 메모리 접근 단계

메모리에 접근하기 위해서, ALU에서는 변위 지정 방식을 통해 접근할 메모리의 주소를 계산하도록 해야 한다 .

텍스트, 스크린샷, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

그림 7. 베이스(변위) 주소 지정 방식[[1]](#footnote-1)

변위 주소 지정 방식은 변위가 되는 레지스터의 값을 기준으로 Offset을 상수로 지정하여 메모리에 접근한다. 이를 위해서 ALU는 레지스터의 값과 부호 확장된 값을 가지고 접근할 메모리의 값을 구한다.

메모리에 저장할 데이터를 전달할 때는, rt 필드로 지정된 레지스터의 값을 저장하도록 한다. 이렇게 메모리에 작성할 데이터와 접근할 주소가 결정되면, Control Signal에 따라서 메모리가 의도된 동작을 수행할 수 있도록 한다.

**e. 결과 값 저장 단계 (레지스터 및 PC 값 업데이트)**

도표, 텍스트, 라인, 기술 도면이(가) 표시된 사진

자동 생성된 설명

최종적으로 ALU에서 연산된 결과 값과 메모리에서 가지고 온 값을 레지스터에 저장한다. 여기에서는 레지스터 컴포넌트에 registerWrite Control Singal에 따라서 레지스터의 값 업데이트 여부를 결정한다.

도표, 텍스트, 라인, 기술 도면이(가) 표시된 사진

자동 생성된 설명

그와 동시에, Jump 혹은 Branch와 같이 PC 값을 업데이트 한다.

해당 구조는 본 강의에서 제공된 Diagram을 기반으로 만들어졌으며, Control Signal을 처리하는 부분에서 opcode 처리하도록 한다. 모든 컴포넌트를 각각 함수로 구현하고, PC, 레지스터, 메모리와 같은 Architectural State는 전역 변수로 선언하여, 각 모듈에서 접근 가능하도록 하였다.

**지원되는 명령어**

해당 Single Cycle 아키텍처에서는 다음과 같은 명령어를 제공한다.

|  |  |  |  |
| --- | --- | --- | --- |
| 명령어 | 명령어 구조/타입 | 동작 | ALU 동작 |
| add | R | R[rd] = R[rs] + R[rt] | ADD |
| addiu | I | R[rt] = R[rs] + s\_imm | ADD |
| addu | R | R[rd] = R[rs] + R[rt] | ADD |
| addi | I | R[rt] = R[rs] + s\_imm | ADD |
| sub | R | R[rd] = R[rs] - R[rt] | SUB |
| subu | R | R[rd] = R[rs] – R[rt] | SUB |
| and | R | R[rd] = R[rs] & R[rt] | AND |
| andi | I | R[rt] = R[rs] & ZeroExtendImm | AND |
| or | R | R[rd] = R[rs] | R[rt] | OR |
| ori | I | R[rt] = R[rs] | ZeroExtendImm | OR |
| nor | R | R[rd] = ~(R[rs] | R[rt]) | NOR |
| lui | I | R[rt] = {Imm, 16’b0} (concatenated) | LUI |
| lw | I | R[rt] = M[R[rs] + s\_imm] | ADD |
| sw | I | M[R[rs] + s\_imm] = R[rt] | ADD |
| slt | R | R[rd] = (R[rs] < R[rt]) ? 1 : 0 | Set Value |
| slti | I | R[rt] = (R[rs] < s\_imm) ? 1 : 0 | Set Value |
| sltiu | I | R[rt] = (R[rs] < s\_imm) ? 1 : 0 | Set Value |
| sll | R | R[rd] = R[rt] << shamt | SLL |
| srl | R | R[rd] = R[rt] >> shamt | SRL |
| j | J | PC = {(PC+4)[31-28],(imm << 2)} | nop |
| jal | J | PC = {(PC+4)[31-28],(imm << 2)}  R[31] = PC + 8 | nop |
| jr | R | PC = R[rs] | nop |
| jalr | R | PC = R[rs], R[31] = PC + 8 | nop |
| beq | I | if(R[rs] == R[rt]) PC = PC + 4 + BranchAddr | SUB |
| bne | I | if(R[rs] != R[rt]) PC = PC + 4 + BranchAddr | SUB |

**3. Implementation**

프로그램을 구현하기 위해서 다음과 같은 요소들로 구성하였다.

모듈 구현과 구현된 함수

* control\_unit
  + set\_control\_signal
* alu
  + do\_arithmetic\_operation
  + get\_ALU\_operation
* memory
  + get\_instruction\_from\_address
  + set\_input\_memory\_and\_return\_data
* register
  + set\_register\_from\_input
  + get\_value\_from\_decoded\_values

이외로, mux와 sign\_extender와 같은 경우 단순한 연산자를 통해서 표현하였다.

