Code Optimization

Code Optimization

- Code Optimization
  - ✓ 실행 파일이 수행 될 때 속도와 메모리 사용측면에서 효율적으로 동작하는 코드를 생성하는 소프트웨어 기법
  - ✓ 코드 최적화는 3가지 측면을 고려함
    - 속도
    - 메모리 요구량
    - 레지스터 사용량
  - ✓ 중요성
    - 컴파일 시간은 중요하지 않으나 컴파일 된 프로그램의 수행시간은 중요
    - 컴파일 된 프로그램은 수행 횟수가 정해져 있지 않음(무한대)
    - 실제로 GCC의 80~90%가 최적화를 위한 코드임.

- Code Optimization

예제코드 1

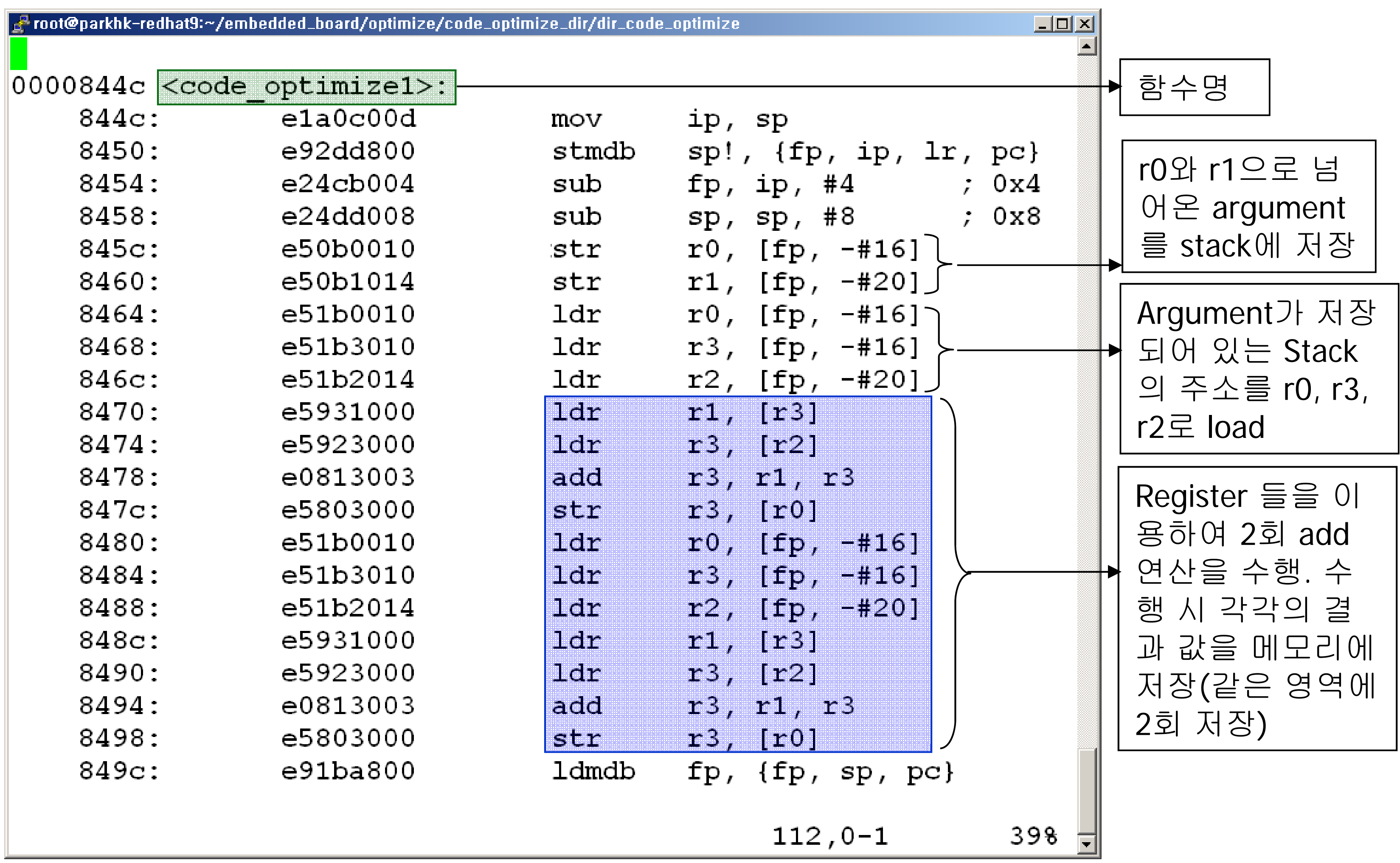
```
/* original source code*/
void code_optimize1(volatile int *v1, volatile int *v2){
        *v1 += *v2;
        *v1 += *v2;
}

/* optimize version */
void code_optimize2(volatile int *v1, volatile int *v2){
        *v1 += 2* *v2;
}
```

Code Optimization

- 예제 코드 비교
  - ✓ V1의 값에 v2의 값을 2회 add연산하는 함수
  - ✓ 메모리 참소 횟수 비교(최소 참조 횟수)
    - code_optimize1()
      - *ip 읽기 횟수 : 2회
      - *jp 읽기 횟수 : 2회
      - *ip 쓰기 횟수 : 2회
    - code_optimize2()
      - *ip 읽기 횟수 : 1회
      - *jp 읽기 횟수 : 1회
      - *ip 쓰기 횟수 : 1회

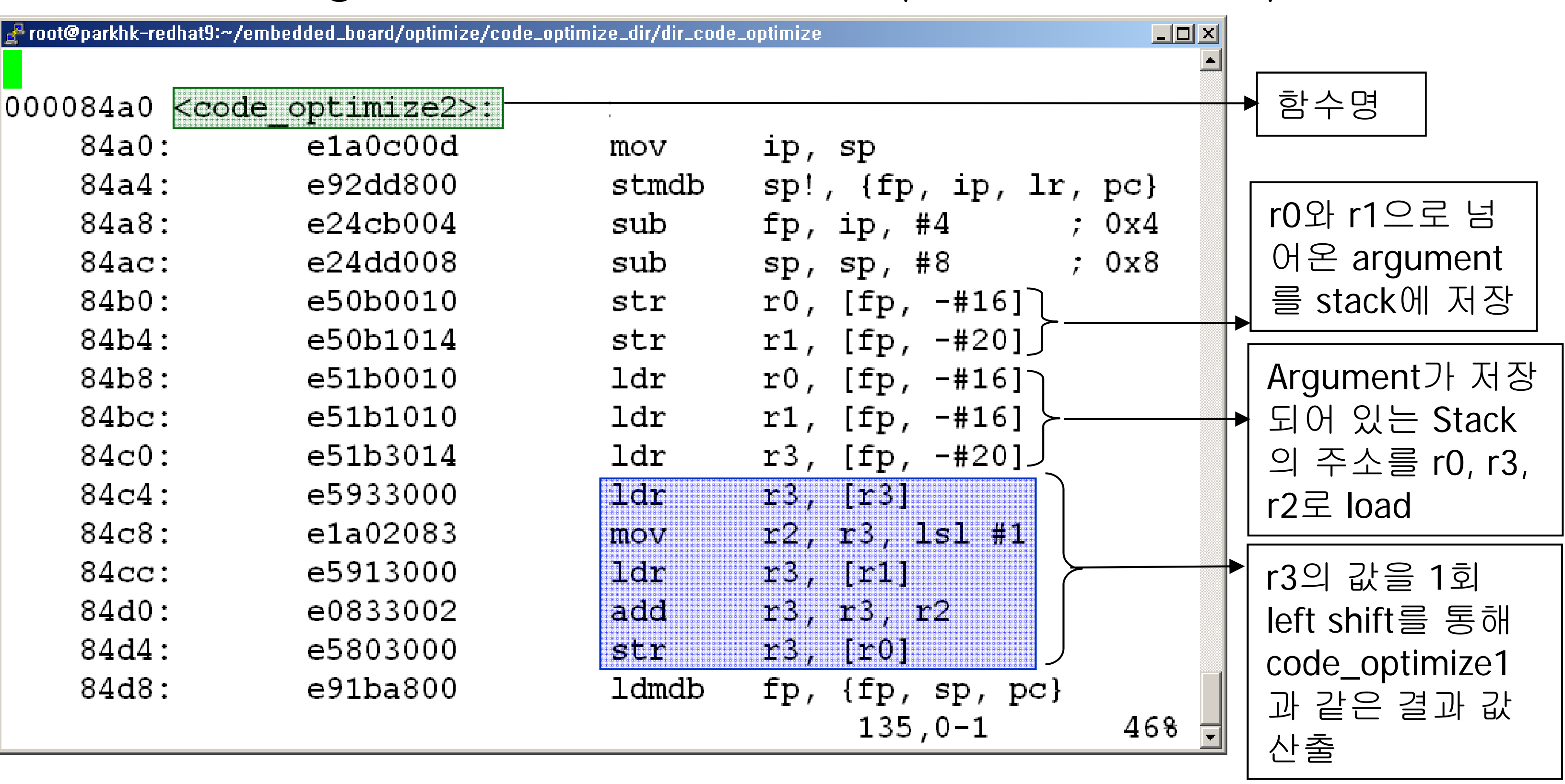
Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o code_optimize code_optimize.c



Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o code_optimize code_optimize.c



Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ objdump결과 부분이 최적화 전후 코드의 변화를 나타내고 있음
  - ✓ code_optimize1()
    - 명령어 수행횟수
      - ldr : 10회
      - add : 2회
      - str : 2회
  - ✓ code_optimize2()
    - 명령어 수행횟수
      - ldr : 2회
      - add : 1회
      - mov : 1회
      - str : 1회

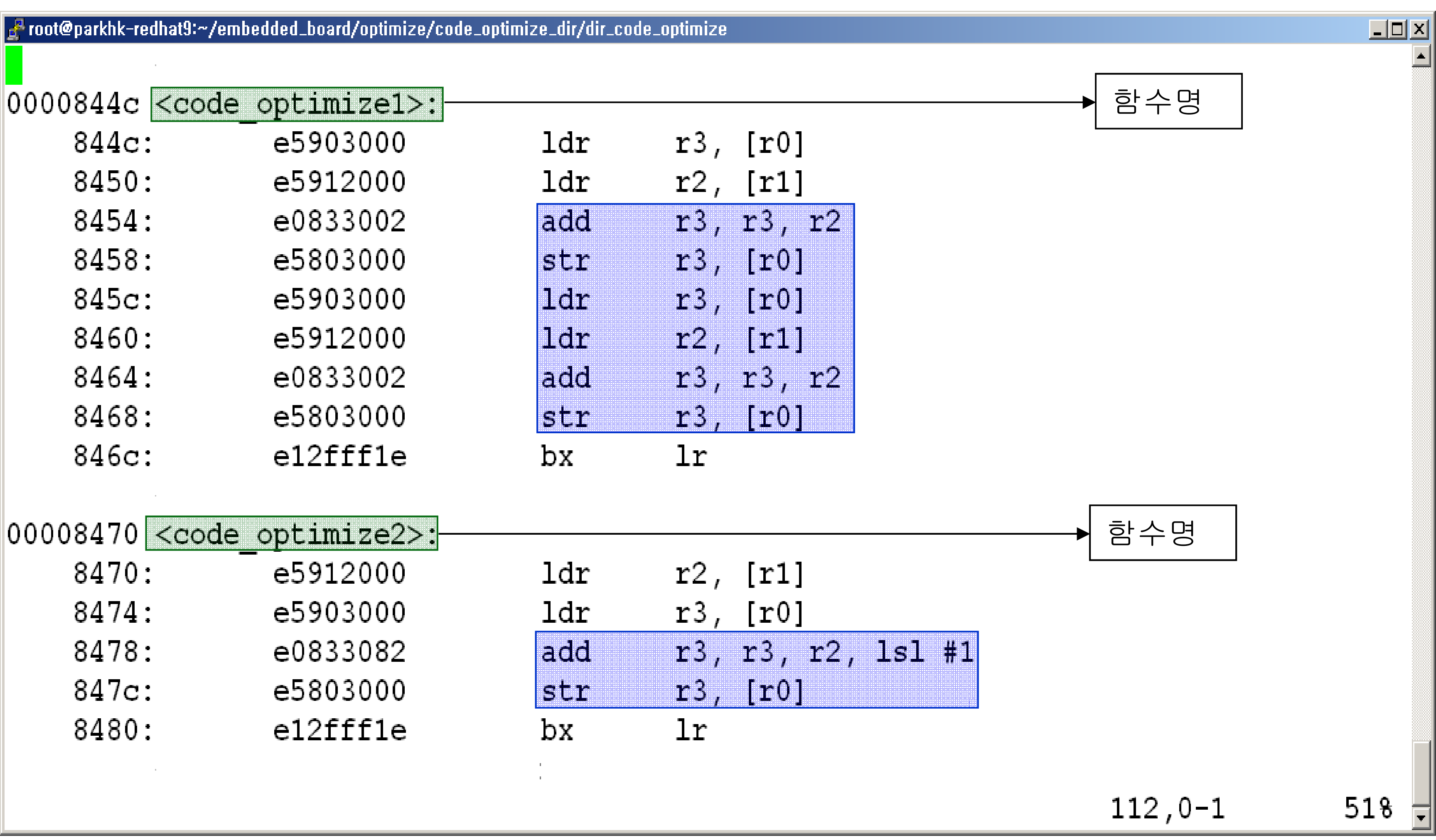
Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 23000nsec, 16000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_code_optimization]$ ./code_optimize

code_optimize1()
Before v1=1, v2=2
Interation : 101
After v1=405, v2=2
0sec 23000nsec

code_optimize1()
Before v1=1, v2=2
Interation : 101
After v1=405, v2=2
0sec 16000nsec

[root@ez-x5 dir_code_optimization]$
```

Code Optimization

- O1옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O1 -lrt -o code_optimize code_optimize.c

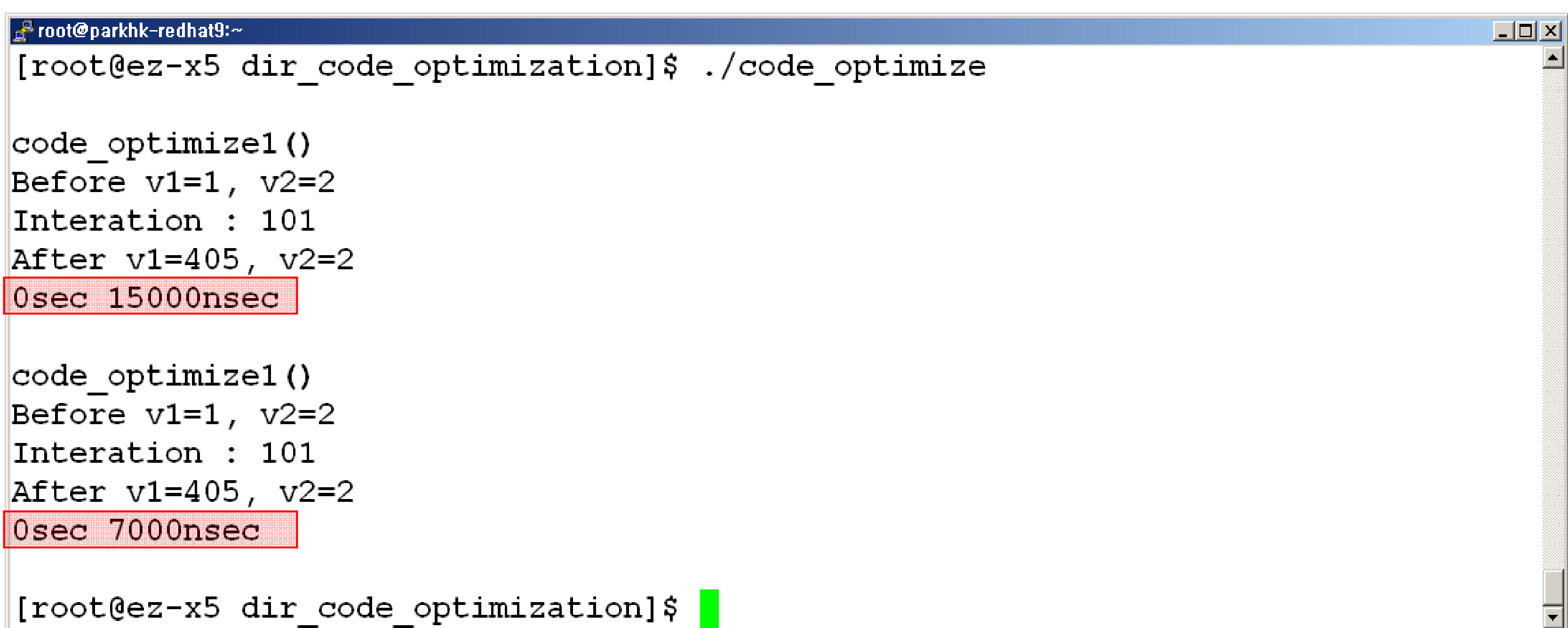


Code Optimization

- O1옵션을 이용한 컴파일 결과
  - ✓ objdump결과 부분이 최적화 전후 코드의 변화를 나타내고 있음
  - ✓ code_optimize1()
    - 결과값 산출 명령어 수행횟수
      - ldr : 4회
      - add : 2회
      - str : 2회
  - ✓ code_optimize2()
    - 결과값 산출 명령어 수행횟수
      - ldr : 2회
      - add : 1회
      - str : 1회

Code Optimization

- O1옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 15000nsec, 7000nsec의 시간이 걸렸음



Code Optimization

- O2옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 15000nsec, 7000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_code_optimization]$ ./code_optimize

code_optimize1()
Before v1=1, v2=2
Interation : 101
After v1=405, v2=2
0sec 14000nsec

code_optimize1()
Before v1=1, v2=2
Interation : 101
After v1=405, v2=2
0sec 7000nsec

[root@ez-x5 dir_code_optimization]$
```

O2옵션을 이용하여 컴파일 한 결과 O1옵션을 이용한 컴파일 결과와 같았음(code_optimize1과 2함수).

Code Optimization

- O3옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 15000nsec, 7000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_code_optimization]$ ./code_optimize

code_optimize1()
Before v1=1, v2=2
Interation : 101
After v1=405, v2=2
0sec 12000nsec

code_optimize1()
Before v1=1, v2=2
Interation : 101
After v1=405, v2=2
0sec 4000nsec

[root@ez-x5 dir_code_optimization]$
```

O3옵션을 이용하여 컴파일 한 결과 O1옵션을 이용한 컴파일 결과와 같았음(code_optimize1과 2함수).

- Code Optimization

예제코드 2

```
void code_opt_swap1(int *v1, int *v2){

        int tmp;

        tmp = *v1;
        *v1 = *v2;
        *v2 = tmp;
}

void code_opt_swap2(int *v1, int *v2){

        (*v1)^=(*v2)^=(*v1)^=(*v2);
}
```

Code Optimization

- 예제 코드 비교
  - ✓ V1의 값과 v2의 값을 swap하는 함수
  - ✓ File size
    - 소스 파일
      - code_opt_swap1() : 224byte
      - code_opt_swap2() : 206byte
    - 실행 파일
      - code_opt_swap1() : 11646byte
      - code_opt_swap2() : 11678byte
    - 컴파일 후 파일의 사이즈는 소스 파일의 라인수에 무관할 수 있음

```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize_swap
[root@parkhk-redhat9 dir_code_optimize_swap]# ls -al
합계 40
drwxr-xr-x    2 root     root         4096 11월  8 13:59 .
drwxr-xr-x    4 root     root         4096 11월  8 13:59 ..
-rwxr-xr-x    1 root     root        11646 11월  7 15:37 optimize_swap1
-rw-r--r--    1 root     root          224 11월  7 15:37 optimize_swap1.c
-rwxr-xr-x    1 root     root        11678 11월  7 15:37 optimize_swap2
-rw-r--r--    1 root     root          206 11월  7 15:37 optimize_swap2.c
[root@parkhk-redhat9 dir_code_optimize_swap]#
```

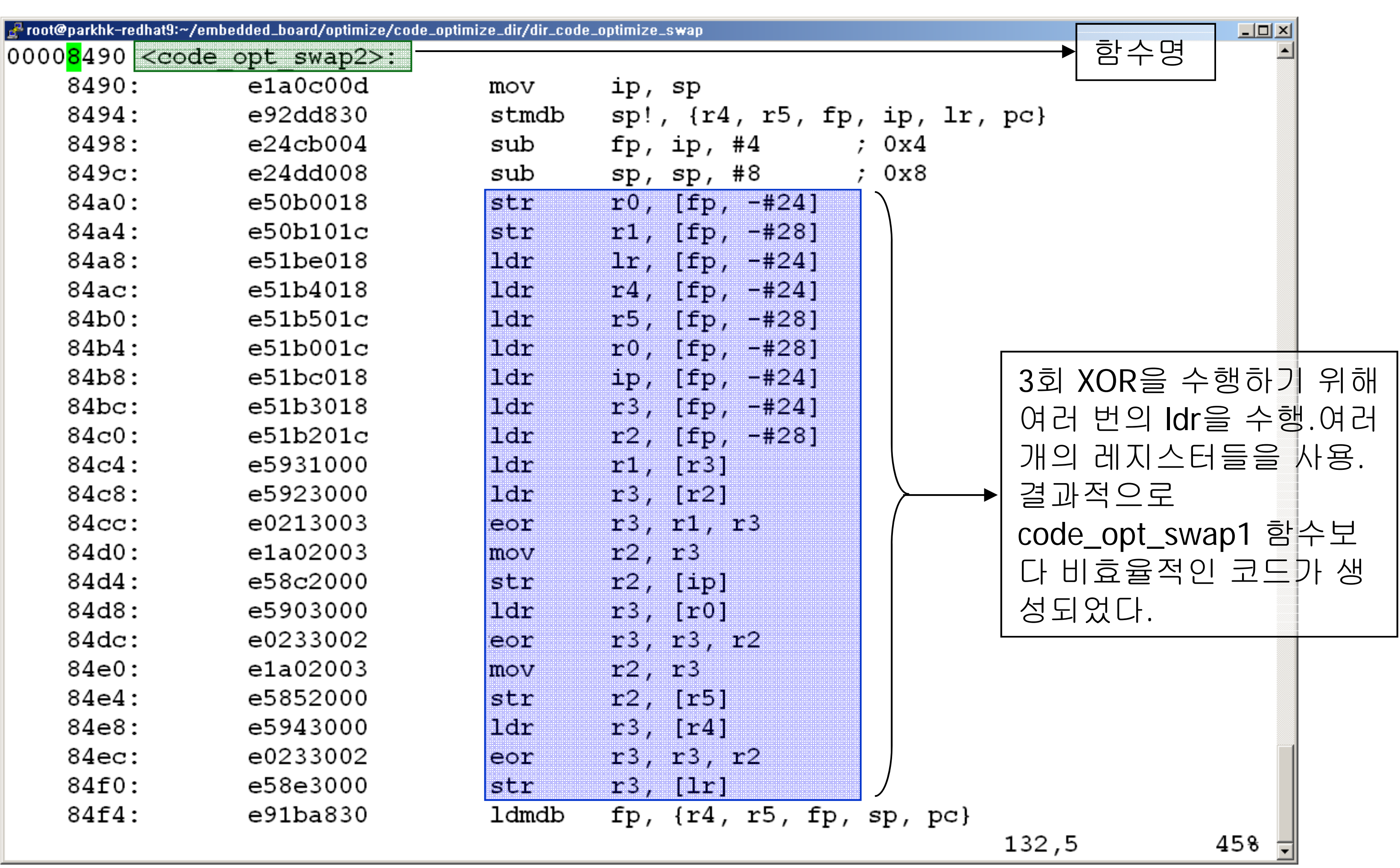
Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o optimize_swap optimize_swap.c

