**MiniC to RISC-V**

컴파일러 개론

최종 프로젝트

201402315

곽현준

**1. Java bytecode vs RISC-V**

Java bytecode는 레지스터를 다로 사용하지 않고 런타임 스택을 사용한다. 반면 RISC-V는 Register와 memory의 구분이 반드시 필요하다.

Java bytecode의 경우 store과 load를 통해 손쉽게 저장하고 로드할 수 있었지만, RISC-V는 각각의 경우에 알맞게 레지스터에 저장할 필요가 있다. -1)

또한 상수를 구현할 때에도 명확한 구분을 할 필요가 있다. RISC-V에서 상수를 구현하기 위한 방법은 레지스터에서 “addi x0 imm” 와 같은 방법으로 상수를 레지스터에 저장하고 이를 사용하는데, 이때 addi에서 사용할 수 있는 immediate의 값은 굉장히 한정적이다. 따라서 상수 값의 크기에 따라 다른 방법으로 구현할 수 있어야 한다. – 2)

RISC-V는 배열을 저장할 때에도 분명한 구분이 있다. 만약 배열 int array[]가 존재한다면, 이는 메모리에 저장되고, array의 주소를 받아 위치를 찾게 된다. - 3)

무엇보다 중요한 것은 함수 구현이다.

함수 구현에 있어 call과 return에서 return할 주소와 값, 또 call할 주소와 값을 올바른 레지스터에 저장하고, 불러오는 것은 구현에 있어 가장 중요한 것 중 하나일 것이다. - 4)

이 과정에 있어 스택 메모리는 필수적이다. miniC에서 dynamic data를 사용하지 않기에 heap의 구현은 제한적이나, function을 구현하는 것에 스택 메모리를 통해 값을 저장, 호출해야 하므로 반드시 구현하여야 한다. - 5)

**2. RISC-V를 구현할 때**

RISC-V로 구현할 때에 중요한 것 중 하나는 레지스터를 활용해야 한다는 것이다.

각 값을 레지스터에 알맞게 저장하고, 이를 호출하는 것은 구현에서 가장 중요한 부분 중 하나일 것이다. 메모리 또한 마찬가지일 것이다.

**3. 구현 계획**

**3.1. 레지스터**

RISC-V에서의 레지스터는 아래와 같다.

|  |  |  |
| --- | --- | --- |
| Register | Usage | Remark |
| x0 | The constant value 0 |  |
| x1 (ra) | Return address |  |
| x2 | Stack pointer |  |
| x3 | Global pointer |  |
| x4 | Thread pointer |  |
| x5 ~ x7 | Temporaries | Can be overwritten by calle |
| x8 ~ x9 | Saved(frame pointer) |  |
| x10 ~ x17 | Function arguments / result |  |
| x18 ~ x27 | Saved | Must be saved/restored by callee |
| x28 ~ x31 | Temporaries | Can be overwritten by callee |

위에 맞게 레지스터에 값을 저장해야 함에 주의한다.

**3.2. 메모리**

메모리의 시작주소는 0x00000000부터 시작하는 것으로 가정하고 시작한다. 실제 Architecture에는 Data Memory와 Instruction Memory의 구분이 있고, 각각의 주소가 필요한 것을 알고 있지만, 구현단계에서는 label을 통해 각 instruction으로 이동할 수 있으므로 label을 이용하여 instruction 간의 이동을 구현하고, 각 instruction이 저장되어있는 PC(instruction memory)에 대한 부분은 제외하고 구현한다.

즉, 메모리의 구현은 Data Memory만을 고려하며, 이 주소의 시작은 0x00000000으로 판단하고, instruction간의 이동은 label을 사용한다.

**3.3. 스택 메모리**

스택 메모리의 경우 명확한 주소를 명시하는 것보다는 stack point(sp)를 통해 공간을 명시하므로 명확한 주소가 필요하지 않을 것이다. (실제 RISC-V에서는 당연히 해당하는 주소가 있다.)

이번 구현에서 스택 메모리에 해당하는 주소는 실제적인 주소값을 사용하지 않고 sp의 증감을 통해 공간을 조작하도록 한다.

**3.4. RISC-V에서의 Instruction**

기본적으로 구현하고자 하는 Instruction은 다음과 같다

* add / sub
* addi
* ld / sd
* logical Instruction ( slli, srli, and, or, xor)
* shift instruction (sll, srl, sra)
* Decision-Making Instruction (beq, bne, bgt, bge, blt, ble)
* Procedure Call Instruction (jal – procedure call, jalr – procedure return)
* Wide Immediate Operands ( lui )

**3.5. 각 Instruction의 bit 표현**

이번 구현에서 주된 목적은 MINIC로 작성된 프로그램을 RISC-V로 컴파일하여 해당 코드가 올바른지에 중점을 두고 있으므로 bit표현에 대한 구현은 따로 하지 않는다. - 단, 구현한다면 굉장히 간단하지만 불필요한 많은 작업으로 인해 큰 의미가 없다고 봄

따라서 비트표현은 따로 구현하지 않는다.