Mobile processor Programming

assignment #2: Simple Calculator with your own ISA

모바일시스템공학과 김태경 (32211203)

목차

Freeday used 3
Freeday left 2

2024.03.31.

1. 프로젝트 목표

MIPS 란?

MIPS는 1981년 개발 이후로 수 십년에 걸쳐 사용된 마이크로프로세서로, 32개의 범용 레지스터를 가지며, ISA(Instruction Set Architecture)을 구현하는 모든 산술 및 논리 연산이 레지스터 내에서 수행됩니다.

MIPS 명령어에는 레지스터 간 연산을 다루는 R-type, 즉시값을 다루어 연산하는 I-type, 명령어의 순서를 조작하는 J-type이 크게 3가지로 구분합니다.
사용되는 31개의 ISA 중에는 데이터를 로드-저장하는 LW(Load Word)와 SW(Store Word)가 있는데 이는 레지스터의 연산명령어를 분리하여 명령어 실행하는 파이프라인을 구축하는데 효율적이다.

ISA 란?

ISA는 “Instruction Set Architecture”의 약어로, 하드웨어와 소프트웨어 간의 표준화된 인터페이스를 정의하여 제조사에 구애받지 않고 호환되도록 보장합니다.
ISA의 주요 구성요소에는 명령어 세트, 명령어 형식, 레지스터 및 메모리 구조, 명령어 실행 동작이 있는데 각각 다음의 역할을 한다.

1) **명령어 세트**: ISA는 컴퓨터가 인식하고 실행할 수 있는 명령어의 집합을 정의하며, 산술-논리 연산, 메모리 액세스, 분기 등의 명령어가 포함될 수 있다.
2) **명령어 형식**: ISA는 명령어가 어떻게 인코딩되고 해석해야 하는지를 정의하며, 명령어의 operand(피연산자)와 필요한 다른 정보를 결정한다.
3) **레지스터 및 메모리 구조**: ISAsms 프로세서가 사용할 수 있는 레지스터와 메모리 구조를 정의하며, 레지스터의 개수, 크기, 메모리 주소 공간 등을 포함한다.
3) **명령어 실행동작**: ISA는 각 명령어에 대한 동작과 그 결과를 명시하며, 명령어의 수행 중 레지스터와 메모리의 상태가 어떻게 변경되는지 설명합니다.

이 프로젝트에서는 ISA가 적혀있는 MIPS Green sheet를 기반하여 Datapath의 구조를 이해하고, Binary 파일을 입력파일로 받는 MIPS single cycle simulator 프로그램을 다양한 기능의 ISA를 구축함으로써 MIPS의 동작원리를 깊이있게 이해하는 것이 목표이다,

2. 프로젝트 조건

기본요구사항

1. Binary 파일을 입력파일로 받아 프로그램을 실행할 수 있어야 한다.
    I.실행 전 바이너리 파일의 모든 내용을 입력받아 데이터로 사용한다.
    I. 메모리는 큰 배열로 정의된 구조입니다.

2. 레지스터는 각 32bit의 32줄의 배열로 이루어져 있다.
    I. 초기 LR(r31)과 SP(r29)의 값은 0xffffffff, 0x10000000이다.

3. 명령어의 단계는 5단계로 나뉜다.
    I. 명령어를 메모리에서 CPU로 이동하는 단계인 **Fetch**
    II. 명령어를 해독하여 동작을 식별하는 **Decode**
    III. ALU연산을 수행하여 계산결과를 출력하는 **Execution**
    IV. stage 메모리에 접근하는 **Memory Access**
    V. 계산결과를 레지스터에 update하는 **Write Back**

4. 각 cycle이 끝나면 simulator는 계산된 결과값을 출력한다.
    I. 각 명령어의 실행 상태와 변경 후의 아키텍처 상태
    II. 프로그램 종료후 실행된 명령어(I, R, J)의 수
    III. 메모리에 접근한 명령어 및 Branch taken이 이뤄진 명령어 수

추가요구사항 (목차 6)

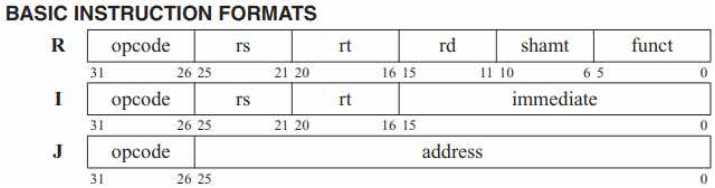
1. 제어신호와 데이터의 이동경로를 분리한다.
2. JALR 명령어를 구현한다.

