**Digital System Design Lab 06**

**2015-18525 김세훈**

**2. 다음의 C 코드와 같은 behavior behavior behavior를 나타내는 어셈블리 코드를 작성하시오 .**

**조건 : 구성한 RISC 의 명령어 사용**

**단, RISC 명령어 구조는 ‘OP DST SRC’ 형태라고 가정하자.**

int a=0; // at 0x0010

int b=3; // at 0x0020

int c=100; //at 0x0030

for (int i=3 ; i != 0 ; i--){

a = a+b;

c = c-17;

}

b=2012;

**2.1 register r0 , r1 만 사용 가능한 경우**

MOVI r0 0x40

MOVI r1 0x03 // r1 == 3

STOR r1 r0 // initiate i=3 at 0x0040

Jump :

CMPI r1 0x00 // r1 == i

Bcond EQ Out // get out of the loop if i==0

MOVI r0 0x10

LOAD r0 r0 // r0 == a

MOVI r1 0x20

LOAD r1 r1 // r1 == b

ADD r0 r1 // a = a+b

MOVI r1 0x10

STOR r0 r1

MOVI r0 0x30

LOAD r1 r0 // r1 == c

SUBI r1 0x11 // c = c-17

STOR r1 r0

MOVI r0 0x40

LOAD r1 r0 // r1 == i

SUBI r1 0x01 // i = i-1

MOVI r0 JUMP

Jcond UC r0

Out :

MOVI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xE3 // r0=2012

MOVI r1 0x20

STOR r0 r1

//a and b are already stored in memory

**2.2 r0 , r1 , r2 , r3 가 사용 가능한 경우**

MOVI r2 0x30

LOAD r2 r2 // r2 == c

MOVI r3 0x03 // r3 == i ==3

Jump :

CMPI r3 0x00

Bcond EQ Out // get out of the loop if i==0

MOVI r0 0x10

LOAD r0 r0 // r0 == a

MOVI r1 0x20

LOAD r1 r1 // r1 == b

ADD r0 r1 // a = a+b

SUBI r2 0x11 // c = c-17

MOVI r1 0x10

STOR r0 r1 // store a in memory

SUBI r3 0x01 // i = i-1

MOVI r0 JUMP

Jcond UC r0

Out :

MOVI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xFF

ADDI r0 0xE3 // r0=2012

MOVI r1 0x20

STOR r0 r1 // store b in memory

MOVI r1 0x30

STOR r2 r1 //store c in memory

//a is already in memory

**2.3 immediate 명령어를 사용하지 않고 register 4개를 사용한 경우**

MOVI r1 0x20

LOAD r1 r1 // r1 == b

MOVI r2 0x30

LOAD r2 r2 // r2 == c

MOVI r3 0x03

MOVI r0 0x40

STOR r3 r0 // i == 3

Jump :

MOVI r0 0x00

CMP r3 r0

Bcond EQ Out // get out of the loop if i==0

MOVI r0 0x10

LOAD r0 r0 // r0 == a

MOVI r1 0x20

LOAD r1 r1 // r1 == b

ADD r0 r1 // a = a+b

MOVI r1 0x11

SUB r2 r1 // c = c-17

MOVI r1 0x10

STOR r0 r1 // store a in memory

MOVI r0 0x01

MOVI r3 0x40

LOAD r3 r3 //r3 == i

SUB r3 r0 // i = i-1

MOVI r0 0x40

STOR r3 r0

MOVI r0 JUMP

Jcond UC r0

Out :

MOVI r0 0xFF

MOVI r1 0xFF

MOVI r2 0xE3

ADD r0 r1

ADD r0 r1

ADD r0 r1

ADD r0 r1

ADD r0 r1

ADD r0 r1

ADD r0 r2// r0=2012

MOVI r1 0x20

STOR r0 r1 // store b in memory

MOVI r1 0x30

STOR r2 r1 // store c in memory

//a is already in memory

**또한 register 개수 및 imm 명령어가 시스템의 성능에 어떠한 영향을 미치는지 장단점에 대해 서술하시오 .**

우선 register 개수가 부족하면 다음과 같은 2가지 문제점이 발생한다.

1. 명령어의 길이가 길어진다. Register를 다른 변수로 재활용하기 위해 기존의 값을 메모리에 저장한 후 덮어 쓸 값을 메모리에서 불러와야 하기 때문이다. 2.1과 2.2만 비교해 봐도 2.2에서는 ro, r1, r2, r3를 각각 a, b, c, i로 ‘dedicated’하게 사용한 반면, 2.1은 끊임없이 LOAD와 STOR명령어를 통해 r0와 r1을 a, b, c, i에 모두 돌아가며 사용하였다. Iron Law에 의하면 성능(수행 시간, wall clock time)은 f\*CPI\*N(f : clock frequency, CPI : cycle per instruction, N: program 길이) 로 표현된다. 이 때 불필요한 LOAD, STOR 명령어들로 인해 N이 증가하게 되며, 따라서 성능은 저하한다.
2. 또한 LOAD와 STOR는 메모리 오퍼레이션으로 메모리 접근 시간은 일반적인 연산 속도보다 느리다. 따라서 잦은 LOAD와 STOR의 호출은 메모리 접근 속도에 의해 전체적인 평균 CPI가 증가하게 만든다. 따라서 이 또한 전체 수행 시간을 증가시키는 요인으로 작용한다.

또한 immediate value를 사용할 수 없을 경우, 이들을 모두 register로 매개하여서 사용해야 하기 때문에 결론적으로는 가용 register의 개수가 줄어드는 것과 같은 효과가 발생한다. 즉 위에서 제시된 문제점들과 동일하게, LOAD/STORE 명령어의 불필요한 사용으로 인해 프로그램의 길이가 길어지며, 메모리 접근 overhead가 발생하는 문제점이 발생한다.