마이크로프로세서 HW3

MOV5를 완성하고, JZ를 이용한 While구문으로 5x5 행렬 곱 연산 및 성능 향상을 이루어내는 과제였습니다.

편리하게 while구문을 만들기 위해 JZ의 조건을 변경한 JZ1, JZ2, JZ3을 추가하였습니다.

편리한 MOV2사용을 위해 MOV2의 조건을 변경한 MOV2\_를 추가하였습니다.

MOV2를 참고하여 MOV5를 만들었습니다.

성능 향상 전

우선 A행렬의 첫 행과 B행렬을 곱하여 C행렬의 첫 행을 완성시키고 모두 메모리에 저장

A행렬의 나머지 행을 메모리 저장하면서 B행렬과 곱하여 C행렬의 나머지를 완성 및 연산 결과값을 메모리에 저장.

JZ가 빈번히 사용될수록, 메모리 저장이 잦을수록 또는 중복되면 clock 수가 늘어날 것으로 판단했습니다. 때문에 행렬곱셈 연산 이외에는 JZ를 최소화 시켜 보았습니다. 또한, 메모리와 레지스터를 왕래하며 값을 옮기면 성능이 저하될 것으로 생각하여 MOV5를 굳이 사용하지 않고 MOV2로 A, B행렬에 대한 정보만 메모리에 저장 시킨 후 행렬 곱 연산 시에는 레지스터 내에서 값을 변경시키며 수행했습니다.

성능 향상 후

A, B행렬을 ADD와 MOV2\_를 반복 수행하여 모두 메모리에 저장 시킨 이후 JZ를 통해 A, B행렬의 곱셈 결과값만을 C행렬 위치의 메모리에 저장하였습니다.

아래에 성능 향상 전/후의 코드 및 설명, 실행 사진을 첨부하며 새로 추가한 MOV5와 임의로 추가한MOV2\_, JZ1, JZ2, JZ3에 대한 CExecute.cpp, CDecode.cpp, CDecode.h 사진을 첨부합니다.

아래는 성능 향상 전 코드와 설명입니다.

0011000000000001

0011000100000101

1000001000000000

0011001100011010

1000010000110000

0011010100110011

1000011000100000

0111011001000000

1111001000000010

1111010000000100

0100001000000000

0100010000010000

0100011101100000

1000011000100000

1010001011111000

1111010100000111

1000001000000000

1000011000000000

1000011110000000

0100001100000000

0100010100000000

1000010000110000

1011001111110000

0011001100011010

0100001000010000

1000010000110000

1000011000100000

1000100100000000

1000101000100000

1111001000000010

0111011001000000

0100011101100000

0100010000010000

0100001000000000

1000011000100000

0100100100000000

1010100111111000

1111010100000111

1000001010100000

1000011000100000

1000011110000000

1000100100000000

0100001100000000

0100010100000000

1000010000110000

1011001111110000

1100010111101000

MOV3 R0 1 //

MOV3 R1 5 //고정 (덧셈에 사용)

MOV4 R2 R0 A행렬 (1,1)

MOV3 R3 26 B행렬 (1,1)

MOV4 R4 R3 B행렬 곱셈 연산용

MOV3 R5 51 C행렬 (1,1)에 MOV2\_ 할 레지스터

MOV4 R6 R2 A행렬 곱셈 연산용

{ JZ2

{ JZ1

MUL R6 R4 ((1x26))

MOV2\_ [R2] R2 [0] 1

MOV2\_ [R4] R4 [25] 26

ADD R2 R0 ((2))

ADD R4 R1 ((31))

ADD R7 R6 // R7 은 곱의 합

MOV4 R6 R2

} ->JZ1 R2<6 (-8)

MOV2\_ [R5] R7 //[50] 곱의합

MOV4 R2 R0 R2=1 리셋

MOV4 R6 R0 R6=1 리셋

MOV4 R7 R8 R7=0 리셋

ADD R3 R0 ((27))

ADD R5 R0 ((51))

MOV4 R4 R3 R4 27으로시작

} -> JZ2 R3<31 (-16)

{ JZ3

MOV3 R3 26

ADD R2 R1 ((6))

MOV4 R4 R3 ((26))

MOV4 R6 R2 ((6)) //곱셈용

MOV4 R9 R0 CNT

MOV4 R10 R2 ((6))

{ JZ1

MOV2\_ [R2] R2

{ JZ2

MUL R6 R4

ADD R7 R6 //곱셈 결과값 더함

ADD R4 R1

ADD R2 R0 ((6)) 7 8 9 10

MOV4 R6 R2

ADD R9 R0 ((2)) 3 4 5 6

} -> JZ1 R9<6 (-8)

MOV2\_ [R5] R7 //[55]

MOV4 R2 R10 ((6))

MOV4 R6 R2 ((6)) 곱셈용 초기화

MOV4 R7 R8 ((0)) 곱셈 결과값 합 초기화

MOV4 R9 R0 ((1)) CNT 초기화

ADD R3 R0 ((27))

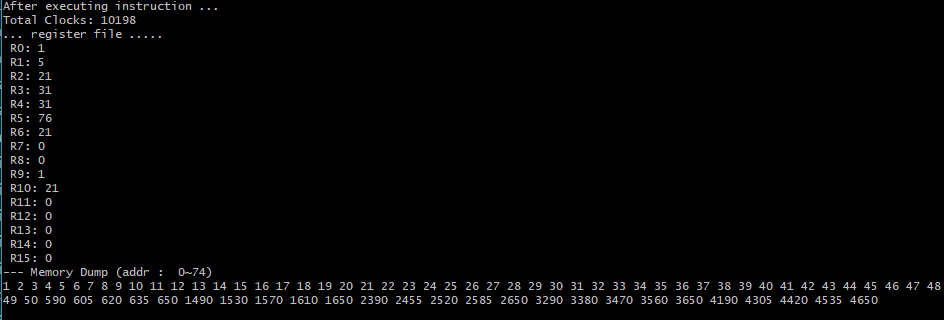
ADD R5 R0 ((56))

MOV4 R4 R3

} -> JZ2 R3<31 (-16)

} -> JZ3 R5 < 76 (-24)

성능 향상 전의 5x5 행렬 곱셈 연산 결과.



