**MOV4 명령어 사용 전 후의 성능 차이**

**보고서 간략 설명**

MOV4 명령어의 사용 유무를 비교하는 동시에 JZ 명령어의 사용 유무 또한 함께 비교했습니다.

따라서, MOV4 명령어 사용 유무에 따른 소스코드의 라인 수와 Total Clks. JZ 명령어 사용 유무에 따른 소스 코드의 라인 수와 Total Clks를 비교했으며, 어떤 코드가 효율적인지에 대해 생각해보았습니다.

**보고서 작성 개요**

1. 비교할 Binary Code 파일 이름
2. MOV4 명령어를 사용하기 않고 JZ 명령어를 사용하지 않은 코드와 이에 대한 설명.
3. MOV4 명령어를 사용하지 않고 JZ 명령어를 사용한 코드와 이에 대한 설명
4. MOV4 명령어를 사용하고 JZ명령어를 사용하지 않은 코드와 이에 대한 설명
5. MOV4 명령어를 사용하고 JZ명령어를 사용한 코드와 이에 대한 설명
6. 비교를 통한 성능차이와 이에 대한 고찰

1.**비교할 Binary Code의 파일 이름**

- MOV4 명령어를 사용하지 않고 JZ 명령어를 사용하지 않은

TPU 명령어 파일 = Compare1MOV4.bin 라인 수: 70

- MOV4 명령어를 사용하지 않고 JZ 명령어를 사용한

TPU 명령어 파일 = CompareJZ1MOV4.bin 라인 수: 16

- MOV4 명령어를 사용하고 JZ 명령어를 사용하지 않은

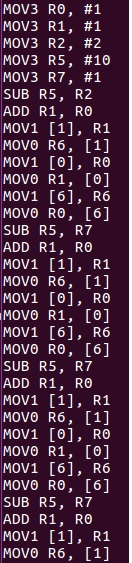
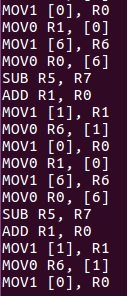
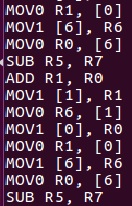
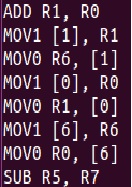
TPU 명령어 파일 = Compare2MOV4.bin 라인 수: 46

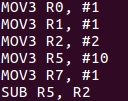
- MOV4 명령어를 사용하고 JZ 명령어를 사용한

TPU 명령어 파일 = CompareJZ2MOV4.bin 라인 수: 13

2**. MOV4 명령어를 사용하지 않고 JZ 명령어를 사용하지 않은 코드와 이에 대한 설명**

<Compare1MOV4.bin의 전체 어셈블리어>



<C코드>

R0 = 1;

R1 = 1;

R2 = 2;

R5 = 10;

R5 = R5 – R2;

여기서 MOV3 R7 #1은 나중에 R--; 을 수행하기 위해

여유 되는 Register R7에 임의로 1을 넣은 것입니다.

<반복되는 부분> 🡪 R5가 8 이므로 While 속 코드가 8번 반복됩니다.

<C코드>

While( R5 ){

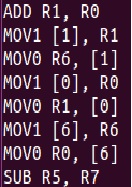
int R6 = R0 + R1;

R1 = R0;

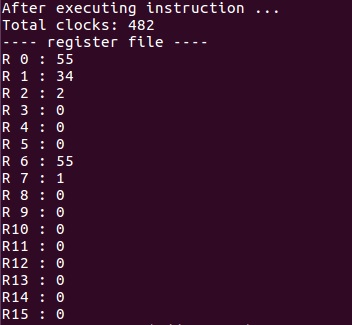
R0 = R6;

R5--;

}

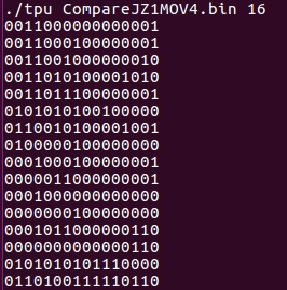


<계산 결과>

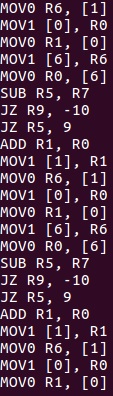
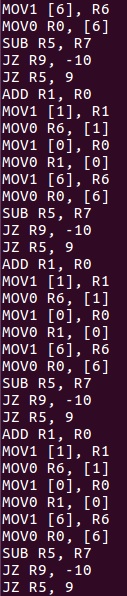
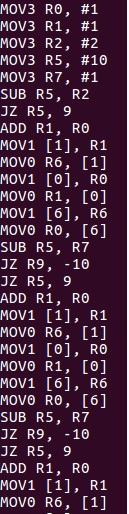


3. **MOV4 명령어를 사용하지 않고 JZ 명령어를 사용한 코드와 이에 대한 설명**

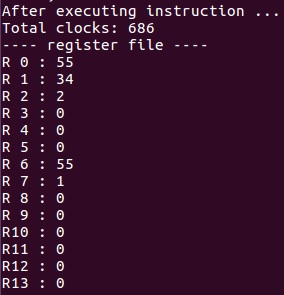
<CompareJZ1MOV4.bin의 전체 Binary Code>



<CompareJZ1MOV4.bin의 전체 어셈블리어>

<계산 결과>

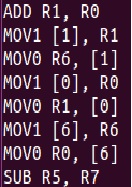
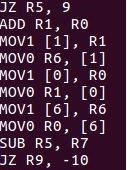


<앞의 Compare1MOV4.bin과의 차이점> : JZ 명령어 사용 유/무의 차이.