제외된 요구사항

1. MIPS ISA의 명령어 중 구현하지 않아도 되는 명령어 6가지
    I. 0x24 Load Byte unsigned
    II. 0x25 Load Halfword unsigned
    III. 0x30 Load Linked
    IV. 0x28 Store byte
    V. 0x38 Store Halfword
    VI. 0x29 Store Conditional

3. 프로젝트에 사용된 개념

MIPS Green Sheet 이해



Binary 파일에서 입력되는 명령어는 16진수의 8자리로 이루어지며, 이를 2진수를 변환하여 32bit의 instruction를 얻을 수 있다. 이 32bit의 명령어를 위 그림 Basic Instruction Formats을 참고하여 쪼개면 각각 다음을 얻을 수 있다.

기본 요소

- opcode[6bit] : 명령어의 종류를 구분하는 기준이 된다. (R-type에서는 00)
- rs, rt, rd[5bit] : 레지스터의 값을 불러오거나 값을 입력하는데 사용된다.
- shamt[5bit] : shift연산에서 shift할 양을 저장한다.
- funct[6bit] : R-type에서는 opcode가 00이기에 funct으로 명령어를 구분한다.
- immdiate[16bit] : 상수 필드값으로 ALU에 사용되는 즉시값을 저장한다.
- address[26bit] : J-type에서의 jump하여 이동할 메모리주소를 저장한다.

요소 활용

```
SignExtImm = { 16{immediate[15]}, immediate }
ZeroExtImm = { 16{1b'0}, immediate }
BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
JumpAddr =   { PC+4[31:28], address, 2'b0 }
```

- ZeroExtImm[32bit] : 논리연산의 ALU가 실행될 때. 상위비트를 0으로 채운다.
- SignExtImm[32bit] : 산술연산의 ALU가 실행시, 상위비트를 imm[15]로 채운다.
- BranchAddr[32bit] : 상위 14bit를 imm[15]로 채우고, 나머지 자리에 imm값을 왼쪽으로 2번 shift한 18bit로 채워 32bit를 만든다.
- JumpAddr[32bit] : 상위 4bit를 pc+4의 값으로 채우고, address값을 왼쪽으로 2번 shift한 28bit로 채워 32bit를 만든다.

MIPS ISA를 구현하기 위해 코딩에서 위의 요소들을 만들기 위해서는 받아온 32bit에 bit encoding을 활용하여 각각의 요소를 구현할 수 있다.

4. 프로그램 빌드 환경

컴파일 환경: Linux Assam, with GCC
컴파일 방법: gcc 파일명.c 으로 실행할 파일을 컴파일
실행방법: 컴파일 후 ./a.out 으로 실행결과 확인
파일위치: Assam/test_prog - simple, simple2, simple3, simple4, gcd, fib.c

5. 프로그램 구성

1). 메모리, pc, 레지스터 구조 정의 및 초기화

- 통용되는 크기인 0x400000으로 메모리를 설정한다.
- 다음 명령어를 가르키는 pc는 변수 선언 및 0으로 초기화 해준다.
- 32개의 레지스터는 배열로 지정하며 29, 31번 레지스터를 제외하면 0으로 초기화 하며, 29번은 0x1000000, 31번은 0xffffffff으로 초기화 한다.

```
int memory[0x400000];
unsigned int pc = 0;
int Regs[32] = { 0, };
```

```
// Initialization: Registers
for (int k = 0; k < 32; k++)
    Regs[k] = 0;
Regs[29] = 0x1000000;
Regs[31] = 0xffffffff;
```

2). Binary 파일 호출 및 오류확인 & 메모리에 명령어 입력

```
// test for checking binary input/output
FILE* fp = NULL;
fp = fopen("fileName", "rb");
if (fp == NULL)
{
    perror("file open error");
}
```

- Binary 파일을 기반하여 메모리의 배열에 각각의 명령어를 입력한다.

```
// Initialization: Loading program
while (1)
{
    int ret = fread(&var, sizeof(var), 1, fp);
    if (ret == 0) break;

    unsigned int b1, b2, b3, b4;
    b1 = var & 0x000000ff;
    b2 = var & 0xff00;
    b3 = var & 0x00ff0000;
    b4 = (var >> 24) & 0xff;
    int res = (b1 << 24) | (b2 << 8) | (b3 >> 8) | b4;

    int instruction = res;

    memory[i] = instruction;
    i++;
}
fclose(fp);
```

3). while 반복문을 설정하여 4, 5번의 과정을 반복한다.