```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize_swap

0000844c <code_opt_swap1>:                                   → 함수명
    844c:       e1a0c00d        mov     ip, sp
    8450:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    8454:       e24cb004        sub     fp, ip, #4          ; 0x4
    8458:       e24dd00c        sub     sp, sp, #12         ; 0xc
    845c:       e50b0010        str     r0, [fp, -#16]
    8460:       e50b1014        str     r1, [fp, -#20]
    8464:       e51b3010        ldr     r3, [fp, -#16]
    8468:       e5933000        ldr     r3, [r3]
    846c:       e50b3018        str     r3, [fp, -#24]
    8470:       e51b2010        ldr     r2, [fp, -#16]
    8474:       e51b3014        ldr     r3, [fp, -#20]
    8478:       e5933000        ldr     r3, [r3]
    847c:       e5823000        str     r3, [r2]
    8480:       e51b2014        ldr     r2, [fp, -#20]
    8484:       e51b3018        ldr     r3, [fp, -#24]
    8488:       e5823000        str     r3, [r2]
    848c:       e91ba800        ldmdb   fp, {fp, sp, pc}
                                                        112,0-1       37%
```

Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o optimize_swap optimize_swap.c

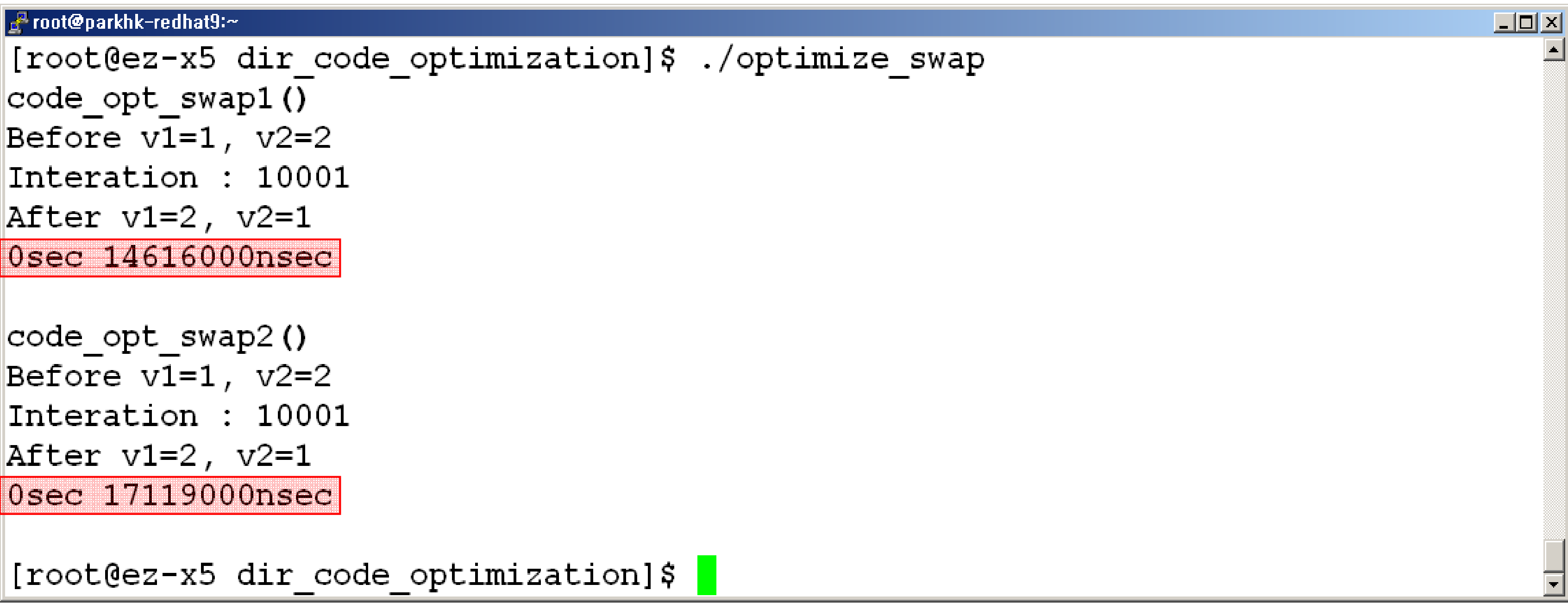


Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ objdump결과 부분이 최적화 전후 코드의 변화를 나타내고 있음
  - ✓ code_opt_swap1()
    - 명령어 수행횟수
      - ldr : 7회
      - str : 5회
  - ✓ code_opt_swap2()
    - 명령어 수행횟수
      - ldr : 11회
      - eor : 3회
      - mov : 2회
      - str : 5회

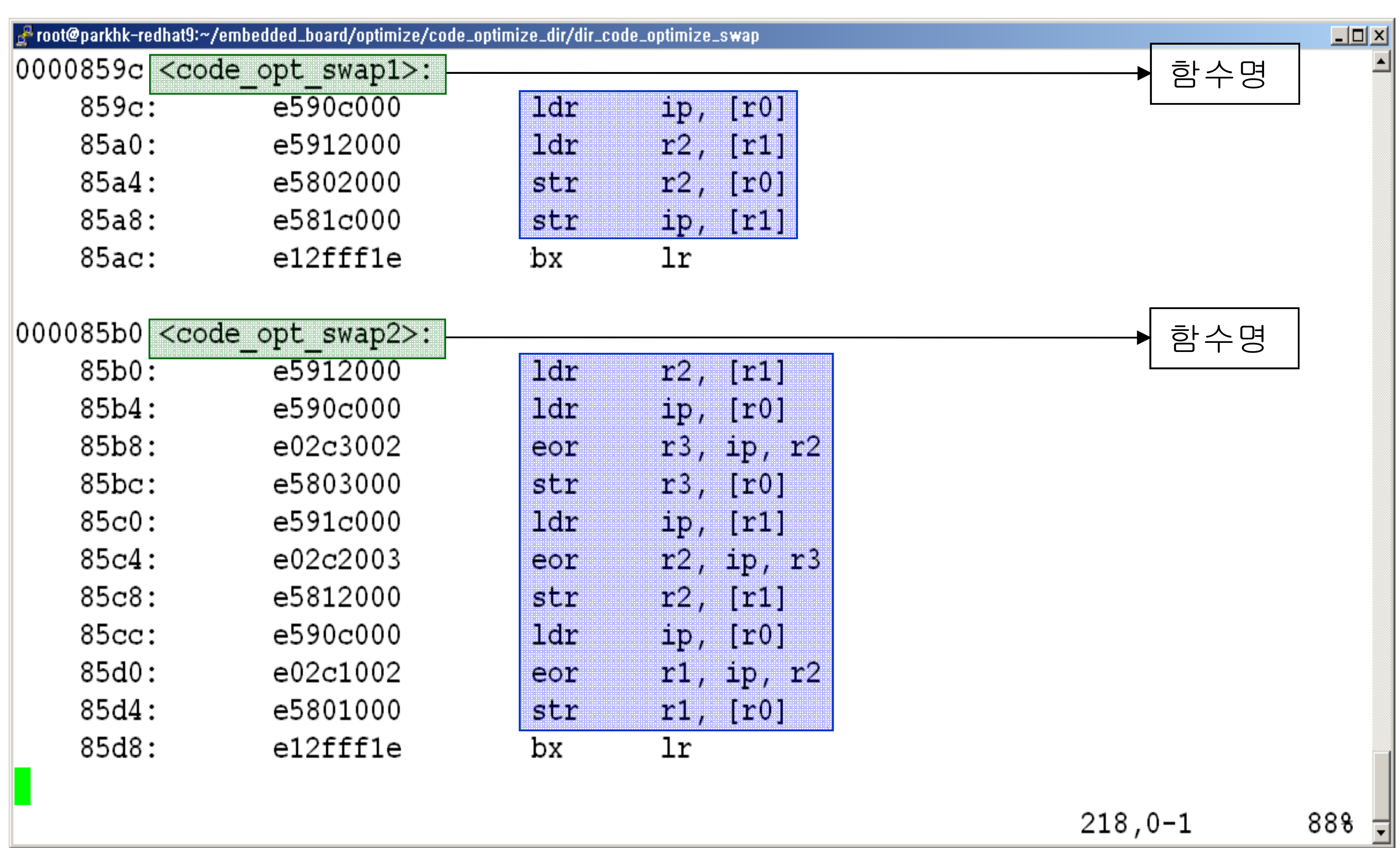
Code Optimization

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 14616000nsec, 17119000nsec의 시간이 걸렸음



Code Optimization

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o optimize_swap optimize_swap.c

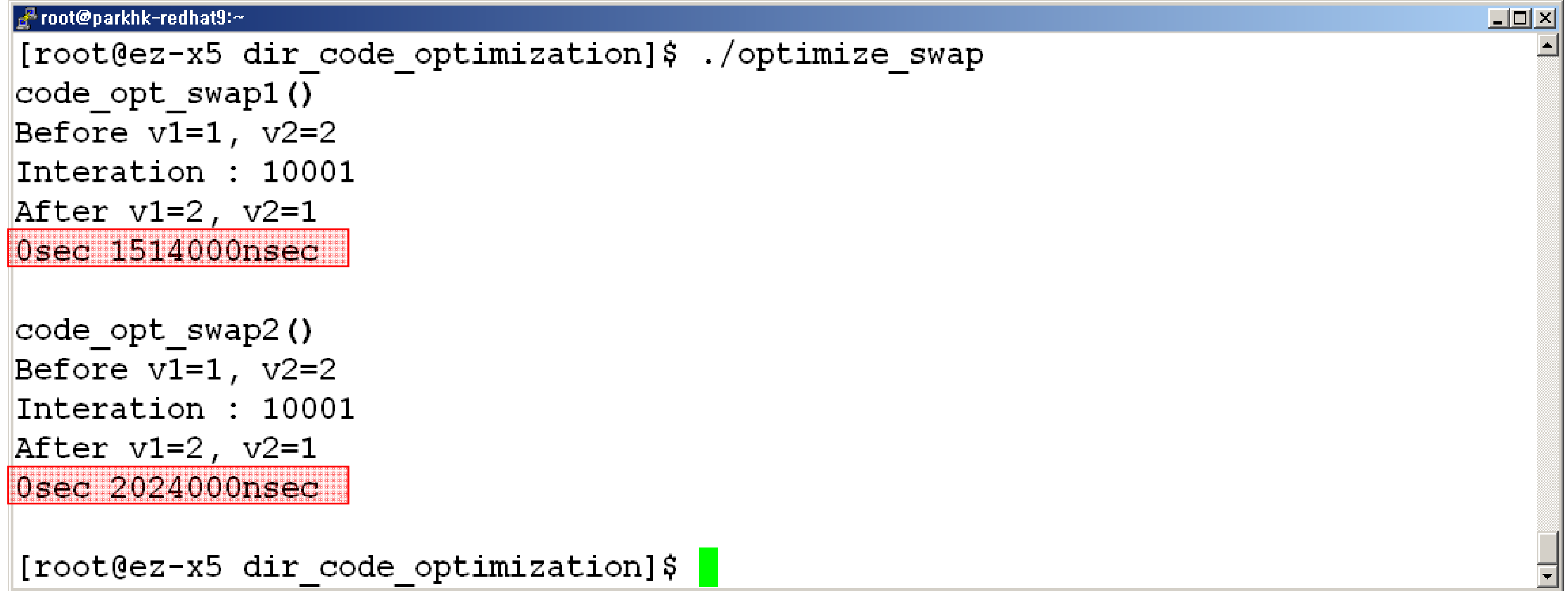


Code Optimization

- O3옵션을 이용한 컴파일 결과
  - ✓ objdump결과 ▒부분이 최적화 전후 코드의 변화를 나타내고 있음
  - ✓ code_opt_swap1()
    - 명령어 수행횟수
      - ldr : 2회
      - str : 2회
  - ✓ code_opt_swap2()
    - 명령어 수행횟수
      - ldr : 4회
      - eor : 3회
      - str : 4회

Code Optimization

- O3옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 1514000nsec, 2024000nsec의 시간이 걸렸음
    - O0옵션으로 컴파일한 경우에 비하여 약 10정도 성능 향상을 보이고 있음



Code Optimization

- Code Optimization
  - ✓ 컴파일 후 명령어 수행 횟수를 최소화 함으로써 성능을 높일 수 있음.
  - ✓ 일반적으로는 길이가 짧은 코드가 적은 명령어를 수행하지만 항상 그러하지는 않음.
  - ✓ 의도적으로 delay를 주는 경우가 아니라면 –O3옵션이 최적화에 큰 역할을 할 수 있음

코드 최적화 기법

- Loop Fusion
- Loop Unrolling
- Loop Invariant Code Motion
- Strength Reduction
- Count up to Zero

Loop Fusion

- Loop Fusion
  - ✓ Loop 최적화 기법
  - ✓ Loop Jamming 이라고도 함
  - ✓ 반복문은 다른 연산에 비하여 상대적으로 많은 cycle을 요구
  - ✓ 반복문의 개수를 최소화 하는 것이 성능향상에 도움이 됨
  - ✓ Fusion이 가능한 반복문에 대하여 루프를 하나로 결합

- Loop Fusion

예제코드

```
/* original source code*/

int array_v1[ARRAY_SIZE];
int array_v2[ARRAY_SIZE];

int loop_fusion1(void){

        int i;
        for(i=0;i<ARRAY_SIZE;i+2){
                array_v1[i] = i;
        }

        for(i=0;i<ARRAY_SIZE;i+2){
                array_v2[i] = i;
        }
}
```

Loop Fusion

- Loop Fusion

예제코드

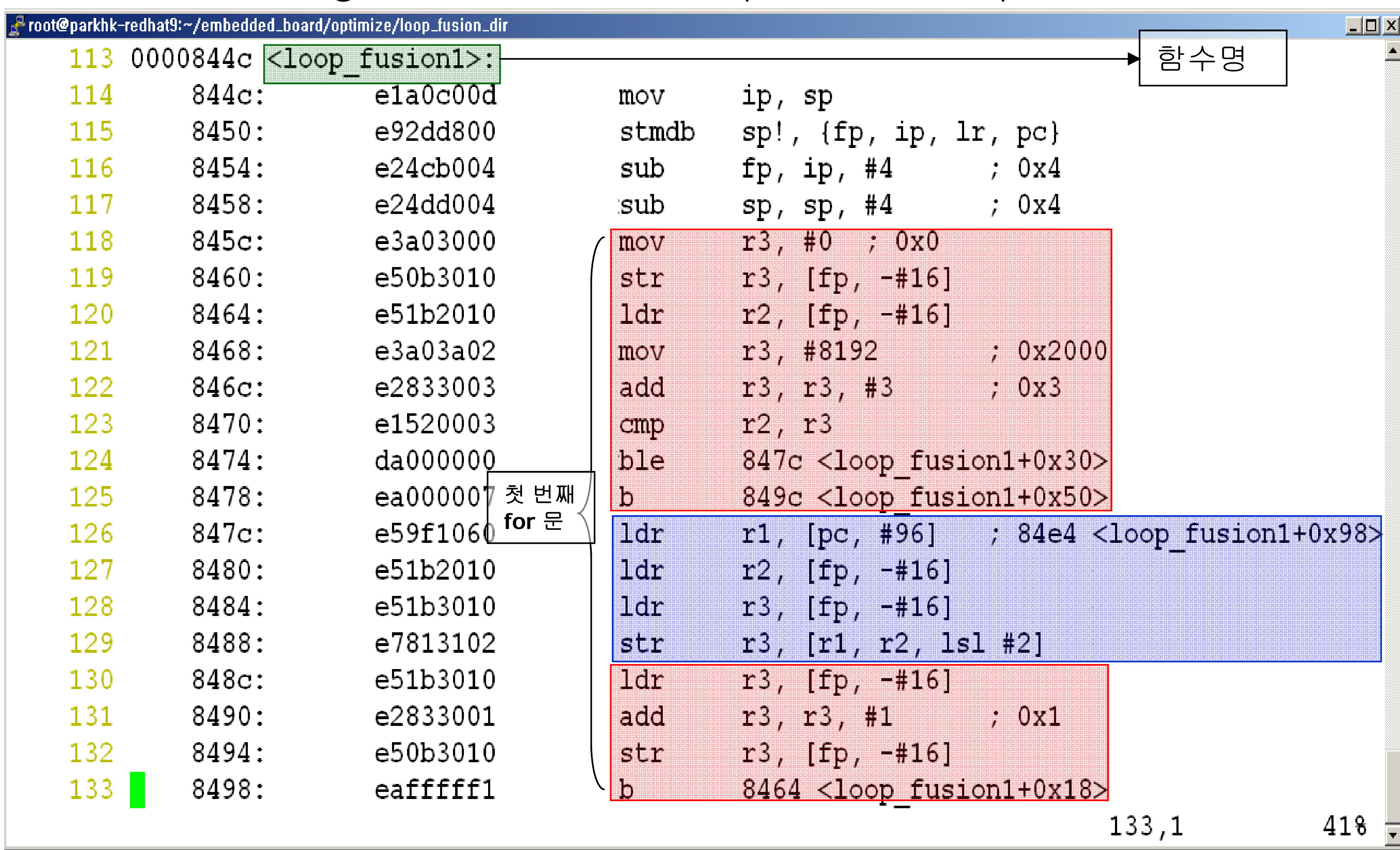
```
/* Loop Fusion technique */

int array_v1[ARRAY_SIZE];
int array_v2[ARRAY_SIZE];

int loop_fusion2(void){

        int i;
        for(i=0;i<ARRAY_SIZE;i+2){

                array_v1[i] = i;
                array_v2[i] = i;
        }
}
```

Loop Fusion

- 예제 코드 비교
  - ✓ Loop 반복 횟수
    - loop_fusion1
      - ARRAY_SIZE * 2
    - loop_fusion2
      - ARRAY_SIZE
  - ✓ 불필요한 루프를 줄임으로써 상대적으로 적은 cycle안에 코드를 수행
  - ✓ 특이 사항
    - 전역 변수 사용시 명시적으로 초기화를 해주지 않으면 zeroed page에 mapping된다. 그래서 전역변수를 처음 접근하게 되면 page fault를 유발하게 되며 그때서야 메모리를 할당 받게 된다. 최적화 코드에 공정성을 기하기 위해 전역 변수들은 미리 한번 초기화를 수행 하였다. 본 모듈에서 사용하는 모든 소스는 배열 초기화를 미리 한 후 테스트 하였다.

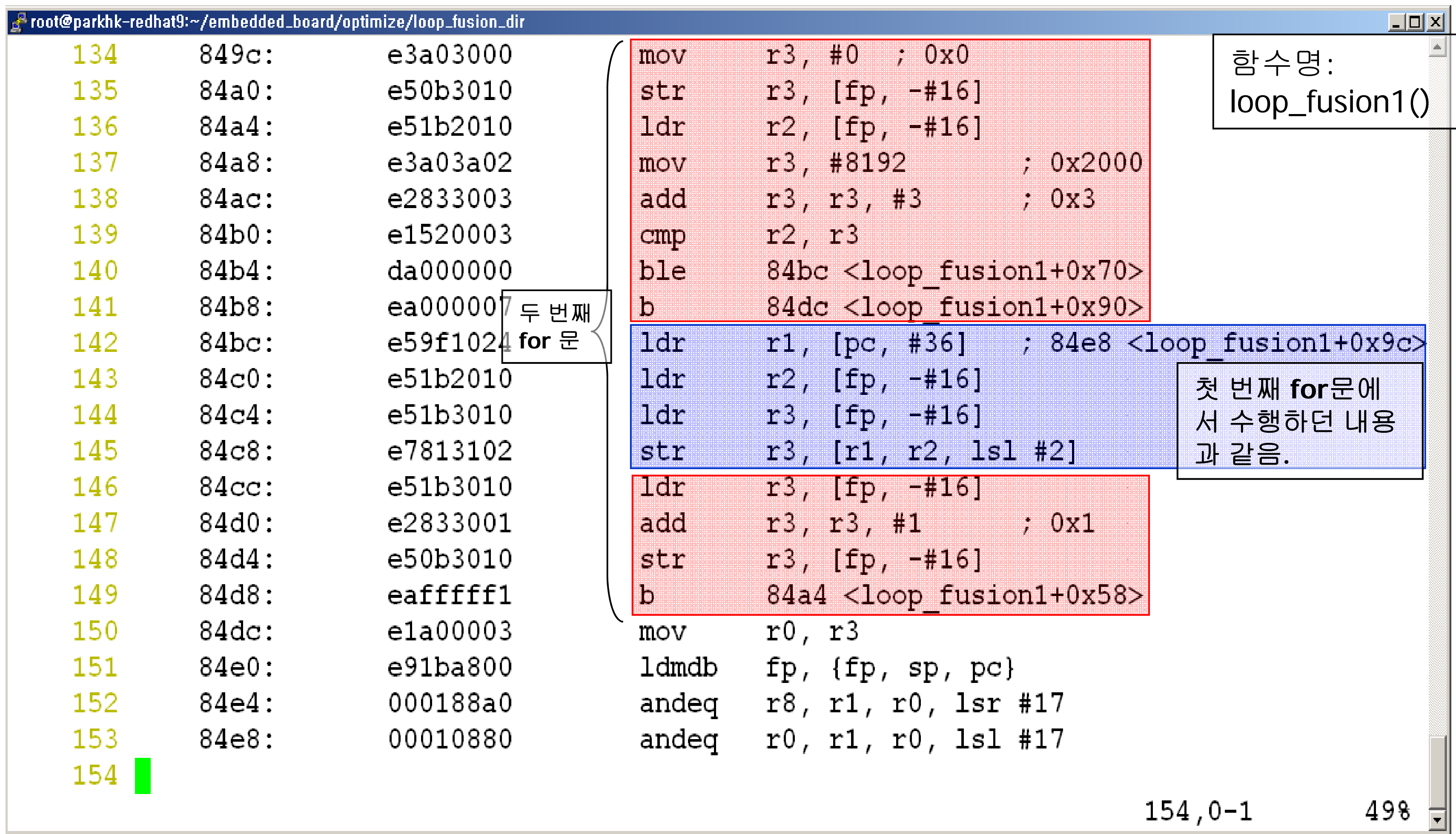
Loop Fusion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c



Loop Fusion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c



Loop Fusion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c

root@parkhk-redhat9:~/embedded_board/optimize/loop_fusion_dir