JZ명령어를 이용하여 while구문을 표현하여 Binary Code가 굉장히 많이 짧아졌습니다.

이 차이를 어셈블리어로 보여드리겠습니다.

(JZ를 사용 안 했을 때의 while구문) (JZ를 사용했을 때의 while구문)

JZ를 사용하면 반복되는 부분의 Binary Code는 Loop를 한 번에 표현할 수 있습니다.

즉, JZ를 사용 안 하면 왼쪽의 코드가 8번 반복되고 JZ를 사용하면 오른쪽의 코드 한 번으로 while구문을 표현할 수 있습니다.

While( R5 ) 🡪 JZ R5 9 : R5 = 0 이 되면 pc = pc + 1 + 9 가 되어 JZ R9 -10 구문을 넘어가게 되어 while구문을 빠져나오게 됩니다. R5 = 0 이 아니라면 JZ R9 -10 : pc = pc + 1 -10 에 의해 다시 <JZ R5 9> 부터 <JZ R9 -10>까지 실행하게 됩니다..

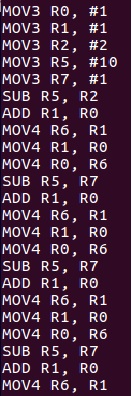
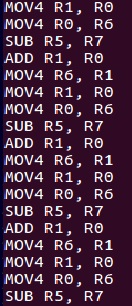
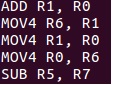
Binary Code로 표현할 때에는 JZ 명령어를 사용하는 게 편하지만 JZ 명령어의 클럭 수가 상당히 크기 때문에 Total clks는 482에서 686으로 크게 증가하게 됩니다.

**MOV4 명령어를 사용하지 않았을 때의 특징**

While문의 코드를 반복할 게 될 때 레지스터에서 다른 레지스터로 데이터를 옮기기 위해서는 메모리를 통해 데이터를 옮겨야 하므로 MOV1, MOV0 명령어가 필수적입니다. 즉, MOV4 명령어의 수행을 MOV1 + MOV0 두 개의 명령어의 조합으로 수행할 수 있습니다. 때문에, 우선 전체적인 코딩의 길이가 길어집니다. MOV1와 MOV0는 각각 8clks입니다. 즉, MOV4의 역할을 16clks로 수행하는 것을 의미합니다. 그래서, 코딩 길이와 함께 Total Clocks = 482로 또한 크게 늘어납니다.

**4.MOV4 명령어를 사용하고 JZ 명령어를 사용하지 않은 어셈블리어와 그에 대한 설명**

<Compare2MOV4.bin의 전체 어셈블리어>

<초기값 입력>

<C코드>

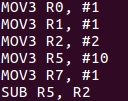
R0 = 1;

R1 = 1;

R2 = 2;

R5 = 10;

R5 = R5 – R2;



<반복되는 부분> 🡪 마찬가지로 8번 반복됩니다.

<C코드>

While( R5 ){

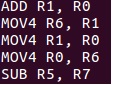
int R6 = R0 + R1;

R1 = R0;

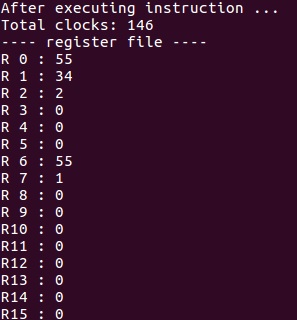
R0 = R6;

R5--;

}

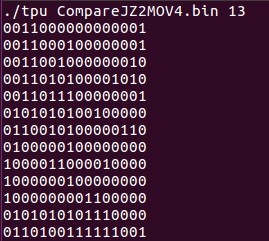


<계산 결과>

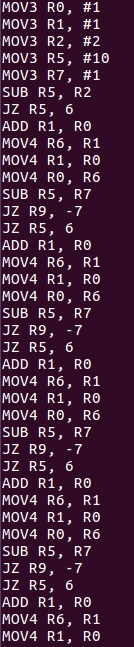
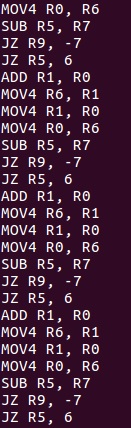