다음은 성능 향상 후 코드와 설명입니다.

0011000000000001

0011000100000101

1000001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

0100001000000000

1111001000000010

1000001000000000

1000001100000000

1000110000000000

1000110100000000

0011111100110011

0000010000011001

1000010101000000

0111001101010000

0100011000110000

0100001000000000

0100010100010000

1000001100100000

0100110100000000

1010110111111001

1111111100000110

0100111100000000

0100010000000000

1000001011000000

1000001100100000

1000010101000000

1000011011100000

1000110100000000

1011010011110000

0100110000010000

0000010000011001

1100111111110111

MOV3 R0 1

MOV3 R1 5

MOV4 R2 R0

MOV2\_ [R2] R2

ADD R2 R0

반복해서 M[49]=50이 될때까지

-----------A/B행렬 완성-------------

MOV4 R2 R0(A행렬 1,1)

MOV4 R3 R0(A행렬 곱셈 연산용)

MOV4 R12 R0 (A행렬 행 이동용)

MOV4 R13 R0 (곱셈 연산 루프를 위한 CNT)

MOV3 R15 51 (행렬곱 담을 공간)

MOV0 R4 25 (B행렬 1,1)

MOV4 R5 R4(B행렬 곱셈 연산용)

{ JZ1

{ JZ2

MUL R3 R5

ADD R6 R3 (곱셈 연산 결과값 누적)

ADD R2 R0

ADD R5 R1

MOV4 R3 R2

ADD R13 R0

} JZ1 R13 <6 (-7) (A행렬의 마지막 열을 넘길 시, 즉 C행렬에 들어갈 1개의 결과값 도출해냈을 때 종료)

MOV2\_ [R15] R6

ADD R15 R0

ADD R4 R0 (B행렬 열 변화)

{ JZ3

MOV4 R2 R12 (A행렬 다음 행 이동)

MOV4 R3 R2 (A행렬 곱셈 연산용 레지스터 갱신)

MOV4 R5 R4 (B행렬 다음 열 이동)

MOV4 R6 R14(누적값 0으로 초기화)

MOV4 R13 R0(CNT 초기화)

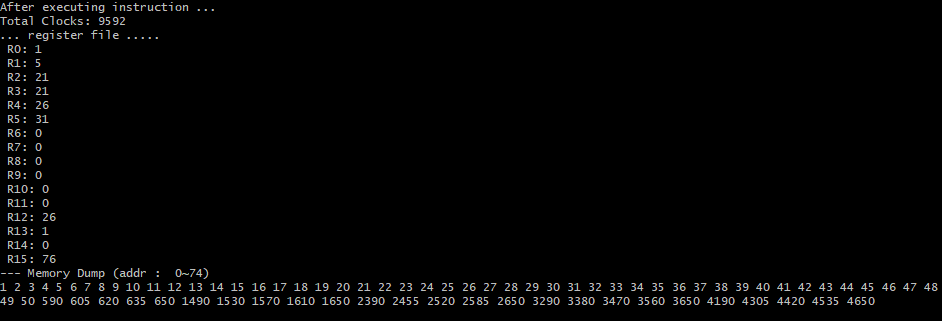
} JZ2 R4 <31 (-16) (B행렬의 마지막 열을 초과할 시, 즉 C행렬의 한 행을 완성하면 종료)

ADD R12 R1 (A행렬 행 이동)

MOV0 R4 25 (B행렬 초기화)

} JZ3 R15 <76 (-19) (C행렬의 마지막 열까지 완성하면 종료)

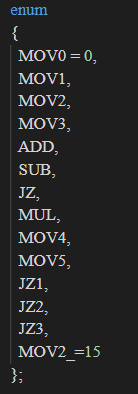
성능 향상 후의 5x5 행렬 곱셈 연산 결과.



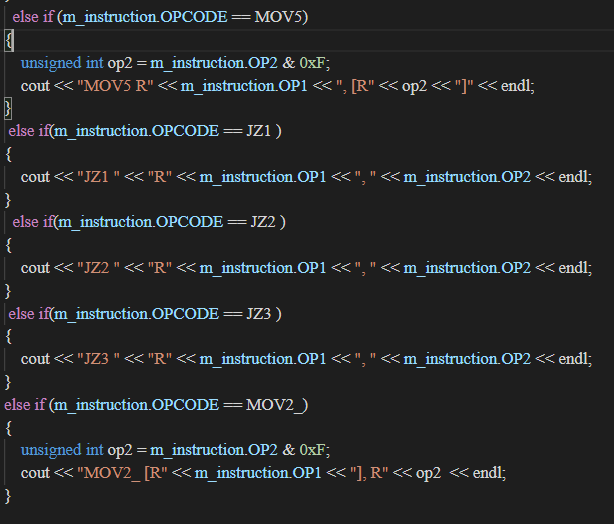
위처럼 A, B행렬의 값에 대한 메모리 저장에 있어서 JZ를 사용하지 않았더니 10198 -> 9592의 성능 향상 결과를 볼 수 있었습니다.

추가한 instruction에 대한 사진들

CDecode.h



CDecode.cpp



CExecute.cpp