```
155 000084ec <loop_fusion2>:
156     84ec:       e1a0c00d        mov     ip, sp
157     84f0:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
158     84f4:       e24cb004        sub     fp, ip, #4          ; 0x4
159     84f8:       e24dd004        sub     sp, sp, #4          ; 0x4
160     84fc:       e3a03000        mov     r3, #0   ; 0x0
161     8500:       e50b3010        str     r3, [fp, -#16]
162     8504:       e51b2010        ldr     r2, [fp, -#16]
163     8508:       e3a03a02        mov     r3, #8192           ; 0x2000
164     850c:       e2833003        add     r3, r3, #3          ; 0x3
165     8510:       e1520003        cmp     r2, r3
166     8514:       da000000        ble     851c <loop_fusion2+0x30>
167     8518:       ea00000b        b       854c <loop_fusion2+0x60>
168     851c:       e59f1030        ldr     r1, [pc, #48]       ; 8554 <loop_fusion2+0x68>
169     8520:       e51b2010        ldr     r2, [fp, -#16]
                                                   169,1          56%
```

함수명

Loop Fusion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c

root@parkhk-redhat9:~/embedded_board/optimize/loop_fusion_dir

```
170    8524:    e51b3010    ldr     r3, [fp, -#16]
171    8528:    e7813102    str     r3, [r1, r2, lsl #2]
172    852c:    e59f1024    ldr     r1, [pc, #36]    ; 8558 <loop_fusion2+0x6c>
173    8530:    e51b2010    ldr     r2, [fp, -#16]
174    8534:    e51b3010    ldr     r3, [fp, -#16]
175    8538:    e7813102    str     r3, [r1, r2, lsl #2]
176    853c:    e51b3010    ldr     r3, [fp, -#16]
177    8540:    e2833001    add     r3, r3, #1       ; 0x1
178    8544:    e50b3010    str     r3, [fp, -#16]
179    8548:    eaffffed    b       8504 <loop_fusion2+0x18>
180    854c:    e1a00003    mov     r0, r3
181    8550:    e91ba800    ldmdb   fp, {fp, sp, pc}
182    8554:    000188a0    andeq   r8, r1, r0, lsr #17
183    8558:    00010880    andeq   r0, r1, r0, lsl #17
184
                                              184,0-1       61%
```

함수명: loop_fusion2()

중복되는 loop 수행을 줄임으로써 코드가 간결화 되었음

Loop Fusion

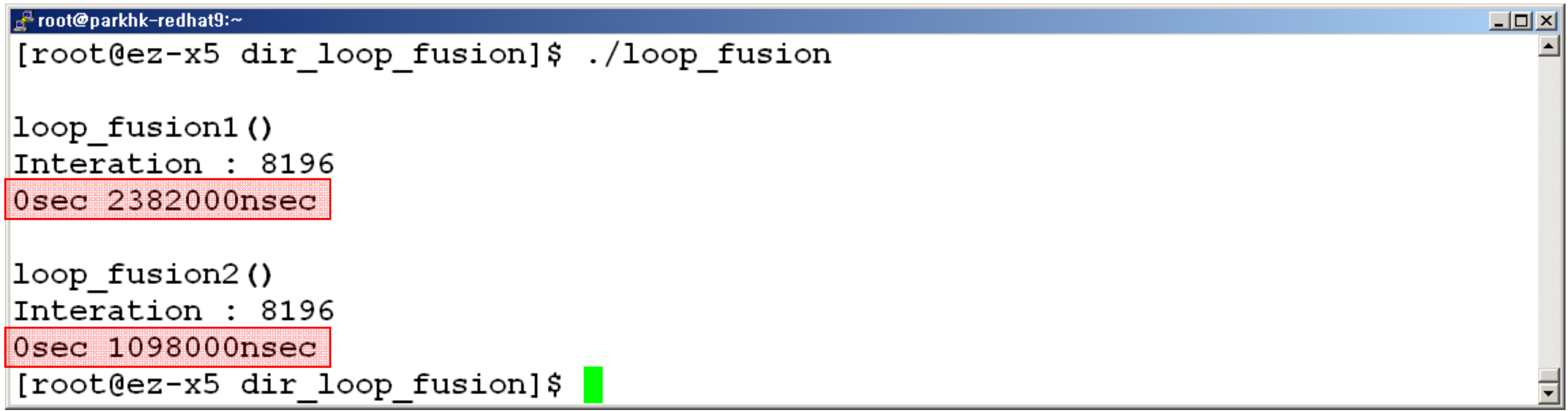
- O0옵션을 이용한 컴파일 결과
  - ✓ ▢부분은 for문을 수행하기 위한 arm assembly이고 ▢부분은 배열에 값을 초기화 해주는 부분이다.

- ✓ loop_fusion1()
  - for loop 수행 코드
    - mov, str, ldr, add, b : 4회
    - cmp, ble : 2회
  - 배열 값 초기화 수행 코드
    - ldr : 6회
    - str : 2회

- ✓ loop_fusion2()
  - for loop 수행 코드
    - mov, str, ldr, add, b : 2회
    - cmp, ble : 1회
  - 배열 값 초기화 수행 코드
    - ldr : 6회
    - str : 2회

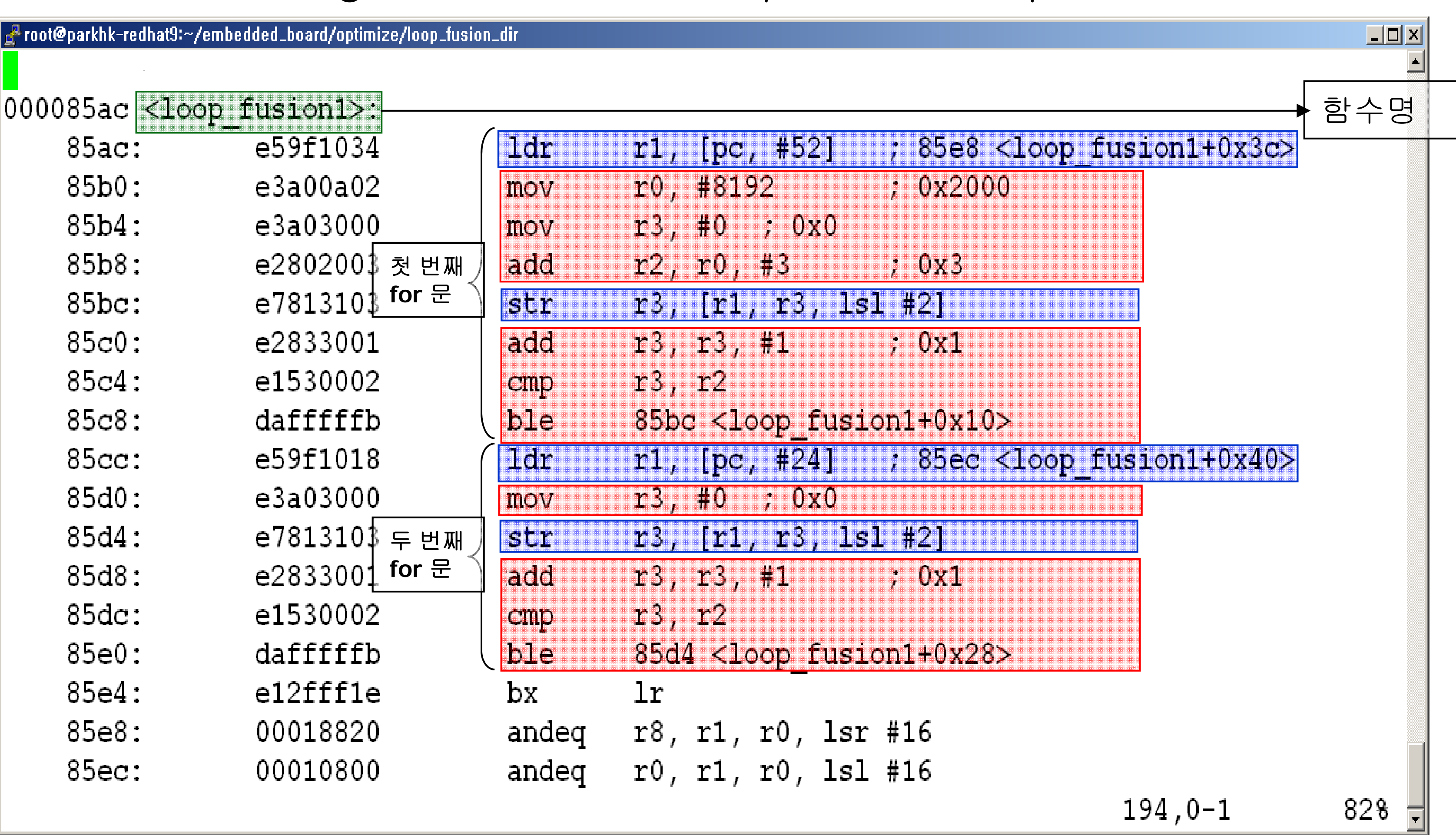
Loop Fusion

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 2382000nsec, 1098000nsec의 시간이 걸렸음



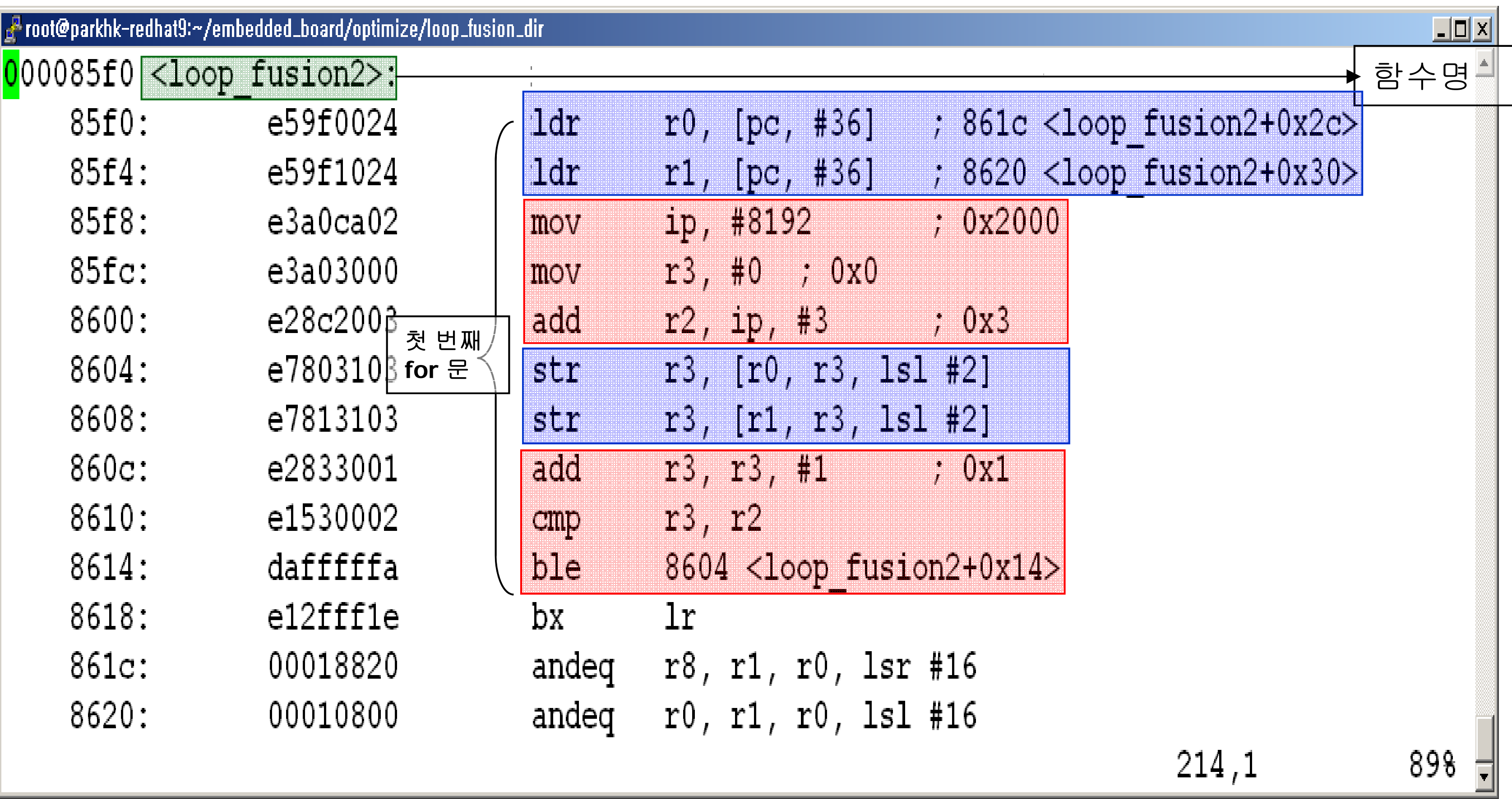
Loop Fusion

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o loop_fusion loop_fusion.c



Loop Fusion

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c



Loop Fusion

- O3옵션을 이용한 컴파일 결과
  - ✓ 부분은 for문을 수행하기 위한 arm assembly이고 부분은 배열에 값을 초기화 해주는 부분이다.
  - ✓ loop_fusion1()
    - for loop 수행 코드
      - mov, add: 3회
      - cmp, ble : 2회
    - 배열 값 초기화 수행 코드
      - ldr : 2회
      - str : 2회
  - ✓ loop_fusion2()
    - for loop 수행 코드
      - mov, add : 2회
      - cmp, ble : 1회
    - 배열 값 초기화 수행 코드
      - ldr : 2회
      - str : 2회
  - ✓ 위의 명령어들이 반복횟수 만큼 수행됨

Loop Fusion

- O3옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 2068000nsec, 792000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_loop_fusion]$ ./loop_fusion

loop_fusion1()
Interation : 8196
0sec 2068000nsec

loop_fusion2()
Interation : 8196
0sec 792000nsec
[root@ez-x5 dir_loop_fusion]$
```

Loop Fusion

- Loop Fusion
  - ✓ 수행 시간
    - 여러 개의 반복문을 하나로 묶음으로써 Loop수행에 따른 cycle을 절약할 수 있다
  - ✓ 효과적인 cache사용으로 인한 성능상 이득을 얻을 수 있음

Loop Unrolling

- Loop unrolling
  - ✓ Loop 최적화 기법
  - ✓ 반복문이 포함하고 있는 원래의 코드를 여러 번 반복하여 작성
  - ✓ 반복문에서 루프 수행 횟수를 감소 시키는 기법
  - ✓ 배열의 크기가 unrolling의 양의 배수가 되도록 작성
    - 배열의 크기가 unrolling양에 비례할 경우 가장 좋은 성능을 보일 수 있음
    - 양의 배수가 안될 경우 루프 바깥쪽에서 unrolling하는 것을 고려

Loop Unrolling

- Loop unrolling

예제코드

```
/* original source code*/

int array_v[ARRAY_SIZE];

int loop_unrolling1(void){

        int i;

        for(i=0; i<ARRAY_SIZE ; i++){
                array_v[i] = i;
        }
}
```

Loop Unrolling

- Loop unrolling

예제코드

```
/*Loop unrolling technique*/

int array_v[ARRAY_SIZE];

int loop_unrolling2(void){

	int i;

	for(i=0; i<ARRAY_SIZE ; i+=2){
		array_v[i] = i;
		array_v[i+1] = i+1;
	}
}
```

Loop Unrolling

- 예제 코드 비교
  - ✓ Loop 반복 횟수
    - loop_unrolling1
      - ARRAY_SIZE
    - loop_unrolling2
      - ARRAY_SIZE/2
  - ✓ 등차수열의 형태를 갖는 반복문에서 배열의 초기화 수행 시 배열을 unrolling을 함으로써 반복 횟수를 줄임

Loop Unrolling

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o loop_unrolling loop_unrolling.c

root@parkhk-redhat9:~/embedded_board/optimize/loop_unrolling_dir

```
0000844c <loop_unrolling1>:
    844c:       e1a0c00d        mov     ip, sp
    8450:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    8454:       e24cb004        sub     fp, ip, #4        ; 0x4
    8458:       e24dd004        sub     sp, sp, #4        ; 0x4
    845c:       e3a03000        mov     r3, #0  ; 0x0
    8460:       e50b3010        str     r3, [fp, -#16]
    8464:       e51b2010        ldr     r2, [fp, -#16]
    8468:       e3a03a02        mov     r3, #8192         ; 0x2000
    846c:       e2833003        add     r3, r3, #3        ; 0x3
    8470:       e1520003        cmp     r2, r3
    8474:       da000000        ble     847c <loop_unrolling1+0x30>
    8478:       ea000007        b       849c <loop_unrolling1+0x50>
    847c:       e59f1020        ldr     r1, [pc, #32]     ; 84a4 <loop_unrolling1+0x58>
    8480:       e51b2010        ldr     r2, [fp, -#16]
    8484:       e51b3010        ldr     r3, [fp, -#16]
    8488:       e7813102        str     r3, [r1, r2, lsl #2]
    848c:       e51b3010        ldr     r3, [fp, -#16]
    8490:       e2833001        add     r3, r3, #1        ; 0x1
    8494:       e50b3010        str     r3, [fp, -#16]
    8498:       eaffff1         b       8464 <loop_unrolling1+0x18>
    849c:       e1a00003        mov     r0, r3
    84a0:       e91ba800        ldmdb   fp, {fp, sp, pc}
    84a4:       00010e20        andeq   r0, r1, r0, lsr #28

                                                        113,5          17%
```

Loop Unrolling

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc -O0 -lrt -o loop_unrolling loop_unrolling.c

root@parkhk-redhat9:~/embedded_board/optimize/loop_unrolling_dir

```
000084a8 <loop_unrolling2>:
    84a8:       e1a0c00d        mov     ip, sp
    84ac:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    84b0:       e24cb004        sub     fp, ip, #4          ; 0x4
    84b4:       e24dd004        sub     sp, sp, #4          ; 0x4
    84b8:       e3a03000        mov     r3, #0   ; 0x0
    84bc:       e50b3010        str     r3, [fp, -#16]
    84c0:       e51b2010        ldr     r2, [fp, -#16]
    84c4:       e3a03a02        mov     r3, #8192           ; 0x2000
    84c8:       e2833003        add     r3, r3, #3          ; 0x3
    84cc:       e1520003        cmp     r2, r3
    84d0:       da000000        ble     84d8 <loop_unrolling2+0x30>
    84d4:       ea000010        b       851c <loop_unrolling2+0x74>
    84d8:       e59f1044        ldr     r1, [pc, #68]       ; 8524 <loop_unrolling2+0x7c>
    84dc:       e51b2010        ldr     r2, [fp, -#16]
    84e0:       e51b3010        ldr     r3, [fp, -#16]
    84e4:       e7813102        str     r3, [r1, r2, lsl #2]
    84e8:       e59f1034        ldr     r1, [pc, #52]       ; 8524 <loop_unrolling2+0x7c>
    84ec:       e51b3010        ldr     r3, [fp, -#16]
    84f0:       e3a02004        mov     r2, #4   ; 0x4
    84f4:       e1a03103        mov     r3, r3, lsl #2
    84f8:       e0833001        add     r3, r3, r1
    84fc:       e0832002        add     r2, r3, r2
    8500:       e51b3010        ldr     r3, [fp, -#16]
    8504:       e2833001        add     r3, r3, #1          ; 0x1
    8508:       e5823000        str     r3, [r2]
    850c:       e51b3010        ldr     r3, [fp, -#16]
    8510:       e2833002        add     r3, r3, #2          ; 0x2
    8514:       e50b3010        str     r3, [fp, -#16]
    8518:       eaffffe8        b       84c0 <loop_unrolling2+0x18>
    851c:       e1a00003        mov     r0, r3
    8520:       e91ba800        ldmdb   fp, {fp, sp, pc}
    8524:       00010e20        andeq   r0, r1, r0, lsr #28
                                                    137,0-1       22%
```

함수명

8196/2회 for문을 수행

Loop Unrolling

- O0옵션을 이용한 컴파일 결과
  - ✓ ▒부분은 for문을 수행하기 위한 arm assembly임
  - ✓ loop_unrolling1()
    - for loop 수행 코드
      - mov, str, ldr, add, b : 2회
      - cmp, ble : 1회
    - 배열 값 초기화 수행 코드
      - ldr : 3회
      - str : 1회
    - 반복횟수
      - 8196
  - ✓ loop_unrolling2()
    - for loop 수행 코드
      - mov, str, ldr, add, b : 2회
      - cmp, ble : 1회
    - 배열 값 초기화 수행 코드
      - ldr : 6회
      - str : 2회
      - mov : 2회
      - add : 3회
    - 반복횟수
      - 4098회

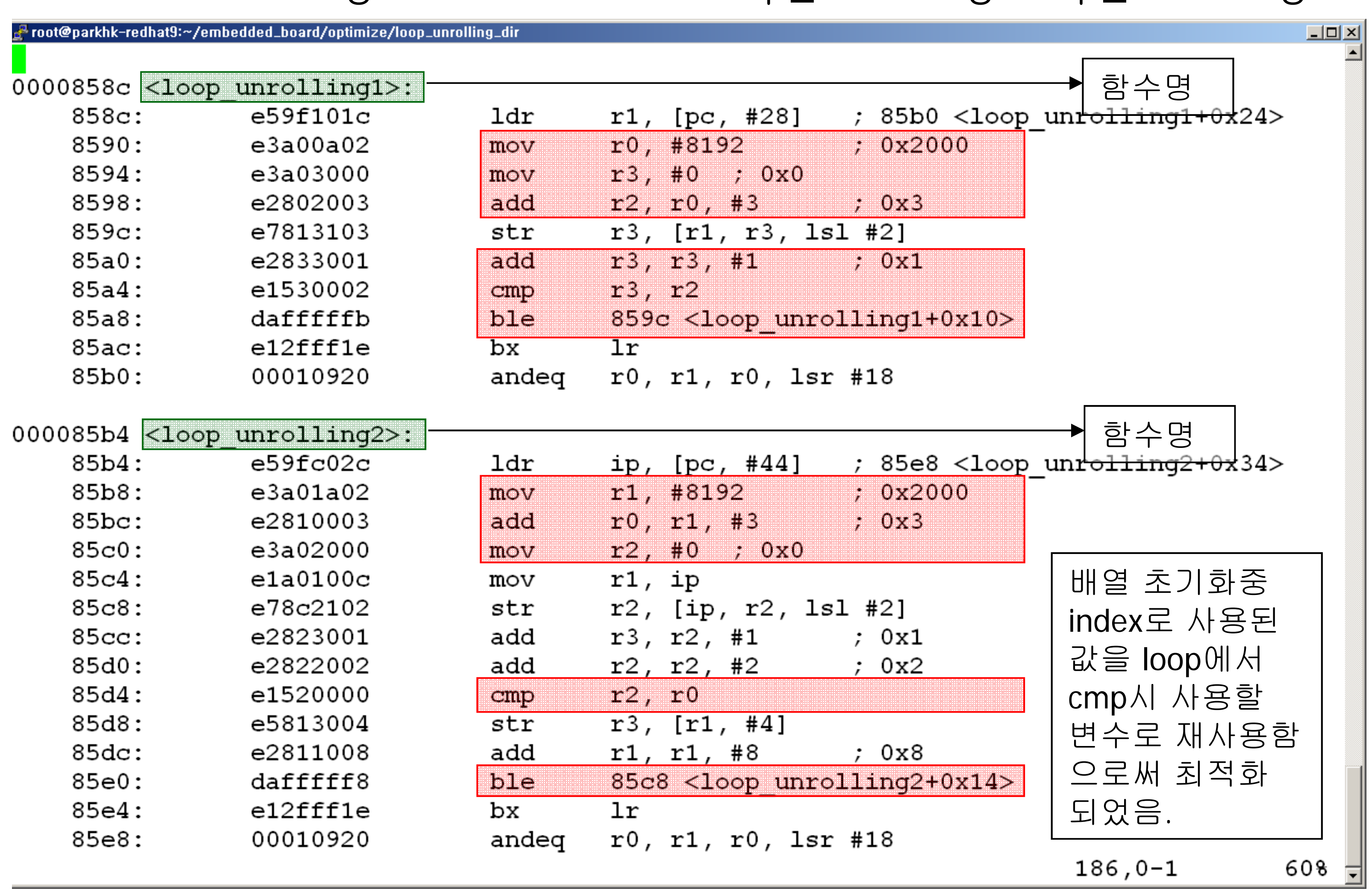
Loop Unrolling

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 1199000nsec, 637000nsec의 시간이 걸렸음

```
[root@ez-x5 dir_loop_unrolling]$ ./loop_unrolling

loop_unrolling1()
Interation : 8196
0sec 1199000nsec

loop_unrolling1()
Interation : 8196
0sec 637000nsec
[root@ez-x5 dir_loop_unrolling]$
```