- pc로 while 반복문의 중단점을 지정한다.
  (pc값이 메모리의 마지막을 가르킬 경우 종료)

```
// terminal condition check
if (pc == 0xffffffff) break;
```

4). 명령어 가져오기(Fetch)

- 메모리 배열에 저장된 명령어를 불러온다

```
int inst = memory[pc / 4];
```

5). 명령어 해독하기(Decode)

- 가져온 32자리의 2진수의 명령어를 shift, &, | 등의 연산자를 활용하여 각각의 변수에 값을 초기화 한다.

```
// Instruction Decode
int opcode = (inst >> 26) & 0x3f;
int rs = (inst >> 21) & 0x1f;
int rt = (inst >> 16) & 0x1f;
int rd = (inst >> 11) & 0x1f;
int shamt = (inst >> 6) & 0x1f;
int func = inst & 0x0000003f;
int imm_ = inst & 0x0000ffff;
int addr = inst & 0x03ffffff;
int s_imm = (imm_ & 0x8000) ? (imm_ | 0xffff0000) : (imm_);
int z_imm = imm_ & 0x0000ffff;
int b_addr = s_imm * 4;
int j_addr = ((pc + 4) & 0xf0000000) | (addr << 2);
```

6). 실행 및 메모리 접근, 레지스터 업데이트 (**Execution**, **Memory**, **Write Back**)

- 다음과 같이 switch를 이중으로 사용함으로서 opcode를 기준으로 명령어의 작업을 수행하거나 opcode가 0일때는 funct를 기준으로 명령어의 작업을 수행한다.
- 각각의 작업에서는 ALU의 연산이 이루어지며, 과정이 보일 수 있도록 출력한다.
- 명령어에 따라 일부는 메모리에 접근하여 값을 불러오기도, 저장하기도 한다.
- 연산의 결과값은 명령어의 작업에 따라 지정된 레지스터에 update된다.
- 작업이 완료된 후 다음 명령어를 불러오기 위해 pc값에 +4를 한다.
- 실행된 명령어를 종류별로 카운트 하기 위하여 I, R, J_inst값을 높여준다.

ex)

```
switch (opcode) {  // Separated by opcode
    case 0x23: // Load Word
        AluResult = Regs[rs] + s_imm;                 // ALU연산
        printf("@ 0x%x LW:  M[ R[%d]:0x%x + 0x%x] -> R[%d]: 0x%x\n",
            pc, rs, Regs[rs], s_imm, rt, AluResult); // ALU연산과정 및 결과 출력
        Regs[rt] = memory[AluResult / 4];           // 연산결과 Write Back
        pc = pc + 4;                                // pc 다음 명령어로 이동
        I_inst = I_inst + 1;                        // 사용된 명령어 count
        break;
        ...//생략

    case 0x00: // Separated by funct
        switch (funct) {
            case n:       //생략
                break;
        default:
            pc = pc + 4;
        break; }
    default:
        break; }
```

6). 최종 결과 출력

- Register[2]의 최종 반환값을 출력
- 실행된 R, I, J-type명령어와 Nop의 수 출력
- Memory Access, Branch실행과 Branch taken이 일어난 횟수 출력

```
// Output result & stats
printf("₩n₩nFinal return value Regs[2]: 0x%x₩n", Regs[2]);
printf("Num of Executed Inst: %d₩nR_inst: %d, I_inst: %d, J_inst:%d, Nop: %d₩n",
    R_inst+I_inst+J_inst+Nop, R_inst, I_inst, J_inst, Nop);
printf("Memory Access inst: %d₩n", M_count);
printf("Num of Branch: %d₩nNum of Branch taken: %d₩n", B_count, B_count_S);
```

6. Optionally 구현

Optionally 1 : JALR 구현 (Jump And Link Addresss)

JALR은 R-type의 명령어이며, 실행되는 pc 값은 0x20이고, 바이너리 파일에서 명령어는 0x0380f809이고 000000/11100/00000/11111/00000/001001(32bit)으로 opcode=0, rs=28, rt=0, rd=31, shamt=0, funct=9를 가진다.

bit-encoding에서 rs 레지스터는 jump target address값을 나타내며, rd 레지스터는 반환되는 address값을 저장하는 역할을 한다.