각 컴포넌트들이 정상적으로 동작하기 위해서, control signal이 있어야 한다. 그에 따라서 각 명령어 별로 활성화되는 control signal는 다음 표와 같다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Control Singal** | **R** | **LW** | **SW** | **J** | **JR** | **JAL** | **JALR** | **BEQ** | **BNE** | **I** | **ANDI/ORI** | **LUI** |
| reg\_dst | **0** | **1** | **\*** | **\*** | **\*** | **\*** | **\*** | **0** | **0** | **1** | **1** | **1** |
| mem\_read | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| mem\_write | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| mem\_to\_reg | **0** | **1** | **0** | **\*** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| jump | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| alu\_src | **0** | **1** | **1** | **\*** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **1** |
| reg\_write | **1** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **1** | **1** | **1** |
| isBEQ | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** |
| isBNE | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** |
| isJR | **0** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **0** | **0** |
| isSlt | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| isSltu | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| set\_ra | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **0** | **0** | **0** |
| isUpperAccess | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** |
| sign\_extend | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

**4. Evaluation**

(테스트 프로그램 준비 과정) 테스트 프로그램은 E-Learning으로 배부된 camp\_test\_prog.tar을 사용하였다. 제공된 test case 들은 다음과 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| 이름 | 간단한 동작 설명 | 명령어 수 | 사용하는 명령어(위에서부터 누적) |
| simple | 메모리 I/O 및 덧셈 연산 | 9 | addiu, sw, move, lw, sw, jr, sll |
| simple2 | 메모리 I/O 및 결과 값 반환 | 12 | li |
| simple3 | 1 ~ 100까지 덧셈 연산 | 28 | addu, slti, bne(bnez) |
| simple4 | 재귀 함수 처리 | 43 | jal |
| gcd | 최대 공약수 연산 | 63 | subu |
| fib | 피보나치 10항까지 연산 | 52 | beq(beqz) |
| fib\_edited | 피보나치 10항까지 연산 | 55 | jalr |
| input4 | 배열에서 101번째로 작은 값 | 25533 | lui |
| add\_function | 덧셈 함수 |  |  |

(테스트 프로그램 소개)Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed.

(각 테스트 프로그램 별, 결과 Formalizing)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Success | v[0] | # of executed Inst | # of  R type | # of  I type | # of  J type | # of  mem access | # of  branch |
| simple | O | 0 | 8 | 4 | 4 | 0 | 2 | 0 |
| simple2 | O | 100 | 10 | 3 | 7 | 0 | 4 | 0 |
| simple3 | O | 5050 | 1330 | 409 | 920 | 1 | 613 | 102 |
| simple4 | O | 55 | 243 | 79 | 153 | 11 | 100 | 10 |
| fib | O | 55 | 2679 | 818 | 1697 | 164 | 1095 | 109 |
| fib\_edit | O | 55 | 3005 | 1200 | 1750 | 55 | 1095 | 109 |
| gcd | O | 1 | 1061 | 359 | 637 | 65 | 486 | 73 |
| input4 | O | 85 | 23372706 | 101527862 | 13219741 | 103 | 7116606 | 2029699 |
| add\_function | O | 24927 | 24 | 7 | 16 | 1 | 10 | 0 |

Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed.

(추가적으로 준비한 테스트 케이스) MIPS 컴파일러를 이용하여, 추가적으로 실행 가능한 바이너리를 준비하였다. Ubuntu linux 22.04 컨테이너 환경에서 mips-linux-gnu-gcc를 이용하여 프로그램을 컴파일 하였다. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed.

**5. Conclusion**

**(여태까지 언급한 부분을 요약 및 작성, 여기에서 개선의 여지?)** Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed. Computer Architecture is the fundamental theory of how the system constructed.

**Appendix. Personal Feeling**

Input4.bin 테스트를 진행할 때, 계속해서 왜 의도한 값이 나오지 않는지 궁금해서, 어떤 instruction을 실행할 때 문제가 발생하는지 Instruction asm 파일을 하나하나 읽어가고, log를 통해서 어떤 부분에서 레지스터 값이 어떻게 변경되었는지 추적했습니다. 그 결과로, 오히려 instruction이 실행되면서 어디에서 문제가 발생했는지 도저히 찾기가 어려웠습니다.

교수님이 언급했던, test file을 바꿔가면서 어떠한 instruction이 추가되었는지, 확인하고 어떤 명령어가 정상적으로 동작하는지 체크하라는 말씀에 기존 명령어가 정상적으로 동작하는지 다시 재고할 수 있었습니다. 제가 MIPS Greet Sheet을 무시하고, 그저 제가 생각했던 Sementic을 기준으로 동작시킴에 따라서 의도하지 않은 결과가 발생하였습니다.

또한, 여기에서 얼마나 우매하게 디버깅을 하고 있었구나를 깨닳았습니다. 문제가 발생한 instruction을 찾기 위해서, test case의 프로시저를 일일이 봐가면서 SW의 변경점을 추적하고 하였습니다. 이는 HW가 ISA를 통해서 얻을 수 있는 이점인 SW를 고려하지 않고 ISA가 정의하는 마이크로아키텍처만 충실하게 구현하면 되는 것을 무시하였습니다.

따라서, 이 과제에서 제가 배운 것은 다음과 같습니다.

첫째, 내가 구현해야 할 사항을 가시화하여 Step by Step으로 구현 목표를 구체적으로 확인할 수 있도록 할 것.

둘째, 각 Test case가 있다면, Test case 간의 차이점을 분명하게 파악할 것.

셋째, 과제 및 개발의 요구 사항을 충분히 숙지하고 요구 사항이 달성되었는지 반복적으로 확인할 것

1. https://www.cs.uregina.ca/Links/class-info/201/SPIM-AddressingMode/lecture.html [↑](#footnote-ref-1)