Loop Unrolling

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt –o loop_unrolling loop_unrolling.c



Loop Unrolling

- O3옵션을 이용한 컴파일 결과
  - ✓ 부분은 for문을 수행하기 위한 arm assembly임

- ✓ loop_fusion1()
  - for loop 수행 코드
    - mov, add : 2회
    - cmp, ble : 1회
  - 배열 값 초기화 수행 코드
    - ldr : 1회
    - str : 1회
  - 반복횟수
    - 8196

- ✓ loop_fusion2()
  - for loop 수행 코드
    - mov: 2회
    - add, cmp, ble : 1회
  - 배열 값 초기화 수행 코드
    - ldr : 1회
    - str : 2회
    - mov : 1회
    - add : 3회
  - 반복횟수
    - 4098회

Loop Unrolling

- O3옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 968000nsec, 441000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_loop_unrolling]$ ./loop_unrolling

loop_unrolling1()
Interation : 8196
0sec 968000nsec

loop_unrolling1()
Interation : 8196
0sec 441000nsec
[root@ez-x5 dir_loop_unrolling]$
```

Loop Unrolling

- unrolling 개수와 최적화 그리고 프로그램 사이즈의 관계
  - ✓ 시간

```
root@parkhk-redhat9:~
[root@ez-x5 nfs]$ ./loop_unrolling
 1 loop unrolling 0sec 1183000nsec
 2 loop unrolling 0sec 628000nsec
 4 loop unrolling 0sec 600000nsec
 8 loop unrolling 0sec 575000nsec
16 loop unrolling 0sec 574000nsec
[root@ez-x5 nfs]$
```

  - unrolling 개수에 비례하여 실행시간이 줄어 들지는 않음

Loop Unrolling

- unrolling 개수와 최적화 그리고 프로그램 사이즈의 관계
  - ✓ 파일 크기

```
root@parkhk-redhat9:~
[root@ez-x5 nfs]$ ls -l
-rwxr-xr-x    1 root     root        13951 Nov  5  2007 loop_unrolling
-rwxr-xr-x    1 root     root        11967 Nov  5  2007 loop_unrolling1
-rwxr-xr-x    1 root     root        12513 Nov  5  2007 loop_unrolling16
-rwxr-xr-x    1 root     root        12063 Nov  5  2007 loop_unrolling4
-rwxr-xr-x    1 root     root        12223 Nov  5  2007 loop_unrolling8
[root@ez-x5 nfs]$
```

    - unrolling 개수에 비례하여 파일의 크기가 커짐

Loop Unrolling

- Loop Unrolling
  - ✓ 수행시간
    - unrolling 개수에 비례하지 않음
    - register이용 개수를 고려하여 작성해야 최적화의 의미가 있음
      - unrolling을 많이 할수록 필요한 register의 개수가 많아짐
  - ✓ 메모리 사용률
    - Unrolling의 깊이가 증가 할수록 메모리 사용량 증가

Loop Invariant Code Motion

- Loop Invariant Code Motion
  - ✓ Loop 최적화 기법
  - ✓ Loop내에 불변하는 계산들은 오버헤드를 유발함
  - ✓ Loop내에서 불변하는 계산 값이나 함수를 코드 바깥쪽으로 분리
    - Example
      - srtlen()이 비교문에 들어가는 경우
      - 변하지 않는 사칙 연산 값
  - ✓ loop내에 불변의 값을 상수 값으로 대체

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

```
/* original source code (calculation)*/

int array_v[ARRAY_SIZE];

int loop_ICM1(int v1, int v2, int v3){

        int i;

        for(i=0; i<ARRAY_SIZE ; i++){
                array_v[i] = ((v1*v1)+(v2*v2)+(v3*v3)+ARRAY_SIZE);
        }
}
```

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

```
/*loop ICM technique (calculation) */

int array_v[ARRAY_SIZE];

int loop_ICM2(int v1, int v2, int v3){

        int i;
        int tmp;

        tmp = ((v1*v1)+(v2*v2)+(v3*v3)+ARRAY_SIZE);


        for(i=0; i<ARRAY_SIZE ; i++){
                array_v[i] = tmp;
        }
}
```

Loop Invariant Code Motion

- 예제 코드 비교
  - ✓ loop_ICM1()
    - ARRAY_SIZE횟수 loop를 수행하면서 매번 매개변수들의 제곱에 ARRAY_SIZE를 더하여 배열을 초기화 ((v1*v1)+(v2*v2)+(v3*v3)+ARRAY_SIZE)를 계산하여 배열에 저장
  - ✓ loop_ICM2()
    - 매개변수들의 제곱에 ARRAY_SIZE를 loop위에서 미리 계산하여 loop내에서 계산과정을 생략
  - ✓ 불필요한 계산과정을 생략 함으로써 코드를 최적화

Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o loop_ICM loop_ICM.c

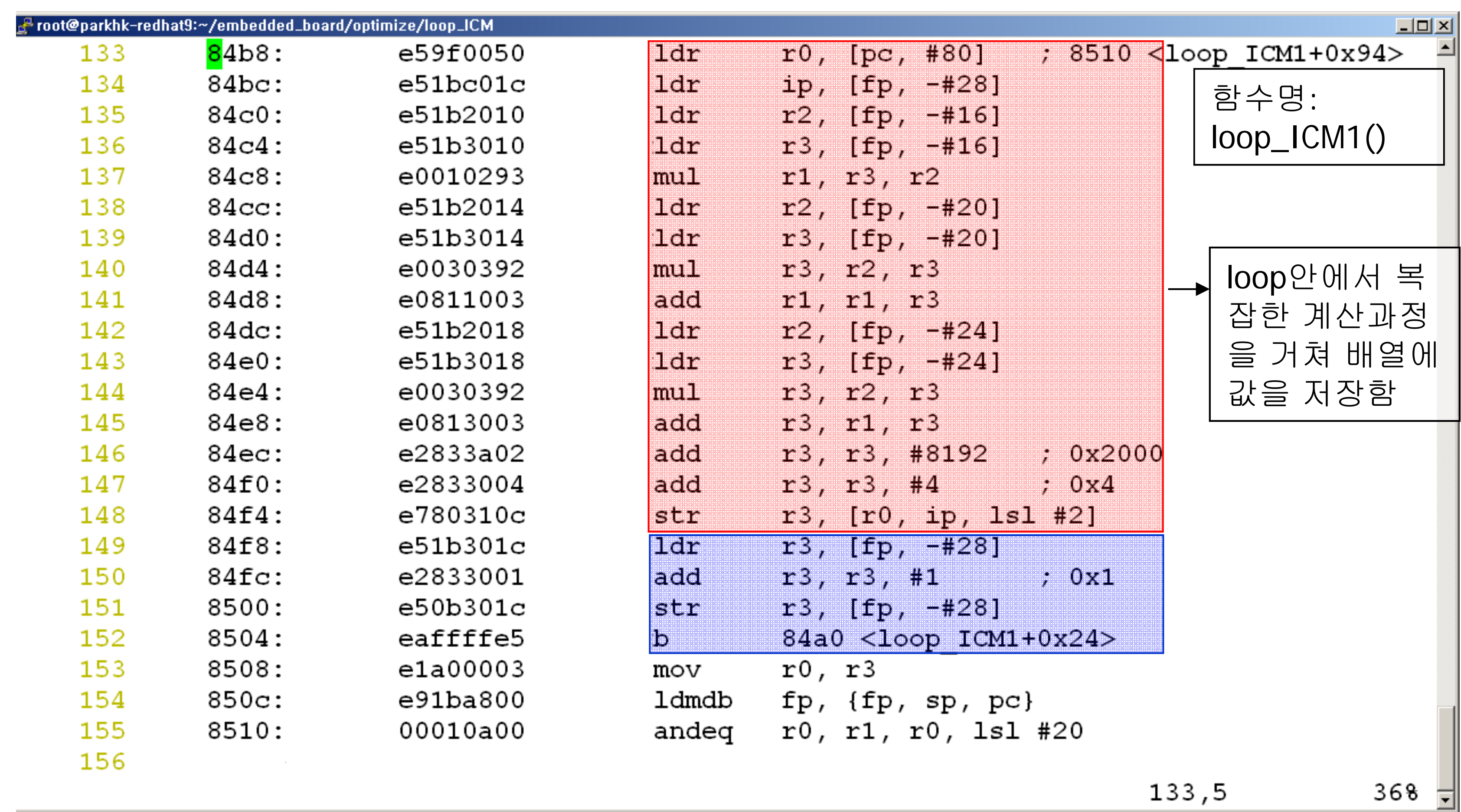
```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
116
117 0000847c <loop_ICM1>:
118     847c:       e1a0c00d        mov     ip, sp
119     8480:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
120     8484:       e24cb004        sub     fp, ip, #4          ; 0x4
121     8488:       e24dd010        sub     sp, sp, #16         ; 0x10
122     848c:       e50b0010        str     r0, [fp, -#16]
123     8490:       e50b1014        str     r1, [fp, -#20]
124     8494:       e50b2018        str     r2, [fp, -#24]
125     8498:       e3a03000        mov     r3, #0   ; 0x0
126     849c:       e50b301c        str     r3, [fp, -#28]
127     84a0:       e51b201c        ldr     r2, [fp, -#28]
128     84a4:       e3a03a02        mov     r3, #8192           ; 0x2000
129     84a8:       e2833003        add     r3, r3, #3          ; 0x3
130     84ac:       e1520003        cmp     r2, r3
131     84b0:       da000000        ble     84b8 <loop_ICM1+0x3c>
132     84b4:       ea000013        b       8508 <loop_ICM1+0x8c>
                                                        116,0-1       31%
```

함수명

argument를 stack에 저장하고 for문 시작 부분

Loop Invariant Code Motion

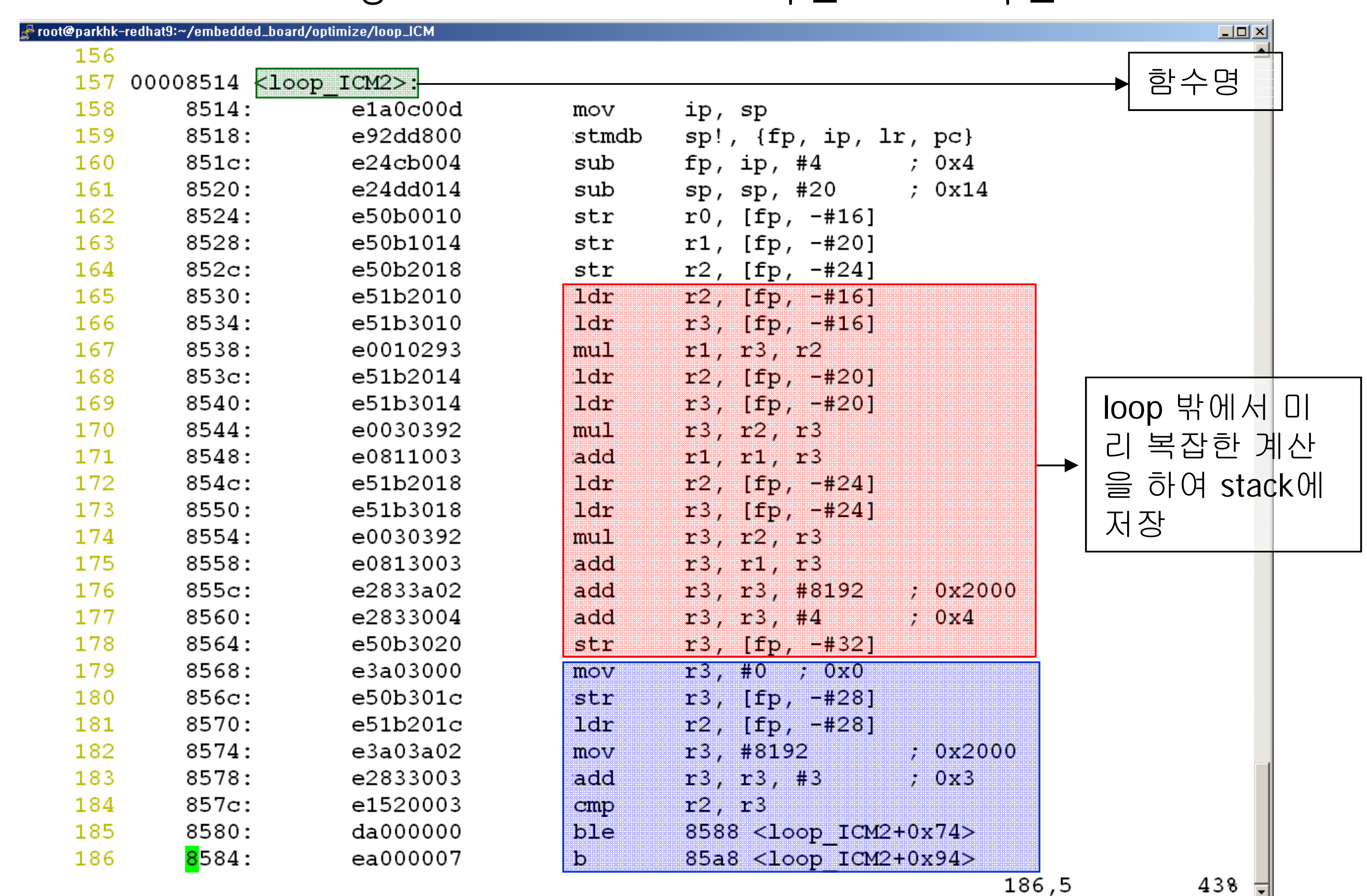
- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o loop_ICM loop_ICM.c



```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
133    84b8:    e59f0050    ldr    r0, [pc, #80]    ; 8510 <loop_ICM1+0x94>
134    84bc:    e51bc01c    ldr    ip, [fp, -#28]
135    84c0:    e51b2010    ldr    r2, [fp, -#16]
136    84c4:    e51b3010    ldr    r3, [fp, -#16]
137    84c8:    e0010293    mul    r1, r3, r2
138    84cc:    e51b2014    ldr    r2, [fp, -#20]
139    84d0:    e51b3014    ldr    r3, [fp, -#20]
140    84d4:    e0030392    mul    r3, r2, r3
141    84d8:    e0811003    add    r1, r1, r3
142    84dc:    e51b2018    ldr    r2, [fp, -#24]
143    84e0:    e51b3018    ldr    r3, [fp, -#24]
144    84e4:    e0030392    mul    r3, r2, r3
145    84e8:    e0813003    add    r3, r1, r3
146    84ec:    e2833a02    add    r3, r3, #8192    ; 0x2000
147    84f0:    e2833004    add    r3, r3, #4       ; 0x4
148    84f4:    e780310c    str    r3, [r0, ip, lsl #2]
149    84f8:    e51b301c    ldr    r3, [fp, -#28]
150    84fc:    e2833001    add    r3, r3, #1       ; 0x1
151    8500:    e50b301c    str    r3, [fp, -#28]
152    8504:    eaffffe5    b      84a0 <loop_ICM1+0x24>
153    8508:    e1a00003    mov    r0, r3
154    850c:    e91ba800    ldmdb  fp, {fp, sp, pc}
155    8510:    00010a00    andeq  r0, r1, r0, lsl #20
156
                                                    133,5         36%
```

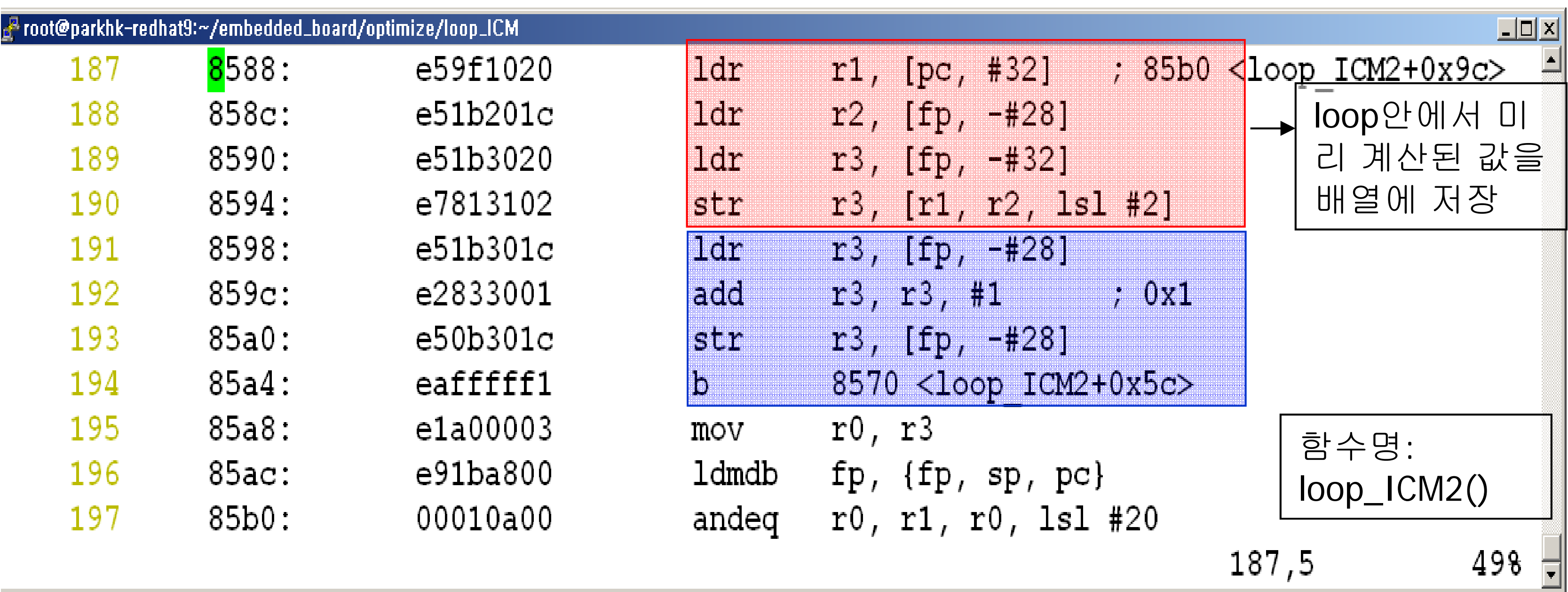
Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o loop_ICM loop_ICM.c



Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o loop_ICM loop_ICM.c



Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ 부분은 for문을 수행하기 위한 arm assembly이고 부분은 수식을 계산하여 배열에 저장하는 부분.

✓ loop_ICM1()
- 수식을 계산하여 배열에 저장하기 위한 명령어 개수
  - ldr : 8회
  - mul : 3회
  - add : 4회
  - str : 1회
- 위의 명령어들이 ARRAY_SIZE 만큼 반복

✓ loop_ICM2()
- 수식을 계산하여 배열에 저장하기 위한 명령어 개수
  - for문 밖에서 계산되어지는 부분
    - ldr : 8
    - mul : 3
    - add : 4
    - str :1
  - for문 안에서 계산되어지는 부분
    - ldr : 3
    - str : 1

Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 876000nsec, 645000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_loop_ICM]$ ./loop_ICM

loop_ICM1()
Interation : 8196
0sec 876000nsec

loop_ICM2()
Interation : 8196
0sec 645000nsec
[root@ez-x5 dir_loop_ICM]$
```

Loop Invariant Code Motion

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 –lrt –o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
000085e4 <loop_ICM1>:                                    → 함수명
    85e4:       e00c0191        mul     ip, r1, r1
    85e8:       e021c090        mla     r1, r0, r0, ip
    85ec:       e59f0028        ldr     r0, [pc, #40]   ; 861c <loop_ICM1+0x38>
    85f0:       e02c1292        mla     ip, r2, r2, r1
    85f4:       e3a02000        mov     r2, #0  ; 0x0
    85f8:       e28c1a02        add     r1, ip, #8192   ; 0x2000
    85fc:       e3a0ca02        mov     ip, #8192       ; 0x2000
    8600:       e2813004        add     r3, r1, #4      ; 0x4
    8604:       e28c1003        add     r1, ip, #3      ; 0x3
    8608:       e7803102        str     r3, [r0, r2, lsl #2]
    860c:       e2822001        add     r2, r2, #1      ; 0x1
    8610:       e1520001        cmp     r2, r1
    8614:       dafffffb        ble     8608 <loop_ICM1+0x24>
    8618:       e12fff1e        bx      lr
    861c:       000108a0        andeq   r0, r1, r0, lsr #17