구상한 방법 : 우선 JALR 명령어가 불러진 상황에서 rs의 값을 rd[31]에 옮겨넣고, memory[pc + 4]에 JALR의 rs와 funct 0x8으로 이루어진 8자리 명령어를 입력하여 JALR의 다음 명령어에서 rs의 주소로 JR명령어가 실행되도록 한다.

2진수 -> 16진수의 변환에서 막혔습니다.

```
case 0x9:  // Jump And Link Register
    AluResult = Regs[rs];
    printf("@ 0x%x JALR: Regs[%d]: 0x%x -> Regs[31], jump_target_address: 0x%x -> pc: 0x%x\n",
        pc, rs, Regs[rs], Regs[rs], AluResult);
    Regs[31] = AluResult;
    int JALR = 0x00;    // 이전의 rs값과 funct에 0x8(JR)을 넣어 8자리 명령어를 만든다.
    memory[(pc + 4) / 4] = JALR;
    pc = pc + 4;
    J_inst = J_inst + 1;
    break;
```

Optionally 2 : 바이너리 명령어 input 프로그램 만들기

X

7. 문제점 및 해결방법

simple.bin 실행중 오류

```
@ 0x0 AddIU :  R[29]: 0x1000000 + 0xfffffff8 -> R[31]: 0xffffff8
@ 0x4 SW:  R[30]: 0x0 -> M[ R[29]:0x1000000 + S_imm 0x4
@ 0x8 AddU:  R[29]: 0x1000000 + R[0]: 0x0 -> R[0]: 0x1000000
@ 0xc Nop@ 0x10 AddU:  R[30]: 0x1000000 + R[0]: 0x0 -> R[0]: 0x1000000
@ 0x14 Nop@ 0x18 AddIU :  R[29]: 0x1000000 + 0x8 -> R[0]: 0x1000008
@ 0x1c JR:  R[31]: 0xffffff8 -> PC: 0x3ffffe0]
@ 0x3ffffe0 Nop
C:\Users\USER\source\repos\0.1.testing\x64\Debug\0.1.testing.exe(프로세
```

* 종료 후 실행내역을 알려주는 값 출력 X

-> log 확인결과 0x0 명령어에서 rt 레지스터가 아닌 rd 레지스터를 호출하여 0x1c 명령어에 0xffffffff(while의 중단점)이 아닌 0xfffffff8으로 jump하여 결과값의 출력이 안 되었습니다.

=> 수정: addiu 명령어 수정 후 원활히 작동하였습니다.

simple2.bin 실행중 오류

* 실행 중 SW에서 중단되는 오류가 발생했습니다.

-> 메모리에서 불러오는 pc값을 4로 나눠어 입력되는 것을 고려하지 않아 Load Word와 Store Word이 작동하지 않았습니다.

=> 수정: SW: memory[AluResult/4] = Regs[rt]; / LW: Regs[rt] = memory[AluResult/4];

simple3.bin 실행중 오류

```
@ 0x0 AddIU :  R[29]: 0x1000000 + 0xfffffffe8 -> R[31]: 0xffffe8
@ 0x4 SW:  R[30]: 0x0 -> M[ R[29]:0x1000000 + S_imm 0x14 ]
@ 0x8 AddU:  R[29]: 0x1000000 + R[0]: 0x0 -> R[0]: 0x1000000
@ 0xc SW:  R[0]: 0x0 -> M[ R[30]:0x1000000 + S_imm 0x8 ]
@ 0x10 SW:  R[0]: 0x1000008 -> M[ R[30]:0x1000000 + S_imm 0xc ]
@ 0x14 SW:  R[0]: 0x100000c -> M[ R[30]:0x1000000 + S_imm 0x8 ]
@ 0x18 Jump:  pc = j_addr: 0x20000044


Final return value Regs[2]: 0x0
Num of Executed Inst : R_inst: 1, I_inst: 5, J_inst:1,
Total_inst: 7
Num of taken Branch: 0
C:\Users\USER\source\repos\0.1.testing\x64\Debug\0.1.testing.exe
```

* Jump 실행중 오류 발생

-> Jumpaddress의 선언 중 비트연산의 구현이 제대로 되지 않았습니다

=> 수정: int j_addr = ((pc + 4) & 0xf0000000) | (addr << 2);

+) ZeroExtImm, BranchAddress의 수정

```
int z_imm = imm_ & 0x0000ffff;
int b_addr = s_imm * 4;
```

이외에 simple4.bin, gcd.bin, fib.bin까지의 구현에는 문제가 없었습니다.