5. **MOV4 명령어를 사용하고 JZ명령어를 사용한 코드와 이에 대한 설명**

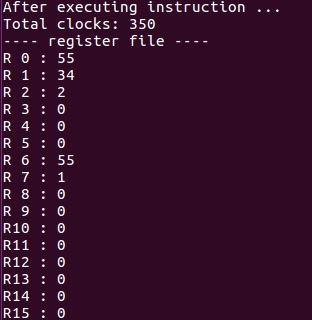
<CompareJZ2MOV4.bin의 Binary Code>



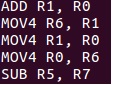
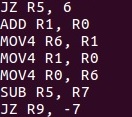
<CompareJZ2MOV4.bin의 전체 어셈블리어>

<계산 결과>



( JZ를 사용 안 했을 때 while구문 ) ( JZ를 사용했을 때 while구문 )

 이 코드를 8번 반복 

위의 비교와 마찬가지로 JZ 명령어를 사용함으로써 while구문이 많이 짧아집니다.

JZ R5 6 🡺 pc = pc + 1 + 6 🡺 R5 = 0 이 되면 JZ R9 -7 뒤로 넘어가 종료되어집니다.

R5 = 0이 아니라면, JZ R9 -7에 의해 pc = pc + 1 -7 🡺 다시 JZ R5 6부터 다시 연산 됩니다.

**MOV4 명령어를 사용했을 때의 특징**

MOV4명령어를 사용하지 않았을 때보다 코딩의 수가 확실히 줄어들었습니다. 그 이유는 메모리를 거치지 않고 바로 레지스터의 데이터를 다른 레지스터로 옮길 수 있기 때문입니다. 즉 MOV1, MOV0 명령어들을 사용하지 않아도 됩니다. 그리고 Total Clocks = 146 또한 줄어들었습니다. 그 이유는 MOV1과 MOV0의 각각의 clks는 8인 반면, MOV4는 2이기 때문입니다. 이로써, MOV4 명령어를 이용함으로써 코딩의 효율도 좋아지고, 실제 수행시간 또한 짧아집니다.

**6.비교를 통한 성능차이와 이에 대한 고찰**

위의 MOV4 명령어를 사용하지 않았을 때와 사용했을 때의 비교를 통해 레지스터의 데이터를 다른 레지스터로 옮길 때에는 MOV4 명령어가 굉장히 효율적인 명령어임을 알 수 있습니다. 그 이유는 메모리를 거치지 않아도 되기 때문인데, 메모리를 거치지 않는다면 수행시간이 줄어들기도 하지만 원래 메모리에 있던 데이터가 손상될 위험도 없어짐을 알 수 있습니다. 이 명령어가 왜 수행시간이 짧은 지를 생각해보자면, MOV4는 메모리에서 데이터를 받아올 필요가 없기 때문에 레지스터 내에서만 수행이 가능합니다. 이 점이 MOV4의 명령어의 수행시간을 크게 줄였을 것이란 생각이 듭니다. 그렇다면, 메모리 측면에서 생각해보자면, 메모리의 데이터를 다른 메모리로 옮기는 명령어가 있다면 이 명령어 또한 큰 효율을 낼 것이라 생각할 수 있습니다. 하지만, 이것은 어느 정도의 한계가 있을 것입니다. 왜냐하면, 메모리의 크기만큼 Operand1의 크기도 늘어나야 되기 때문입니다. Operand1의 크기가 Operand2만큼 늘어나지 않는다면, 결국에는 메모리의 데이터를 다른 메모리로 옮기기 위해서는 레지스터를 거치게 될 것입니다. 더 나아가, 레지스터에서 다른 레지스터로 데이터를 옮길 때, MOV4와 ADD, SUB 명령어를 합한 명령어도 생각해볼 수 있을 것 같습니다. MOV4는 Operand2에서 4bit를 이용하고 ADD, SUB 또한 4bit를 사용합니다.

예를 들어서, MOV4+ADD 명령어를 MOVA라고 한다면,

MOVA R0 R1+R2 🡪 MOVA 0000 00010010으로 레지스터 R1과 레지스터 R2의 데이터를 읽어와 더한 뒤 레지스터 R0에 옮기는 명령어를 생각해 볼 수 있습니다. MOV4+SUB도 이와 마찬가지로 생각해볼 수 있습니다. 저의 예상이지만, 이 경우도 레지스터 내에서만 이루어지기 때문에 수행시간 또한 짧을 것이라 생각이 되며 최고의 효율을 낼 수 있을 것이라 생각이 됩니다.

이처럼 명령어는 만들기만 한다면 무수히 많은 종류의 명령어가 생길 수 있을 것이라 생각합니다. 하지만, 명령어가 많아진다면 그만큼 많은 종류의 명령어를 저장할 수 있어야 하므로 최저의 공간으로 최고의 효율을 낼 수 있는 명령어들의 조합이 중요하며, 편한 명령어가 있더라도 그 명령어의 clks를 고려하여 가장 효율적인 코딩을 하는 것이 중요하다 생각합니다.