00008620 <loop_ICM2>:                                    → 함수명
    8620:       e00c0191        mul     ip, r1, r1
    8624:       e021c090        mla     r1, r0, r0, ip
    8628:       e59f0028        ldr     r0, [pc, #40]   ; 8658 <loop_ICM2+0x38>
    862c:       e02c1292        mla     ip, r2, r2, r1
    8630:       e3a02000        mov     r2, #0  ; 0x0
    8634:       e28c1a02        add     r1, ip, #8192   ; 0x2000
    8638:       e3a0ca02        mov     ip, #8192       ; 0x2000
    863c:       e2813004        add     r3, r1, #4      ; 0x4
    8640:       e28c1003        add     r1, ip, #3      ; 0x3
    8644:       e7803102        str     r3, [r0, r2, lsl #2]
    8648:       e2822001        add     r2, r2, #1      ; 0x1
    864c:       e1520001        cmp     r2, r1
    8650:       dafffffb        ble     8644 <loop_ICM2+0x24>
    8654:       e12fff1e        bx      lr
    8658:       000108a0        andeq   r0, r1, r0, lsr #17
                                                200,1         78%
```

Loop Invariant Code Motion

- O3옵션을 이용한 컴파일 결과
  - ✓ 컴파일러에 의한 최적화에 의해 같은 어셈블리어를 생성해냄
  - ✓ 결과적으로 최적화 코드가 명시적인 변수 선언에 의해 약간의 오버헤드가 생김
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 413000nsec, 453000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_loop_ICM]$ ./loop_ICM

loop_ICM1()
Interation : 8196
0sec 413000nsec

loop_ICM2()
Interation : 8196
0sec 453000nsec
[root@ez-x5 dir_loop_ICM]$
```

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

```
/* original source code (function call)*/

int array_v[ARRAY_SIZE];

int loop_ICM3(char *count){

        int i;

        for(i=0; i < atoi(count) ; i++){
                array_v[i] = i;
        }
}
```

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

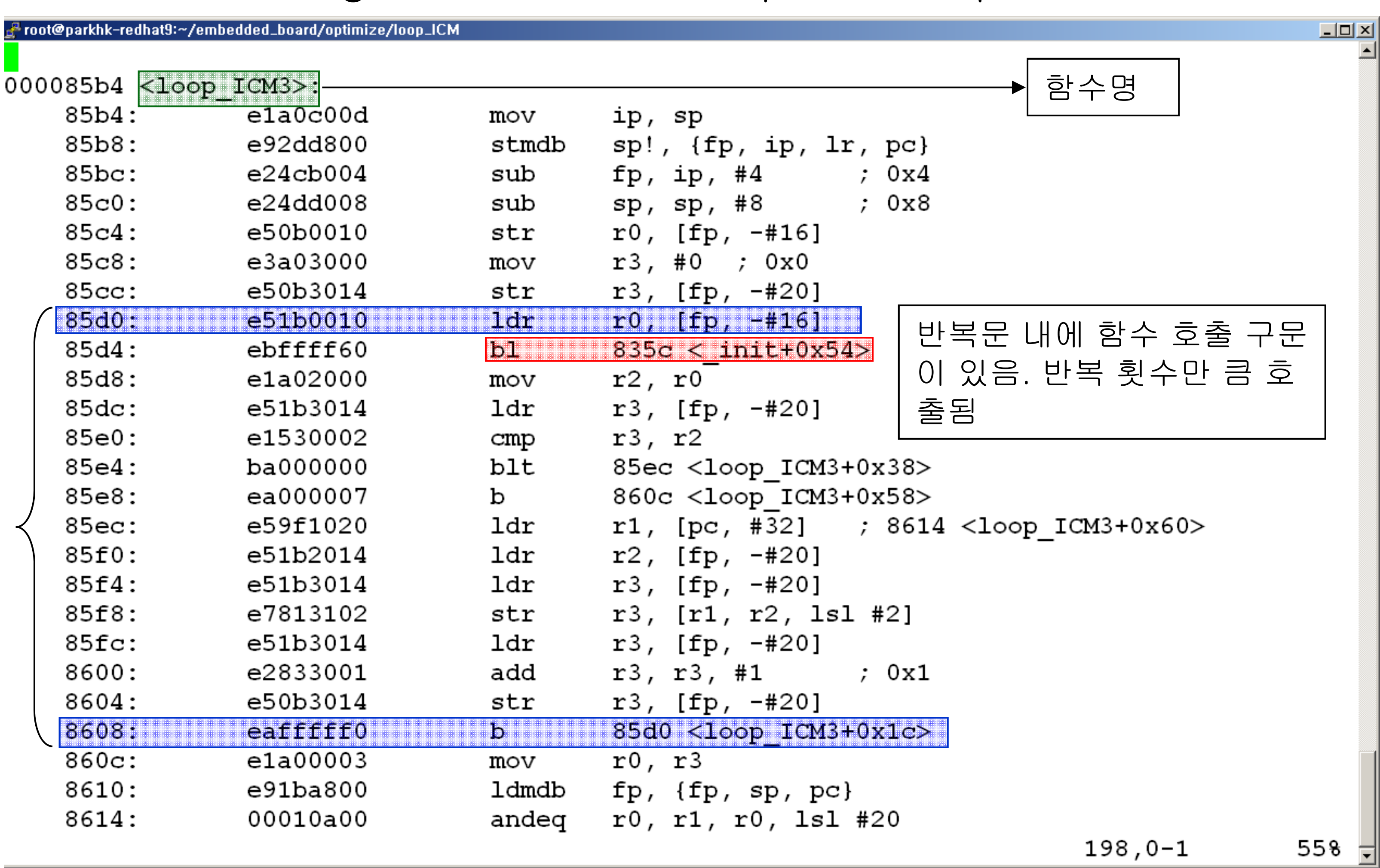
```
/*loop ICM technique (function call)*/

int array_v[ARRAY_SIZE];

int loop_ICM4(char *count){

        int i;
        int cnt;

        cnt = atoi(count);
        for(i=0; i<cnt ; i++){
                array_v[i] = i;
        }
}
```

Loop Invariant Code Motion

- 예제 코드 비교
  - ✓ 매개변수로 전달 받은 문자를 정수형으로 변환하여 해당 숫자만큼 반복문을 수행함.
  - ✓ loop_ICM3
    - 반복문 수행 시 매번 atoi()를 호출
  - ✓ loop_ICM4
    - atoi()호출 결과 값을 상수 값으로 대체하여 loop내에서 호출 횟수를 줄임
  - ✓ 호출 결과 값을 미리 저장해 놓음으로써 함수 호출 횟수를 줄임

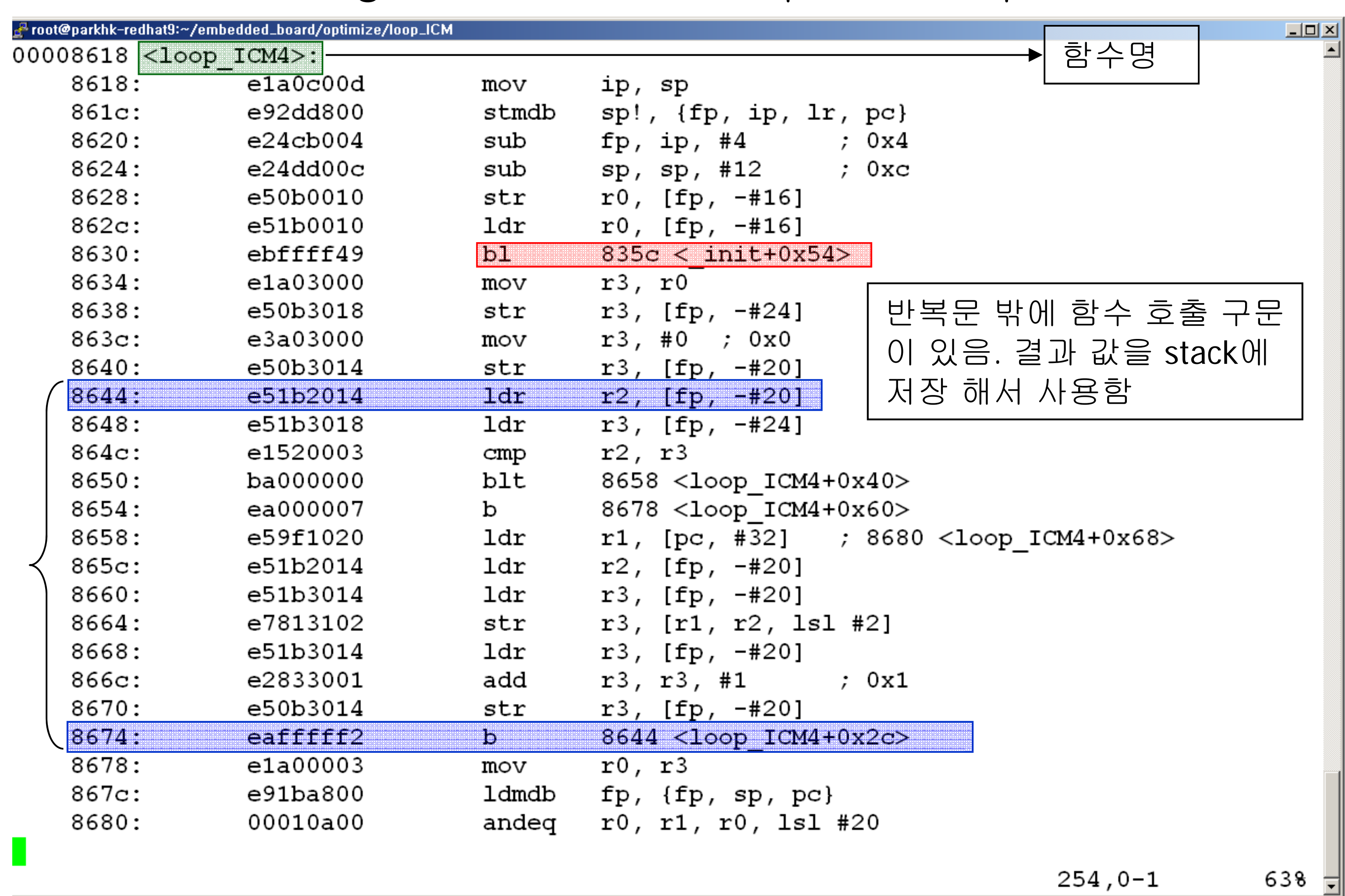
Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o loop_ICM loop_ICM.c



Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 –lrt –o loop_ICM loop_ICM.c



Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ 부분은 for문에서 반복되는 구간이고 부분은 함수를 호출하는 부분이다
  - ✓ loop_ICM3()
    - 반복문에서 루프를 수행할 때 매번 atoi()함수를 호출함.
  - ✓ loop_ICM4()
    - 반복문에서 밖에서 atoi()함수를 호출하고 atoi()의 반환 값을 stack에 저장. 반복문 내에서는 stack에 있는 값을 사용.

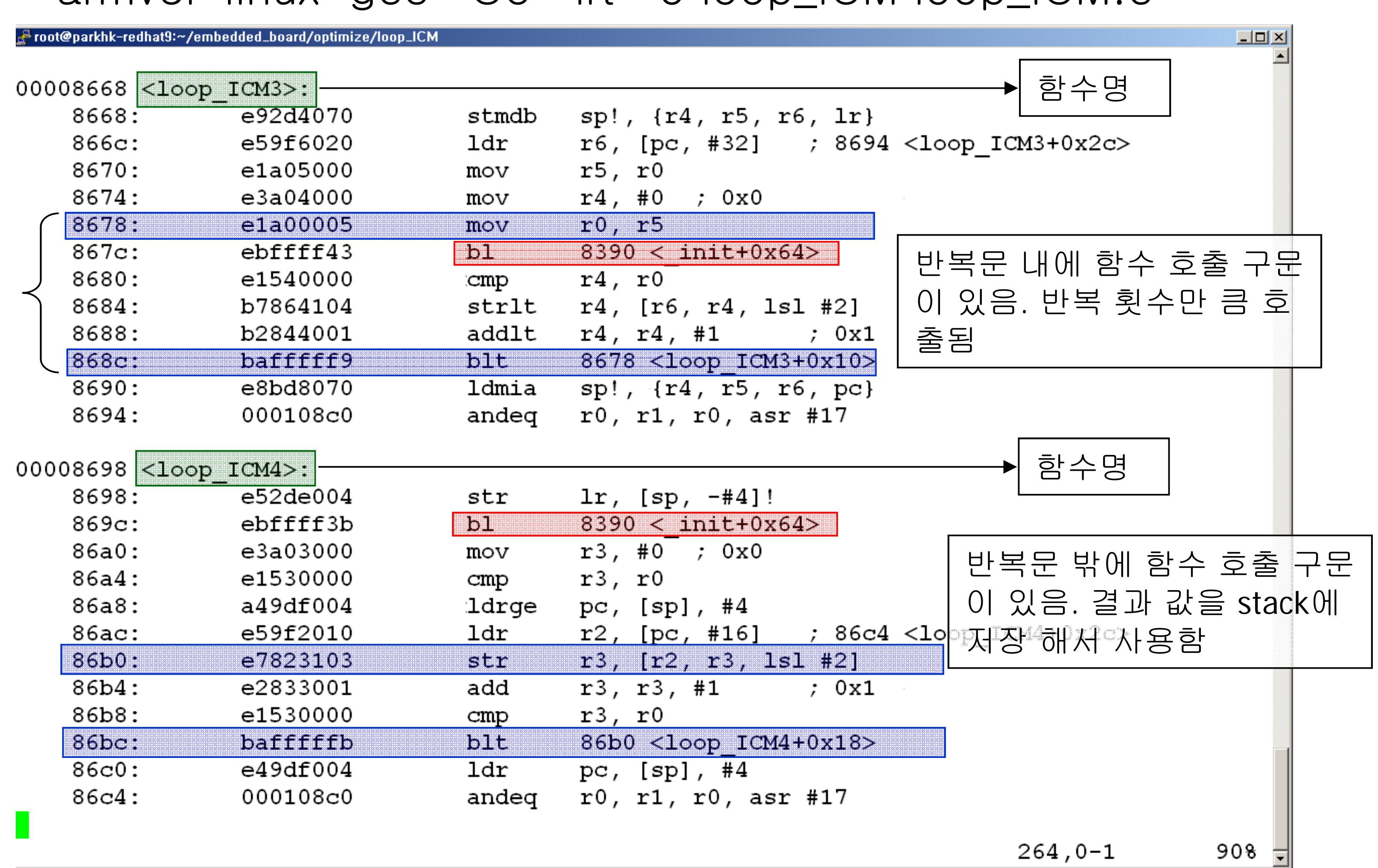
Loop Invariant Code Motion

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 14634000nsec, 658000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_loop_ICM]$ ./loop_ICM

loop_ICM3()
Interation : 8196
0sec 14634000nsec

loop_ICM4()
Interation : 8196
0sec 658000nsec
[root@ez-x5 dir_loop_ICM]$
```

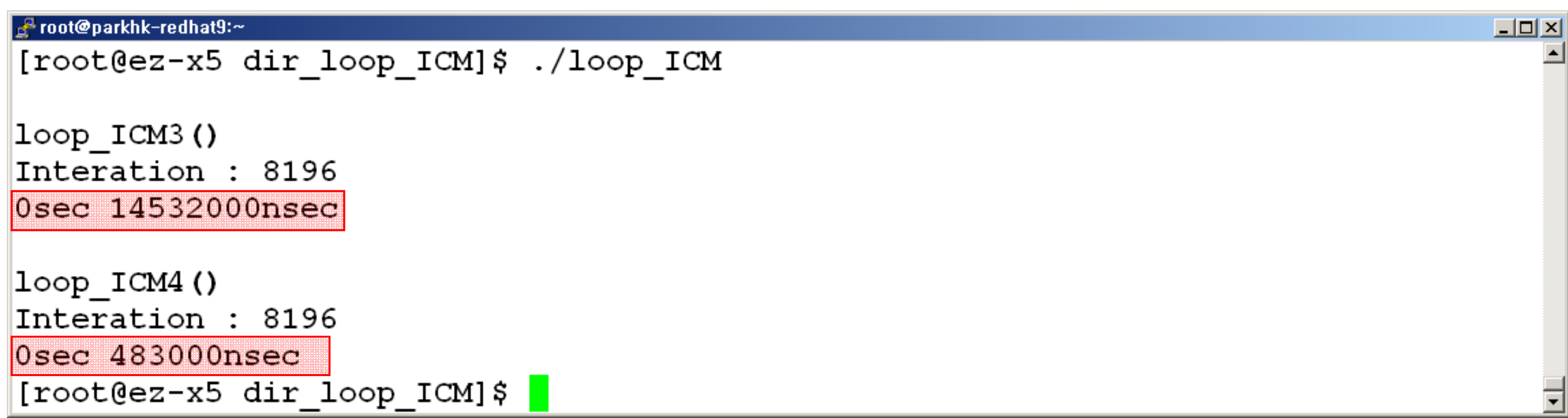
Loop Invariant Code Motion

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 –lrt –o loop_ICM loop_ICM.c



Loop Invariant Code Motion

- O3옵션을 이용한 컴파일 결과
  - ✓ 부분은 for문에서 반복되는 구간이고 부분은 함수를 호출하는 부분이다
  - ✓ 수식을 계산하는 것과는 다르게 최적화 후에도 loop_ICM3에서 함수 호출되는 부분이 for문 안에 있음
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 14532000nsec, 483000nsec의 시간이 걸렸음



Loop Invariant Code Motion

- Loop Invariant Code Motion
  - ✓ 불변하는 코드를 루프 밖으로 이동시켜 결과값을 stack에 저장하여 최적화를 할 수 있음.
    - 수식이 간단하다면?
      - ldr명령어는 메모리에 접근하므로 비용이 많이 듦. loop안에서 계산하는 것이 더 효율적 일 수 있음
        - ex) 계산 값이 define값을 이용하여 간단한 상수로 계산되어 지는 경우
  - ✓ O3옵션을 이용하면 최적화를 해줌
  - ✓ 함수 호출의 경우 O0에 대비하여 O3옵션으로 컴파일 하더라도 이득이 별로 없음
    - 함수를 호출하여 수행하기 때문

Strength Reduction

- Strength Reduction
  - ✓ 고비용의 연산자들 저비용 연산자를 이용하여 연산되도록 코드를 최적화하는 기법
    - Example
      - 나머지 연산을 비트 연산으로 대체
      - sin, cos, pow 등 수학 함수를 호출하는 루틴 중 연산자로 계산시 간단하게 구현 가능한 부분에 대하여 연산자로 대체

- Strength Reduction

예제코드

```
/* original source code (Operator)*/

int array_v[ARRAY_SIZE];

int strength_reduction1(void){

        int i;

        for(i=0; i < ARRAY_SIZE ; i++){
                array_v[i] = i % 8;
        }
}
```

Strength Reduction

- Strength Reduction

예제코드

```
/*Strength reduction technique (Operator) */

int array_v[ARRAY_SIZE];

int strength_reduction2(void){

        int i;

        for(i=0; i < ARRAY_SIZE ; i++){
                array_v[i] = i & 7;
        }
}
```

Strength Reduction

- 예제 코드 비교
  - ✓ strength_reduction1
    - 반복문 수행 시 매번 나머지 연산
    - 나머지 연산은 많은 비용이 듦
  - ✓ strength_reduction2
    - 반복문 수행 시 나머지 연산을 비트 연산으로 대체
    - 나머지 연산에 비하여 적은 비용으로 같은 결과 값을 산출함

Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o strength_reduction strength_reduction.c

root@parkhk-redhat9:~/embedded_board/optimize/strength_reduction_dir

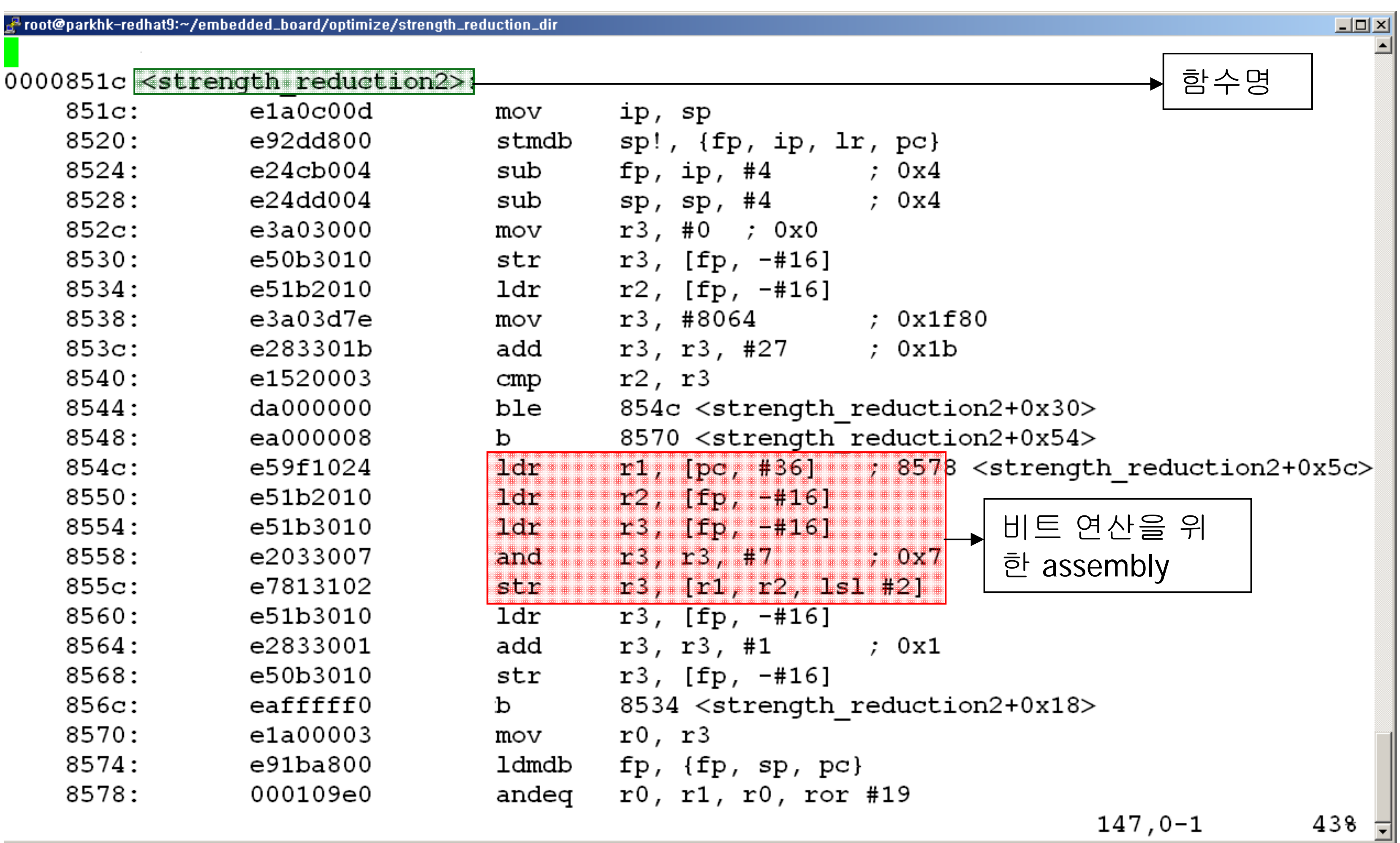
```
000084a8 <strength_reduction1>:
    84a8:       e1a0c00d        mov     ip, sp
    84ac:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    84b0:       e24cb004        sub     fp, ip, #4      ; 0x4
    84b4:       e24dd004        sub     sp, sp, #4      ; 0x4
    84b8:       e3a03000        mov     r3, #0  ; 0x0
    84bc:       e50b3010        str     r3, [fp, -#16]
    84c0:       e51b2010        ldr     r2, [fp, -#16]
    84c4:       e3a03d7e        mov     r3, #8064       ; 0x1f80
    84c8:       e283301b        add     r3, r3, #27     ; 0x1b
    84cc:       e1520003        cmp     r2, r3
    84d0:       da000000        ble     84d8 <strength_reduction1+0x30>
    84d4:       ea00000d        b       8510 <strength_reduction1+0x68>
    84d8:       e59f1038        ldr     r1, [pc, #56]   ; 8518 <strength_reduction1+0x70>
    84dc:       e51b0010        ldr     r0, [fp, -#16]
    84e0:       e51b2010        ldr     r2, [fp, -#16]
    84e4:       e1a03fc2        mov     r3, r2, asr #31
    84e8:       e1a03ea3        mov     r3, r3, lsr #29
    84ec:       e0823003        add     r3, r2, r3
    84f0:       e1a031c3        mov     r3, r3, asr #3
    84f4:       e1a03183        mov     r3, r3, lsl #3
    84f8:       e0633002        rsb     r3, r3, r2
    84fc:       e7813100        str     r3, [r1, r0, lsl #2]
    8500:       e51b3010        ldr     r3, [fp, -#16]
    8504:       e2833001        add     r3, r3, #1      ; 0x1
    8508:       e50b3010        str     r3, [fp, -#16]
    850c:       eaffffeb        b       84c0 <strength_reduction1+0x18>
    8510:       e1a00003        mov     r0, r3
    8514:       e91ba800        ldmdb   fp, {fp, sp, pc}
    8518:       000109e0        andeq   r0, r1, r0, ror #19