8. 실행결과

```
Final return value Regs[2]: 0x0
Num of Executed Inst: 8
(R_inst: 3, I_inst: 4, J_inst:0, Nop: 1)
Memory Access inst: 2
Num of Branch: 0
Num of Branch taken: 0
```

1) simple.bin의 실행결과
Regs[2] = 0

```
Final return value Regs[2]: 0x64
Num of Executed Inst: 10
(R_inst: 3, I_inst: 7, J_inst:0, Nop: 0)
Memory Access inst: 4
Num of Branch: 0
Num of Branch taken: 0
```

2) simple2.bin의 실행결과
Regs[2] = 100

```
Final return value Regs[2]: 0x13ba
Num of Executed Inst: 1330
(R_inst: 104, I_inst: 920, J_inst:1, Nop: 305)
Memory Access inst: 613
Num of Branch: 102
Num of Branch taken: 101
```

3) simple3.bin의 실행결과
Regs[2] = 5050

```
Final return value Regs[2]: 0x37
Num of Executed Inst: 243
(R_inst: 60, I_inst: 153, J_inst:11, Nop: 19)
Memory Access inst: 100
Num of Branch: 10
Num of Branch taken: 9
```

4) simple4.bin의 실행결과
Regs[2] = 55

```
Final return value Regs[2]: 0x1
Num of Executed Inst: 1061
(R_inst: 222, I_inst: 637, J_inst:65, Nop: 137)
Memory Access inst: 486
Num of Branch: 73
Num of Branch taken: 45
```

5) gcd.bin의 실행결과
Regs[2] = 1

```
Final return value Regs[2]: 0x37
Num of Executed Inst: 2679
(R_inst: 546, I_inst: 1697, J_inst:164, Nop: 272)
Memory Access inst: 1095
Num of Branch: 109
Num of Branch taken: 54
```

6) fib.bin의 실행결과
Regs[2] = 55

7) input4.bin의 실행결과
pc값 기준 0x18e38에서 BNE가 실행되며 pc값을 0x18dd0으로 옮기는데
[pc : 0x18dd0~0x19e38]의 구간이 계속 반복되며 종료되지 않았습니다.
이전단계들에서 이상이 있는지 확인해 보았지만 찾지 못하였습니다.

8) fib.bin의 JALR 실행결과
구현 실패 pc + 4의 바이너리

9. 느낀점

과제를 막 받았을 때는 MIPS에 대해 부분부분만 이해하고 있을 뿐 전체적인 구상을 할 수가 없는 상태였고, Binary 파일을 읽어들이는 방법만 아니라 기본적인 file을 읽어들이는 것도 익숙하지 않았기에 어디서부터 시작해야할지 막막한 상황이었습니다. 기존의 수업을 들으면서 MIPS에서 명령어별로 어떻게 실행되는지에 대해서만 이해하고 있었다면, 중간고사의 datapath를 그리는 것에 깊게 파고들기 시작하고서 전체적인 MIPS의 구조에 대해서 체계적으로 이해할 수 있었습니다.

다행히도 수업 중 교수님이 예시로 만들어 주신 틀을 기반으로 작성하여 가장 난해했던 Binary 파일을 읽어들이는 문제는 다행히 큰 문제없이 넘어갈 수 있었으며, 파일의 입출력 하는 방법에 대해서 다시 자세히 공부해야겠다는 생각이 들었습니다.

사용해본 적 없는 비트 연산자를 활용하여 명령어들의 요소를 선언하는 것이 가장 어려워했던 부분인데, 잘 작동하는 것 같으면서도 자잘한 오류들과 결과값에 오차가 생기는 원인은 명령어들의 요소를 잘못 선언하여 생기는 문제들이 대다수였기에, 기본기가 중요하다는 점을 다시 한번 깨달았습니다.

현재 만들어둔 코드들은 switch case를 이용하여 Execution, Memory Access, Write Back이 한곳에 묶여있어 스파게티형 코드로 만들어둔 상태인데 각각의 명령어의 실행 5단계를 명확하게 구분하고, ALU의 실행, Memory의 구현 등의 모듈화를 진행하고 싶었지만, 과제 기한으로 인해 모듈화를 진행하지는 못한 것이 아쉬웠습니다.

다만 수업 중 배운 single-cycle보다 cycle의 소모에서 효율적인 pipeline이 다음 과제로 주어진 상황이기에 지금 만들어둔 코드를 전면적으로 수정하면서 모듈화를 진행해볼 예정입니다.