                                                  147,0-1        35%
```

함수명

나머지 연산을 위한 assembly

Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 –lrt –o strength_reduction strength_reduction.c



Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ 부분이 나머지 연산을 하기 위한 assembly 코드

- ✓ strength_reduction1()
  - % 연산 비용
    - ldr : 3회
    - mov : 4회
    - add : 1회
    - rsb : 1회
    - str : 1회
- ✓ strength_reduction2()
  - % 연산 비용
    - ldr : 3회
    - and : 1회
    - str : 1회

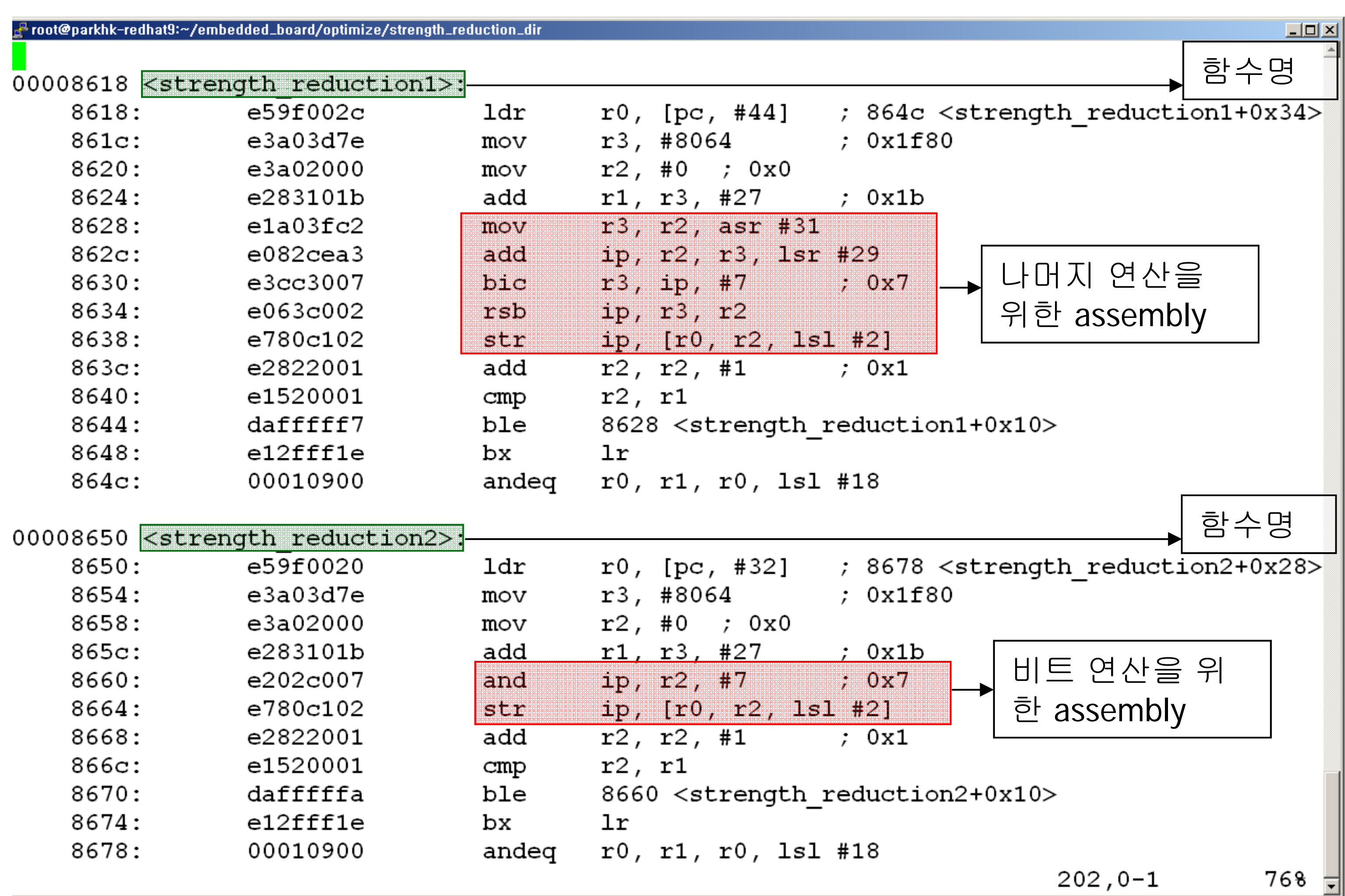
Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 707000nsec, 615000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction

strength_reduction1()
Interation : 8092
0sec 707000nsec

strength_reduction2()
Interation : 8092
0sec 615000nsec
[root@ez-x5 dir_strength_reduction]$
```

Strength Reduction

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o strength_reduction strength_reduction.c



Strength Reduction

- O3옵션을 이용한 컴파일 결과
  - ✓ 부분이 나머지 연산을 하기 위한 assembly 코드
  - ✓ strength_reduction1()
    - % 연산 비용
      - mov : 1회
      - add : 1회
      - rsb : 1회
      - bic : 1회
      - str : 1회
  - ✓ strength_reduction2()
    - % 연산 비용
      - and : 1회
      - str : 1회
  - ✓ 최적화 옵션으로 O3을 사용하였을 경우 고비용의 연산자가 없고 코드가 간결해짐
  - ✓ 실험 결과 거의 성능차가 거의 없음

Strength Reduction

- O3옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 470000nsec, 468000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction

strength_reduction1()
Interation : 8092
0sec 470000nsec

strength_reduction2()
Interation : 8092
0sec 468000nsec
[root@ez-x5 dir_strength_reduction]$
```

Strength Reduction

- Strength Reduction

예제코드

```
/* original source code (Function call)*/

int strength_reduction3(void){

        int i;

        for(i=0; i < ARRAY_SIZE ; i++){
                array_v[i] = pow(i, 2);
        }
}
```

- Strength Reduction

예제코드

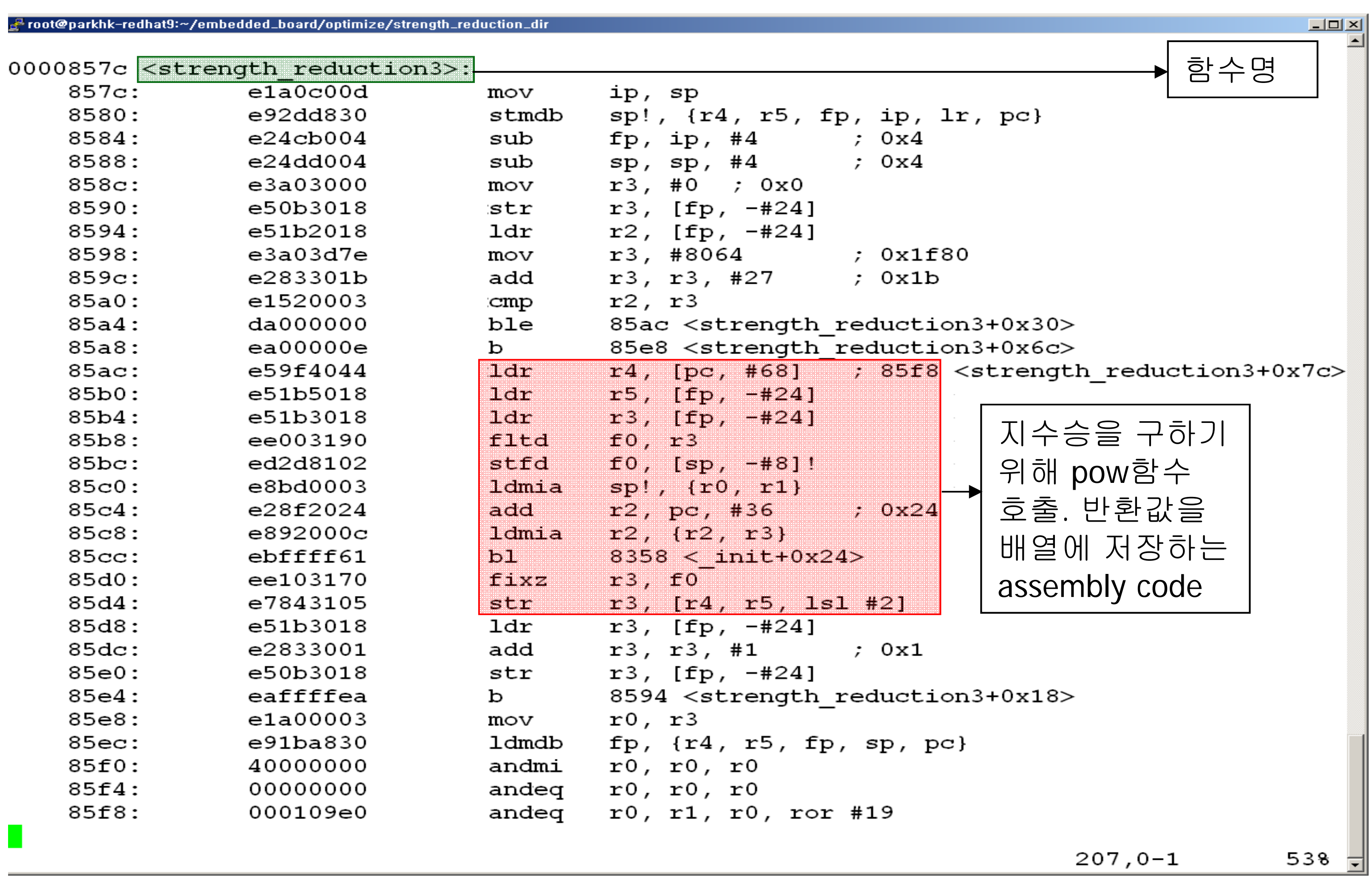
```
/*Strength reduction technique (Function call) */

int strength_reduction4(void){

        int i;

        for(i=0; i < ARRAY_SIZE ; i++){
                array_v[i] = i*i;
        }
}
```

Strength Reduction

- 예제 코드 비교
  - ✓ strength_reduction3
    - 반복문 수행 시 매번 pow()함수 호출하여 제곱 값 산출
    - 수학 함수를 호출하여 값 산출 시 많은 비용이 듦
  - ✓ strength_reduction3
    - 반복문 수행 시 수학 함수 호출을 산술연산으로 대체
    - 수학 함수 호출에 비하여 적은 비용으로 같은 결과 값을 산출함

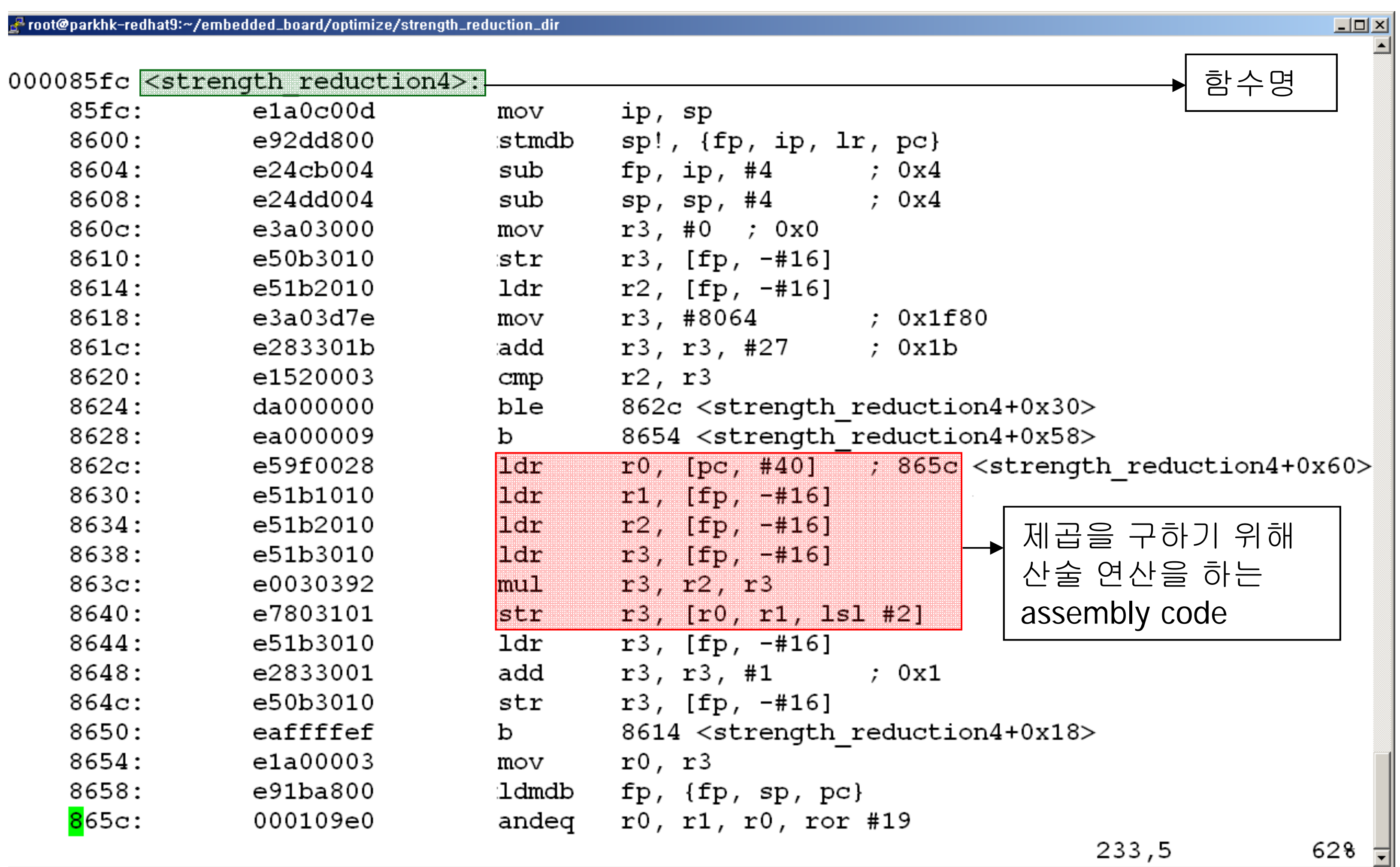
Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o strength_reduction strength_reduction.c



Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 –lrt –o strength_reduction strength_reduction.c



Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ ▨부분이 제곱 연산을 하기 위한 assembly 코드

- ✓ strength_reduction3()
  - 제곱 연산 비용(pow()호출)
    - ldr : 3회
    - fltd : 1회
    - stfd : 1회
    - ldmia : 2회
    - add : 1회
    - fixz : 1회
    - str : 1회
    - bl : 1회
- ✓ strength_reduction4()
  - 제곱 연산 비용(산술연산)
    - ldr : 4회
    - mul : 1회
    - str : 1회

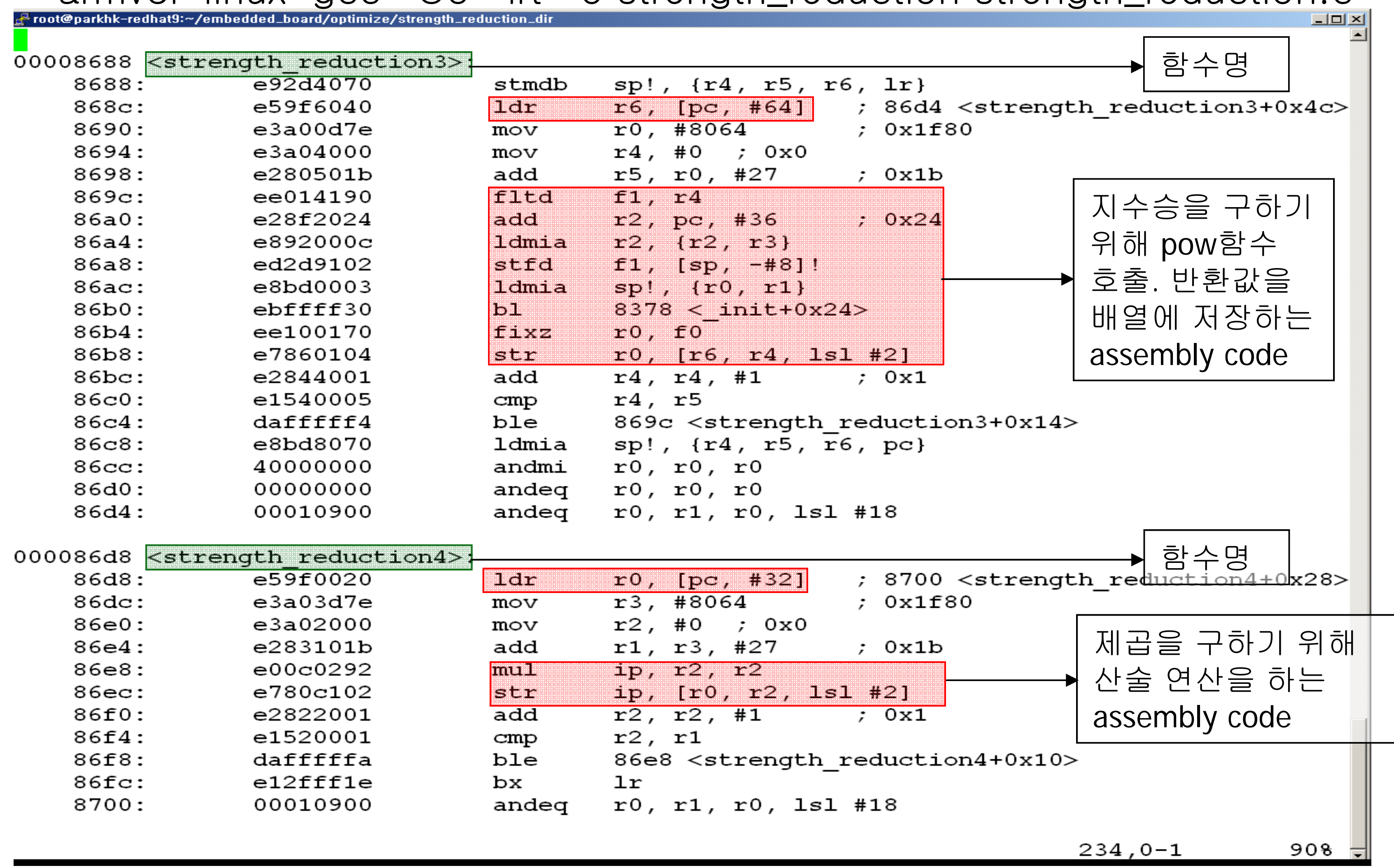
Strength Reduction

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 259531000nsec, 678000nsec의 시간이 걸렸음

```
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction

strength_reduction3()
Interation : 8092
0sec 259531000nsec

strength_reduction4()
Interation : 8092
0sec 678000nsec
[root@ez-x5 dir_strength_reduction]$
```

Strength Reduction

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 –lrt –o strength_reduction strength_reduction.c



Strength Reduction

- O3옵션을 이용한 컴파일 결과
  - ✓ 부분이 제곱 연산을 하기 위한 assembly 코드
  - ✓ strength_reduction3()
    - O0 옵션으로 컴파일 했을 경우와 비교하여 상대적인 성능 향상은 미비함
    - 함수 호출로 인한 오버헤드가 대부분
  - ✓ strength_reduction3()
    - O0 옵션으로 컴파일 했을 경우와 비교하여 상대적으로 70% 정도의 성능 향상이 있음.

Strength Reduction

- O3옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 257165000nsec, 486000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction

strength_reduction3()
Interation : 8092
0sec 257165000nsec

strength_reduction4()
Interation : 8092
0sec 486000nsec
[root@ez-x5 dir_strength_reduction]$
```

Strength Reduction

- Strength Reduction
  - ✓ modular 연산 등 고비용의 산술연산을 비트 연산등의 저비용 연산으로 교체 하는 것이 효과적
  - ✓ 수학 함수를 호출하여 연산을 하는 것은 산술연산에 비하여 고비용이 듦
  - ✓ 간단한 연산의 경우 산술연산을 하는 것이 효과적
  - ✓ 코드의 복잡도, 가독성, 속도 향상 등을 고려하여 최적화

Count up to Zero

- Count up to Zero
  - ✓ 루프의 종료 조건을 0과 비교하는 것이 효율적
  - ✓ 대부분의 아키텍처에서는 0에 도달할 때 ZERO 플래그를 재설정하게 됨
  - ✓ 변수를 0과 비교하게 되면 명시적인 비교문이 생략됨

Count up to Zero

- Count up to Zero

예제코드

```
/* original source code */

int count_up_to_zero1(int *input){

        int i;
        int sum = 0;

        for(i=0; i<ARRAY_SIZE ; ++i){
                sum += *(input++)+i;
        }

        return sum;
}
```

Count up to Zero

- Count up to Zero

예제코드

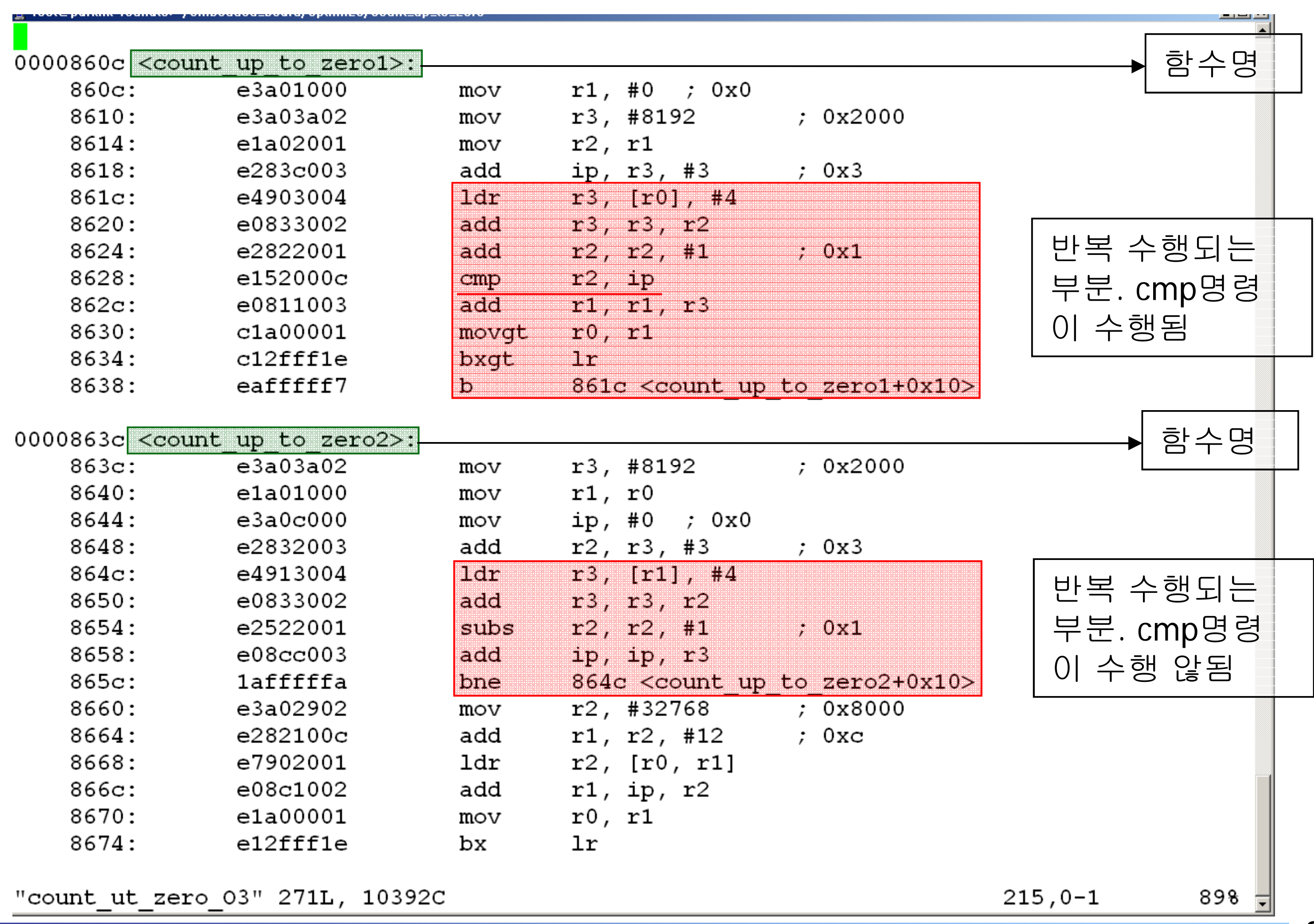
```
/*Count up to Zero*/

int count_up_to_zero2(int *input){

        int i;
        int sum = 0;

        for(i=ARRAY_SIZE-1; i!=0 ; --i){
                sum += *(input++)+i;
        }
        sum += *(input++)+i;

        return sum;
}
```

Count up to Zero

- 예제 코드 비교
  - ✓ Count_up_to_zero1
    - 루프문 종료 조건을 양의 상수와 비교
    - 조건문에 사용되는 변수 i를 loop 내부에서 사용
  - ✓ count_up_to_zero2
    - 루프문 종료 조건을 0과 비교
    - 조건문에 사용되는 변수 i를 loop 내부에서 사용

Count up to Zero

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o count_up_to_zero count_up_to_zero2.c



Count up to Zero

- O3옵션을 이용한 컴파일 결과
  - ✓ 명시적인 조건 비교(cmp)를 하지 않음.
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 513000nsec, 503000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_count_up_to_zero]$ ./count_ut_zero

count_up_to_zero1()
Interation : 8196
0sec 513000nsec
Sum : 67166220

count_up_to_zero2()
Interation : 8196
0sec 503000nsec
Sum : 67166220
[root@ez-x5 dir_count_up_to_zero]$
```

Count up to Zero

- Count up to Zero

예제코드

```
/* original source code */

int count_up_to_zero1(int *input){

        int i;
        int sum = 0;

        for(i=0; i<ARRAY_SIZE ; ++i){
                sum += *(input++);
        }

        return sum;
}
```

Count up to Zero

- Count up to Zero

예제코드

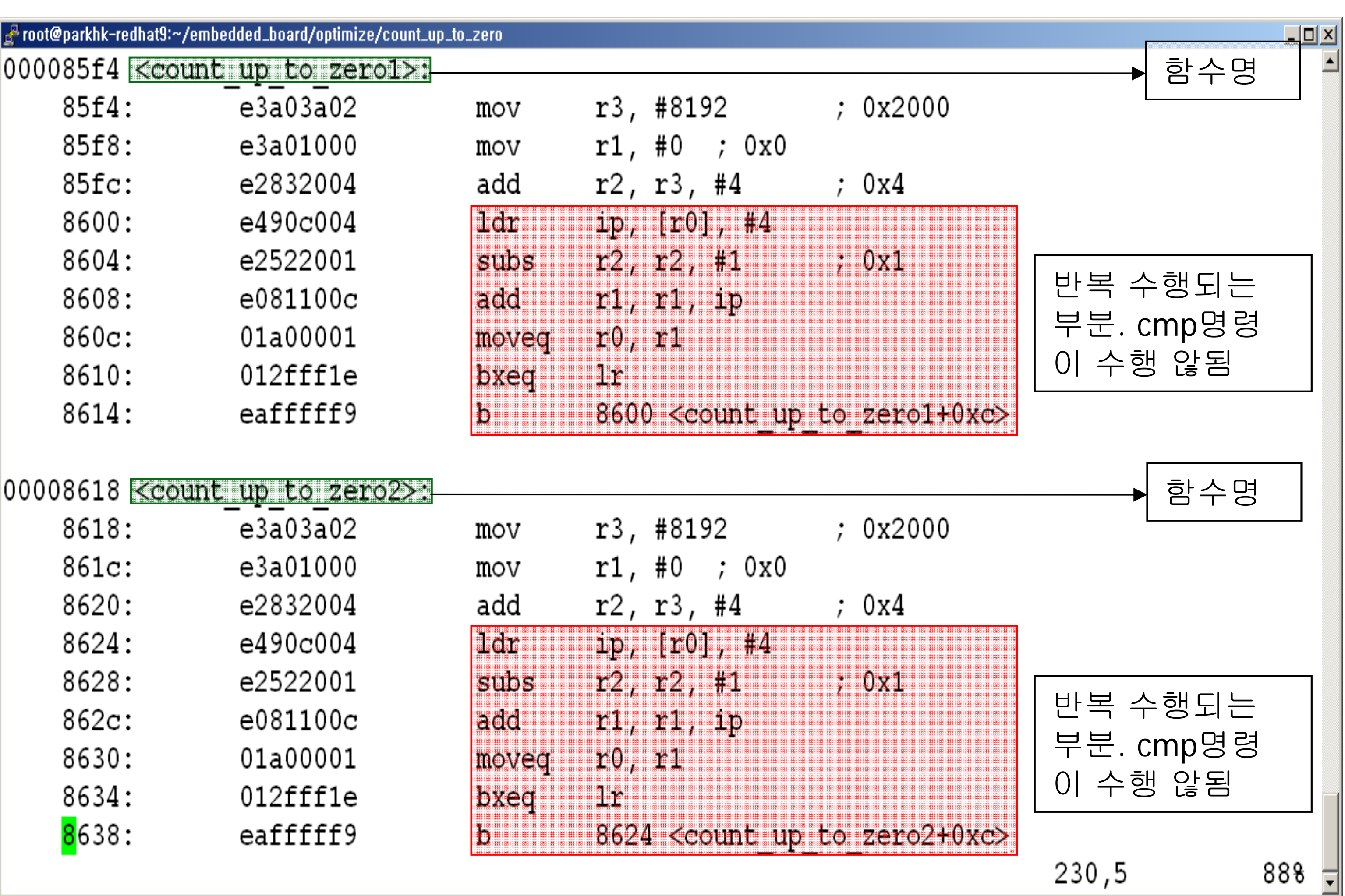
```
/*Count up to Zero*/

int count_up_to_zero2(int *input){

        int i;
        int sum = 0;

        for(i=ARRAY_SIZE; i!=0 ; --i){
                sum += *(input++);
        }

        return sum;
}
```

Count up to Zero

- 예제 코드 비교
  - ✓ Count_up_to_zero1
    - 루프문 종료 조건을 양의 상수와 비교
    - 조건문에 사용되는 변수 i를 loop 내부에서 미사용
  - ✓ count_up_to_zero2
    - 루프문 종료 조건을 0과 비교
    - 조건문에 사용되는 변수 i를 loop 내부에서 미사용

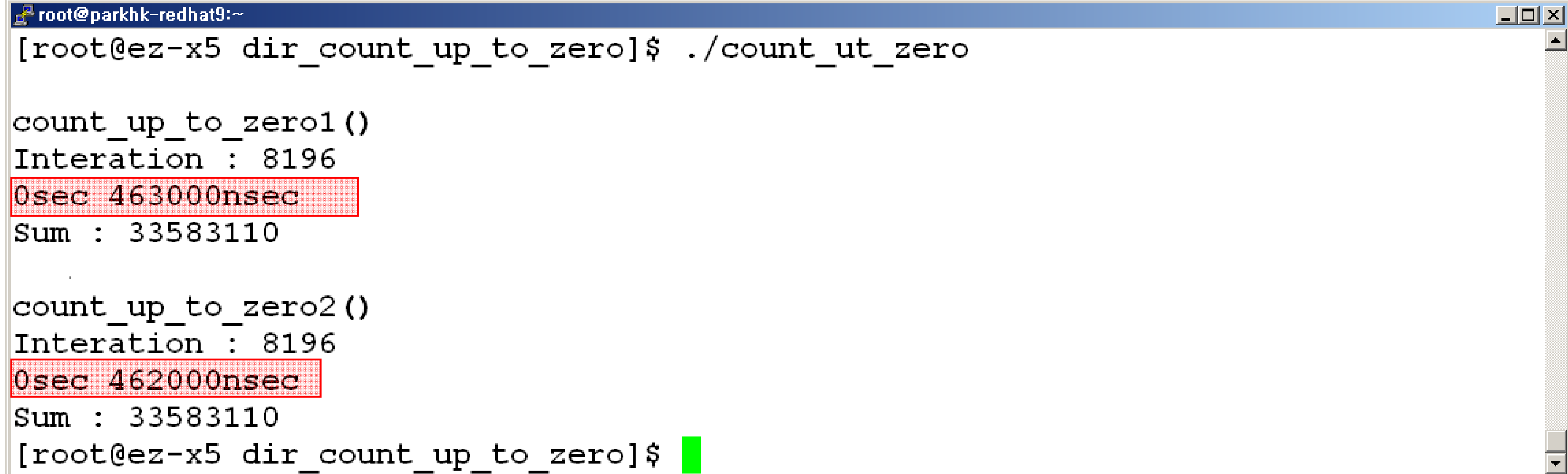
Count up to Zero

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o count_up_to_zero count_up_to_zero2.c



Count up to Zero

- O3옵션을 이용한 컴파일 결과
  - ✓ loop 내에서 조건 비교 변수(i)를 사용하지 않음으로써 최적화 되었음
  - ✓ 결국 두 개의 예제는 똑같은 assembly code를 생성하였음
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 463000nsec, 462000nsec의 시간이 걸렸음



```
[root@ez-x5 dir_count_up_to_zero]$ ./count_ut_zero

count_up_to_zero1()
Interation : 8196
0sec 463000nsec
Sum : 33583110

count_up_to_zero2()
Interation : 8196
0sec 462000nsec
Sum : 33583110
[root@ez-x5 dir_count_up_to_zero]$
```

Count up to Zero

- 조건 비교를 양수가 아닌 0값과 함으로써 명시적 cmp 명령이 생략 될 수 있음.
- 내부적으로 조건 비교 변수를 사용하지 않으면 최적화 옵션에 따라 cmp 명령이 없는 코드로 최적화 될 수 있음

Register Allocation

- Register Allocation
  - ✓ 레지스터 사용(일반적 용도)
    - 컴파일러에서는 지역변수를 r3등에 저장하거나 sp를 ip에 저장하여 레지스터 사용을 최대화함

Reg. 번호	Reg. 이름	Reg. 사용
r0	a1	argument 1
r1	a2	argument 2
r2	a3	argument 3
r3	a4	argument 4
r4	v1	register variable 1
r5	v2	register variable 2
r6	v3	register variable 3
r7	v4	register variable 4

Reg. 번호	Reg. 이름	Reg. 사용
r8	v5	register variable 5
r9	v6/sb	register variable 6 / Static base
r10	v7/sl	register variable 7 /stack limit
r11(fp)	v8/fp	frame pointer
r12(ip)	ip	scratch register
r13(sp)	sp	stack pointer
r14(lr)	lr	link register
r15(pc)	pc	program counter

Register Allocation

- Register Allocation
  - ✓ C 컴파일러가 변수를 위해 사용할 수 있는 레지스터
    - r0~r12, r14
    - 실제로 몇몇 컴파일러에서는 리턴 주소를 저장하는 lr과 임시 작업용 레지스터인 ip는 할당하지 않음.
      - 레지스터 할당을 보장하기 위해서는 최대 12개의 register만을 사용하는 것이 바람직함
  - ✓ 사용가능 register보다 많은 변수를 사용하면 stack에 저장
    - 변수들이 메모리에 저장됨(spilled)
  - ✓ 최적화 방안
    - spilled되는 변수 최소화
    - 자주 접근되는 변수는 레지스터에 저장
      - 컴파일러의 register 할당에 의존하는 것이 바람직함

- Register Allocation

예제코드

```
/* Register Allocation*/

int global_sum, global_a, global_s, global_m, global_d ;

int reg_Alloc1(int v1, int v2){

        global_a = v1 + v2;     global_s = v1 - v2;
        global_m = v1 * v2;     global_d = v1 / v2;

        //중간 생략 : 전역 변수를 이용하여 복잡한 연산 수행

        global_sum = global_a + global_s + global_m + global_d;
        global_sum &= 0xff00;
        if(global_sum > 0) return 0;
        return -1;
}
```

Register Allocation

- Register Allocation

예제코드

```
/* Register Allocation*/

int global_sum, global_a, global_s, global_m, global_d ;

int reg_Alloc2(int v1, int v2){

        register int local_a;       register int local_s;       register int local_d;
        register int local_m;       register int local_sum = 0;

       //중간 생략 : 지역 변수를 이용하여 복잡한 연산 수행
       //                (reg_Alloc1과 같은 연산 수행)

        global_sum = local_sum;       global_a = local_a;       global_s = local_s;
        global_m = local_m;       global_d = local_d;

        if(local_sum > 0) return 0;

        return -1;
}
```

Register Allocation

- 예제 코드 비교
  - ✓ reg_Alloc1()
    - 전역변수를 사용하여 복잡한 연산을 수행함
  - ✓ reg_Alloc2()
    - reg_Alloc1()과 수행 내용은 같음.
    - 레지스터 변수를 선언하고 전역 변수의 값을 저장함
    - 레지스터 변수를 사용하여 복잡한 연산을 수행
    - 레지스터 변수의 값을 전역 변수에 저장

Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o reg_alloc1 reg_alloc1.c
  - ✓ armv5l-linux-gcc –O0 -lrt -o reg_alloc2 reg_alloc2.c
  - ✓ 예제 코드에서는 많은 연산을 수행 함으로 일부 어셈블리만 발췌하여 작성하였음.
    - reg_alloc1 의 명령어 라인 수
      - 266 line
    - reg_alloc2 의 명령어 라인 수
      - 233 line

Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 –lrt –o reg_alloc1 reg_alloc1.c

root@parkhk-redhat9:~/embedded_board/optimize/register_allocation/reg_alloc_global

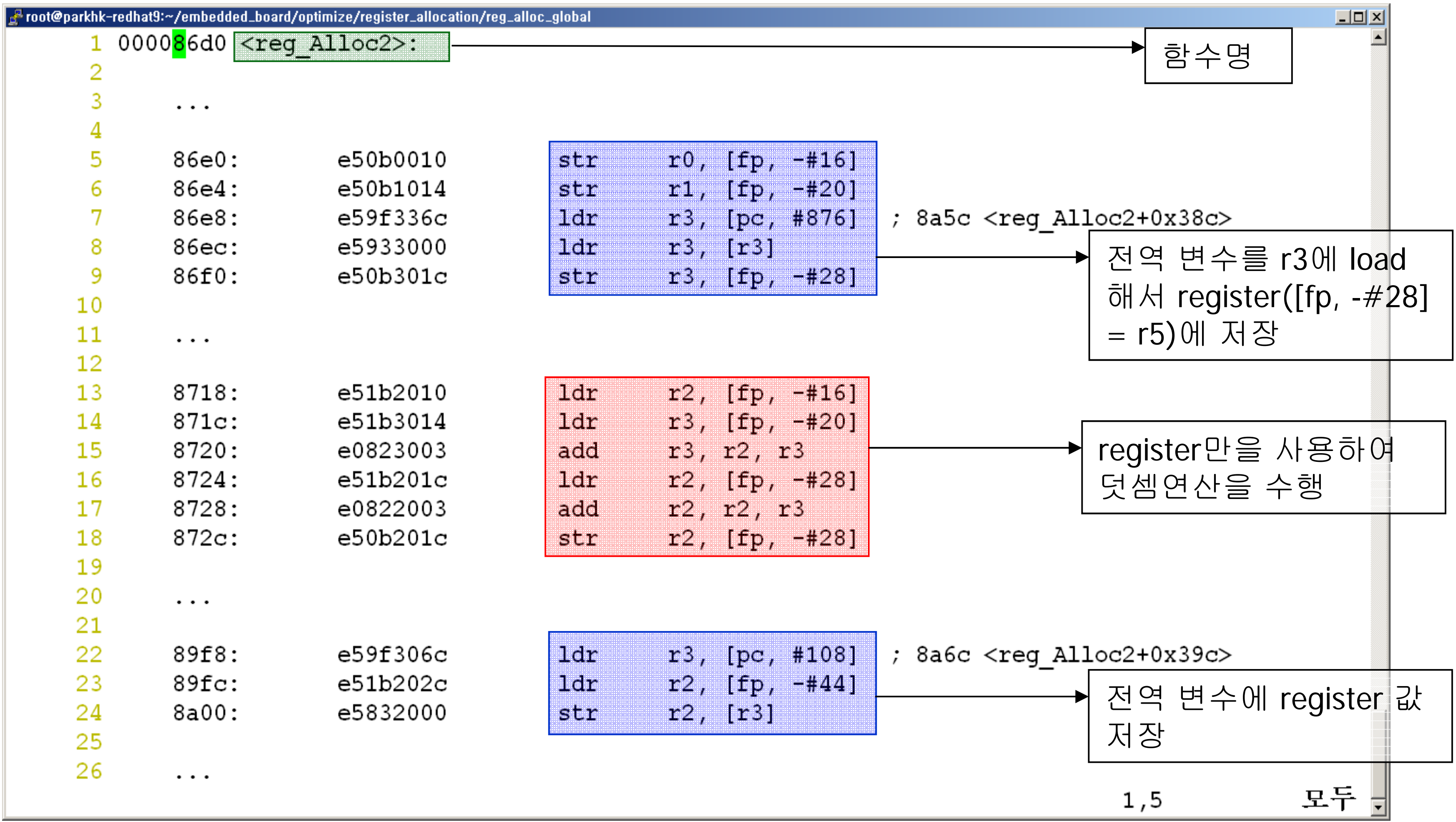
```
 1 000086d0 <reg_Alloc1>:
 2
 3     ...
 4
 5     86e8:    e59f03f0    ldr    r0, [pc, #1008]  ; 8ae0 <reg_Alloc1+0x410>
 6     86ec:    e59f13ec    ldr    r1, [pc, #1004]  ; 8ae0 <reg_Alloc1+0x410>
 7     86f0:    e51b2018    ldr    r2, [fp, -#24]
 8     86f4:    e51b301c    ldr    r3, [fp, -#28]
 9     86f8:    e0822003    add    r2, r2, r3
10     86fc:    e5913000    ldr    r3, [r1]
11     8700:    e0833002    add    r3, r3, r2
12     8704:    e5803000    str    r3, [r0]
13
14     ...
                                              1,1          꼭대기
```

함수명

전역 변수를 사용하여 덧셈 연산 수행 (모든 연산은 메모리 접근을 수반함)

Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o reg_alloc2 reg_alloc2.c



Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ 부분이 제곱 연산을 하기 위한 assembly 코드
  - ✓ 부분은 전역 변수를 지역 변수로 대체하기 위해 데이터를 전달하는 코드
    - 부가적인 작업이지만 전체 수행 코드를 보면 이득임.
    - 전역 변수를 사용한 연산이 적을 때는 손해가 될 수도 있음
  - ✓ reg_Alloc1()
    - 전역 변수를 사용함으로써 연산 시 메모리 접근은 수반함
  - ✓ reg_Alloc2()
    - 전역변수를 지역 변수로 대체함으로써 전역 변수의 값을 읽어 오거나 연산 후 전역 변수에 값을 저장하는 부가 작업 수행
    - register만을 사용한 연산을 수행함으로써 부가작업으로 인한 오버헤드보다 더 많은 이득을 얻음.

Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 52126000nsec, 39330000nsec의 시간이 걸렸음

```
[root@ez-x5 dir_reg_alloc]$ ./reg_alloc1 ; ./reg_alloc2

reg_Alloc1()
global_a : 72 global_s : 88 global_m : 8 global_d : 8
Interation : 8196 Return Value : -8194 Sum of Random Num. 0
0sec 52126000nsec

reg_Alloc2()
global_a : 72 global_s : 88 global_m : 8 global_d : 8
Interation : 8196 Return Value : -8194 Sum of Random Num. 0
0sec 39330000nsec
[root@ez-x5 dir_reg_alloc]$
```

Register Allocation

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o reg_alloc1 reg_alloc1.c
  - ✓ armv5l-linux-gcc –O3 -lrt -o reg_alloc2 reg_alloc2.c
  - ✓ 예제 코드에서는 많은 연산을 수행 함으로 일부 어셈블리만 발췌하여 작성하였음.
    - reg_alloc1 의 명령어 라인 수
      - 152 line
    - reg_alloc2 의 명령어 라인 수
      - 89line

Register Allocation

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o reg_alloc1 reg_alloc1.c

root@parkhk-redhat9:~/embedded_board/optimize/register_allocation/reg_alloc_global

```
 1 0000862c <reg_Alloc1>:
 2
 3      ...
 4
 5      8630:       e59fe264        ldr     lr, [pc, #612]   ; 889c <reg_Alloc1+0x270>
 6      8634:       e59f9264        ldr     r9, [pc, #612]   ; 88a0 <reg_Alloc1+0x274>
 7      8638:       e59fb264        ldr     fp, [pc, #612]   ; 88a4 <reg_Alloc1+0x278>
 8      863c:       e59e4000        ldr     r4, [lr]
 9      8640:       e5995000        ldr     r5, [r9]
10      8644:       e59bc000        ldr     ip, [fp]
11      8648:       e0264091        mla     r6, r1, r0, r4
12      864c:       e0803001        add     r3, r0, r1
13      8650:       e0612000        rsb     r2, r1, r0
14      8654:       e0854003        add     r4, r5, r3
15      8658:       e08c5002        add     r5, ip, r2
16      865c:       e58e6000        str     r6, [lr]
17      8660:       e58b5000        str     r5, [fp]
18      8664:       e1a08000        mov     r8, r0
19      8668:       e1a0a001        mov     sl, r1
20      866c:       e5894000        str     r4, [r9]
21
22      ...
23
                                                        1,1           모두
```

함수명

전역 변수를 사용하여 곱셈 덧셈 뺄셈 연산 수행 (모든 연산은 메모리 접근을 수반함) 최적화를 위해 연산 수행의 순서를 바꿨음.

Register Allocation

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o reg_alloc2 reg_alloc2.c

root@parkhk-redhat9:~/embedded_board/optimize/register_allocation/reg_alloc_global

```
1  0000862c <reg_Alloc2>:
2
3      ...
4
5      8630:    e59f8140    ldr   r8, [pc, #320]  ; 8778 <reg_Alloc2+0x14c>
6      8634:    e59f5140    ldr   r5, [pc, #320]  ; 877c <reg_Alloc2+0x150>
7      8638:    e59f4140    ldr   r4, [pc, #320]  ; 8780 <reg_Alloc2+0x154>
8      863c:    e59f2140    ldr   r2, [pc, #320]  ; 8784 <reg_Alloc2+0x158>
9      8640:    e5986000    ldr   r6, [r8]
10     8644:    e594c000    ldr   ip, [r4]
11     8648:    e5957000    ldr   r7, [r5]
12     864c:    e0803001    add   r3, r0, r1
13     8650:    e061e000    rsb   lr, r1, r0
14     8654:    e5924000    ldr   r4, [r2]
15     8658:    e0266091    mla   r6, r1, r0, r6
16     865c:    e0875003    add   r5, r7, r3
17     8660:    e1a08000    mov   r8, r0
18     8664:    e1a0a001    mov   sl, r1
19     8668:    e08c700e    add   r7, ip, lr
20
21     ...
22
23     8738:    e59f203c    ldr   r2, [pc, #60]   ; 877c <reg_Alloc2+0x150>
24     873c:    e59f303c    ldr   r3, [pc, #60]   ; 8780 <reg_Alloc2+0x154>
25     8740:    e0810004    add   r0, r1, r4
26     8744:    e5825000    str   r5, [r2]
27     8748:    e59f2028    ldr   r2, [pc, #40]   ; 8778
28     874c:    e5837000    str   r7, [r3]
29     8750:    e59f302c    ldr   r3, [pc, #44]   ; 8784 <reg_Alloc2+0x158>
30     8754:    e5826000    str   r6, [r2]
31
32     ...
                                       1,1          꼭대기
```

함수명

전역 변수를 register에 저장

register만을 사용하여 덧셈연산을 수행

전역 변수에 register 값 저장

Register Allocation

- O3옵션을 이용한 컴파일 결과
  - ✓ (빨간색) 부분이 제곱 연산을 하기 위한 assembly 코드
  - ✓ (파란색) 부분은 전역 변수를 지역 변수로 대체하기 위해 데이터를 전달하는 코드
    - 부가적인 작업이지만 전체 수행 코드를 보면 이득임.
    - 전역 변수를 사용한 연산이 적을 때는 손해가 될 수도 있음
  - ✓ reg_Alloc1()
    - 컴파일러에 의해서 코드 최적화
      - 명령어 순서가 바뀜: register에 들어있는 값 재사용
  - ✓ reg_Alloc2()
    - 메모리에 있는 전역변수 값을 직접 지역변수에 넣음
      - O0옵션에서는 다른 register(r3)를 사용하였음.

Register Allocation

- O3옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 3045000nsec, 2017000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 dir_reg_alloc]$ ./reg_alloc1 ; ./reg_alloc2

reg_Alloc1()
global_a : 1704 global_s : -57372 global_m : 25015 global_d : 0
Interation : 8196 Return Value : -10 Sum of Random Num. 34816
0sec 3045000nsec

reg_Alloc2()
global_a : 1704 global_s : -57372 global_m : 25015 global_d : 0
Interation : 8196 Return Value : -10 Sum of Random Num. 34816
0sec 2017000nsec
[root@ez-x5 dir_reg_alloc]$
```

Register Allocation

- 서브루틴 호출 시 매개변수 처리
  - ✓ 4-레지스터 규칙
    - 4개 이하의 인자를 매개변수로 받는 함수가 5개 이상의 인자를 매개변수로 받는 함수 보다 효율적으로 처리됨
    - 4개의 매개변수는 각각 레지스터r0~r3로 보내지고 나머지 매개변수는 stack에 쌓이게 됨
  - ✓ 5개 이상의 매개변수를 받아야 하는 경우 구조체를 이용하여 전달하는 것이 효과적임
  - ✓ 컴파일러에 의한 최적화 이유로 구조체를 이용한 매개변수 전달의 최적화가 상쇄될 수 있음

Register Allocation

- Function calls

예제코드

```
/*Function calls*/
main(){
	...
	for(i=0 ; i<MAX_L; i++){
		func_reg1(init_data[0], init_data[1], init_data[2], init_data[3],
				init_data[4],init_data[5],init_data[6] ,cond);
	}
	...
}
int func_reg1(int v1, int v2, int v3, int v4, int v5, int v6, int v7, int cond)
{
	v1 &= v2;	v3 |= v1;	v4 &= v3;
	v5 |= v4;	v6 &= v5;	v7 |= v6;

	if(cond > 0)
		return v7;
	return v6;
}
```

Register Allocation

- Function calls

예제코드

```
/*Function calls*/

main(){
        ...
        for(i=0;i<MAX_L;i++){
                func_reg2(init_data[0], init_data[1], init_data[1], ars);
        }
        ...
}

int func_reg2(int v1, int v2, int v3, arg_reg_struct ars)
{

        v1 &= v2;       v3 |= v1;       ars.v1 &= v3;
        ars.v2 |= ars.v1;       ars.v3 &= ars.v2;       ars.v4 |= ars.v3;

        if(ars.cond > 0)
                return ars.v4;
        return ars.v3;
}
```

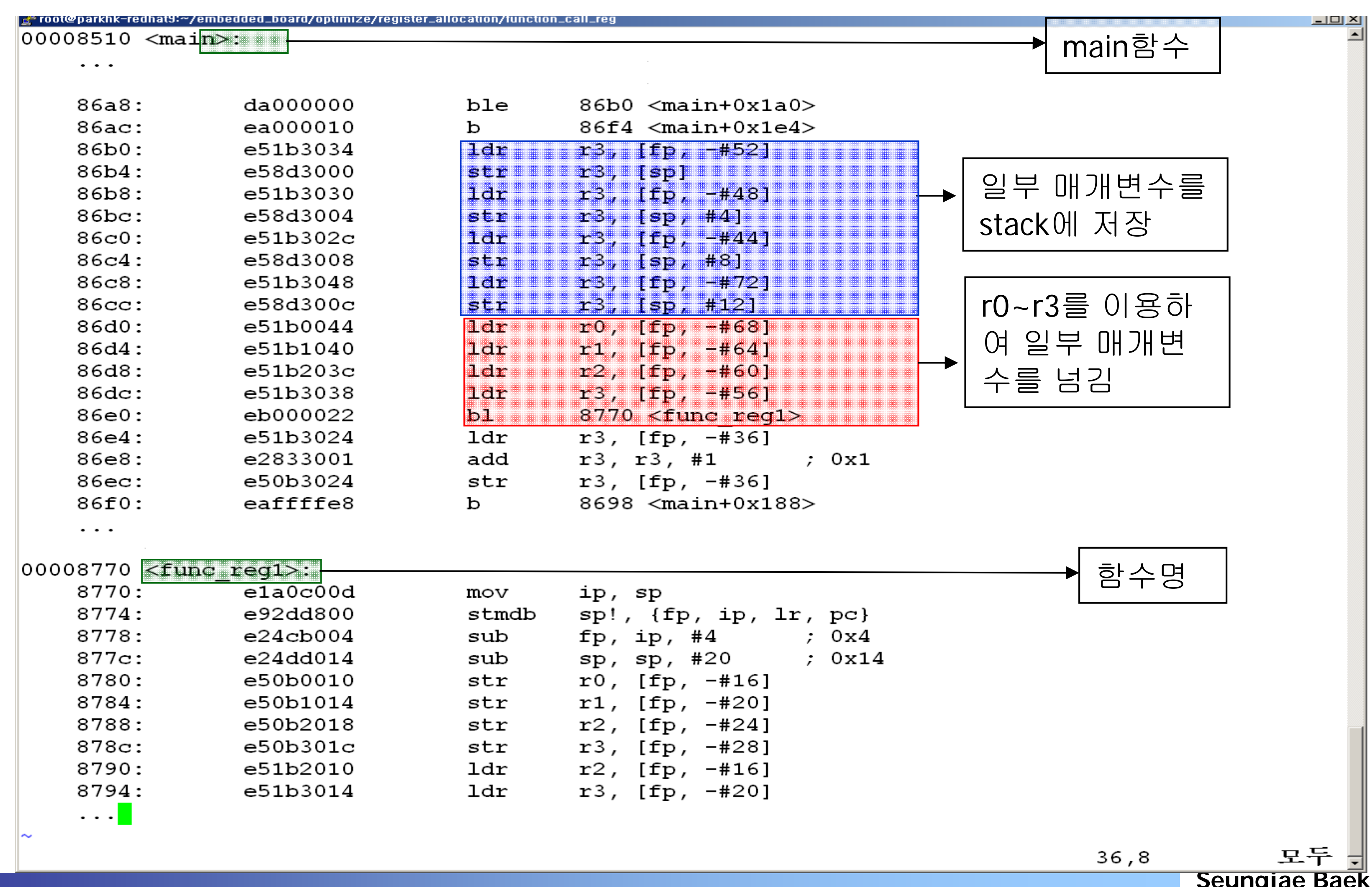
Register Allocation

- 예제 코드 비교
  - ✓ 첫 번째 예제 코드
    - 정수형 변수 8개를 매개변수로 넘김
  - ✓ 두 번째 예제 코드
    - func_reg1()과 수행 내용은 같음.
    - 전달되는 매개변수의 개수를 4개 이하로 유지하기 위해 구조체 사용

```
typedef struct reg_struct{
        int v1;
        int v2;
        int v3;
        int v4;
        int cond;
}arg_reg_struct;
```

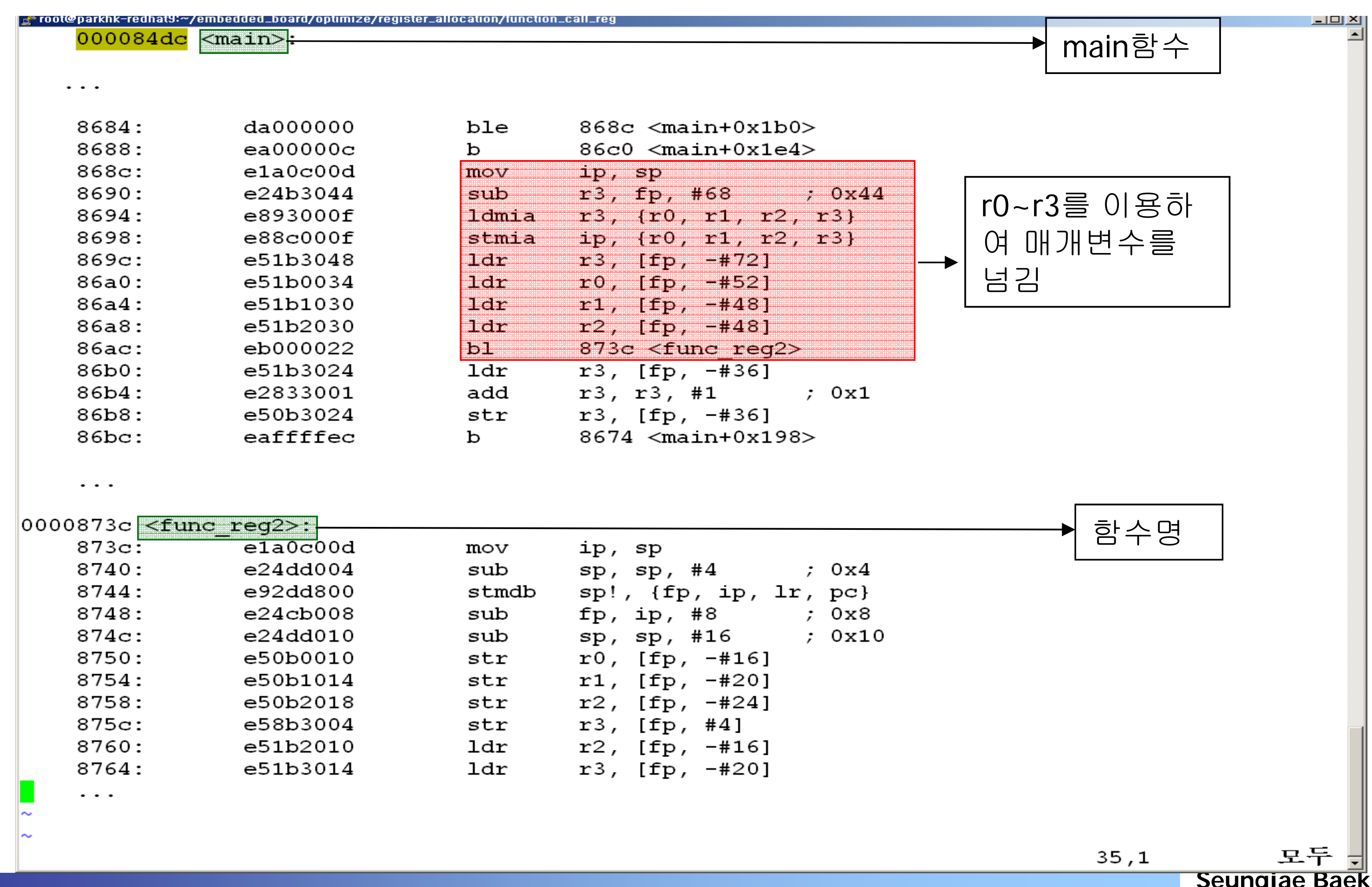
Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o register_allocation register_allocation.c



Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o register_allocation register_allocation.c



Register Allocation

- O0옵션을 이용한 컴파일 결과
  - ✓ ▢부분은 register를 이용한 인자 전달 과정
  - ✓ ▢부분은 stack을 이용한 인자 전달 과정
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 8541000nsec, 8434000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 function]$ ./func_reg1 ; ./func_reg2

func_reg1()
Interation : 32784
0sec 8541000nsec

func_reg2()
Interation : 32784
0sec 8434000nsec
[root@ez-x5 function]$
```

Register Allocation

- Register Allocation
  - ✓ 일반적으로 전역 변수를 사용하여 연산을 수행 하는 것은 부가적인 오버헤드를 유발함.
  - ✓ 전역변수의 사용은 메모리 참조 연산을 수반하기 때문에 전역변수를 사용하여 복잡한 연산을 하는 것은 비효율적임
  - ✓ 지역변수를 이용하여 연산을 수행하고 그 결과값을 전역변수에 저장하는 것이 효과적임
  - ✓ 5개 이상의 매개변수를 받아야 하는 경우 구조체를 이용하여 전달하는 것이 효과적임
  - ✓ 컴파일러에 의한 최적화 이유로 구조체를 이용한 매개변수 전달의 최적화가 상쇄될 수 있음

Software Pipelining

- Software Pipelining
  - ✓ 명령어 마다 각기 다른 사이클 소요
    - 아키텍처마다 다름
    - ARM9TDMI
      - 덧셈, 뺄셈, 논리 연산과 같은 ALU연산은 한 사이클 소요
      - 곱셈과 나눗셈은 오퍼랜드의 값에 따라 다양한 사이클 소요
      - 분기 명령은 3사이클 소요
      - N개를 저장하거나 로드하는 경우 N 사이클 소요
        - 단 1개 일 경우 2사이클 소요
        - PC를 로드한다면 2사이클 추가 소요
  - ✓ 단순화한 ARM9 파이프라인

fetch	decode	ALU	LS1	LS2

- Xscale의 경우 7단계 파이프라인
- ARM11의 경우 8단계 파이프라인

LS1, LS2 : load-store 명령관련 작업

Software Pipelining

- Register Allocation

예제코드

```
/* Register Allocation*/

unsigned long sw_pipeline1(int v1, int v2, int v3, int v4){

	unsigned long i;
	for(i=0; i<ARRAY_SIZE ; i++){
		array_v1[i] = v1 + i;
		array_v11[i] = array_v1[i] & 0xff00;
		array_v2[i] = v2 + i;
		array_v12[i] = array_v2[i] & 0xff00;
		array_v3[i] = v3 + i;
		array_v13[i] = array_v3[i] & 0xff00;
		array_v4[i] = v4 + i;
		array_v14[i] = array_v4[i] & 0xff00;
	}
	return i;
}
```

Software Pipelining

- Register Allocation

예제코드

```
/* Register Allocation*/

unsigned long sw_pipeline2(int v1, int v2, int v3, int v4){

        unsigned long i;
        for(i=0; i<ARRAY_SIZE ; i++){
                array_v1[i] = v1 + i;
                array_v2[i] = v2 + i;
                array_v3[i] = v3 + i;
                array_v4[i] = v4 + i;
                array_v11[i] = array_v1[i] & 0xff00;
                array_v12[i] = array_v2[i] & 0xff00;
                array_v13[i] = array_v3[i] & 0xff00;
                array_v14[i] = array_v4[i] & 0xff00;
        }
        return i;
}
```

Software Pipelining

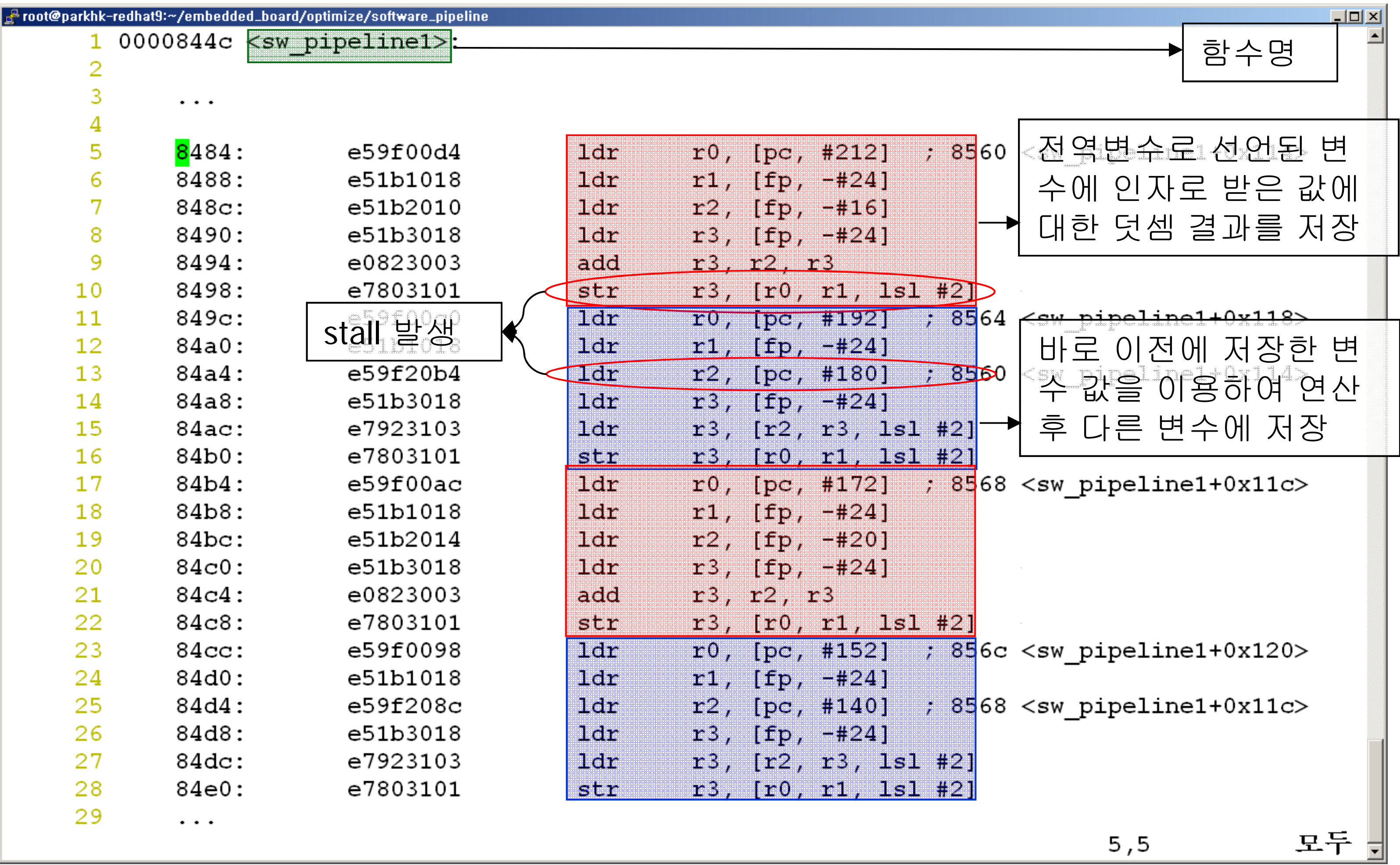
- 예제 코드 비교
  - ✓ sw_pipeline1()
    - 배열에 값을 저장한 후 그 값을 바로 다음에 연산에 사용
  - ✓ sw_pipeline2()
    - sw_pipeline1()과 같은 결과값을 가짐
    - 바로 이전 연산에 종속되지 않도록 작성

Software Pipelining

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 -lrt -o sw_pipeline1 sw_pipeline1.c
  - ✓ armv5l-linux-gcc –O0 -lrt -o sw_pipeline2 sw_pipeline2.c
  - ✓ 예제 코드에서는 많은 연산을 수행 함으로 일부 어셈블리만 발췌하여 작성하였음.
    - sw_pipeline1의 명령어 라인 수
      - 78 line
    - sw_pipeline2의 명령어 라인 수
      - 78 line

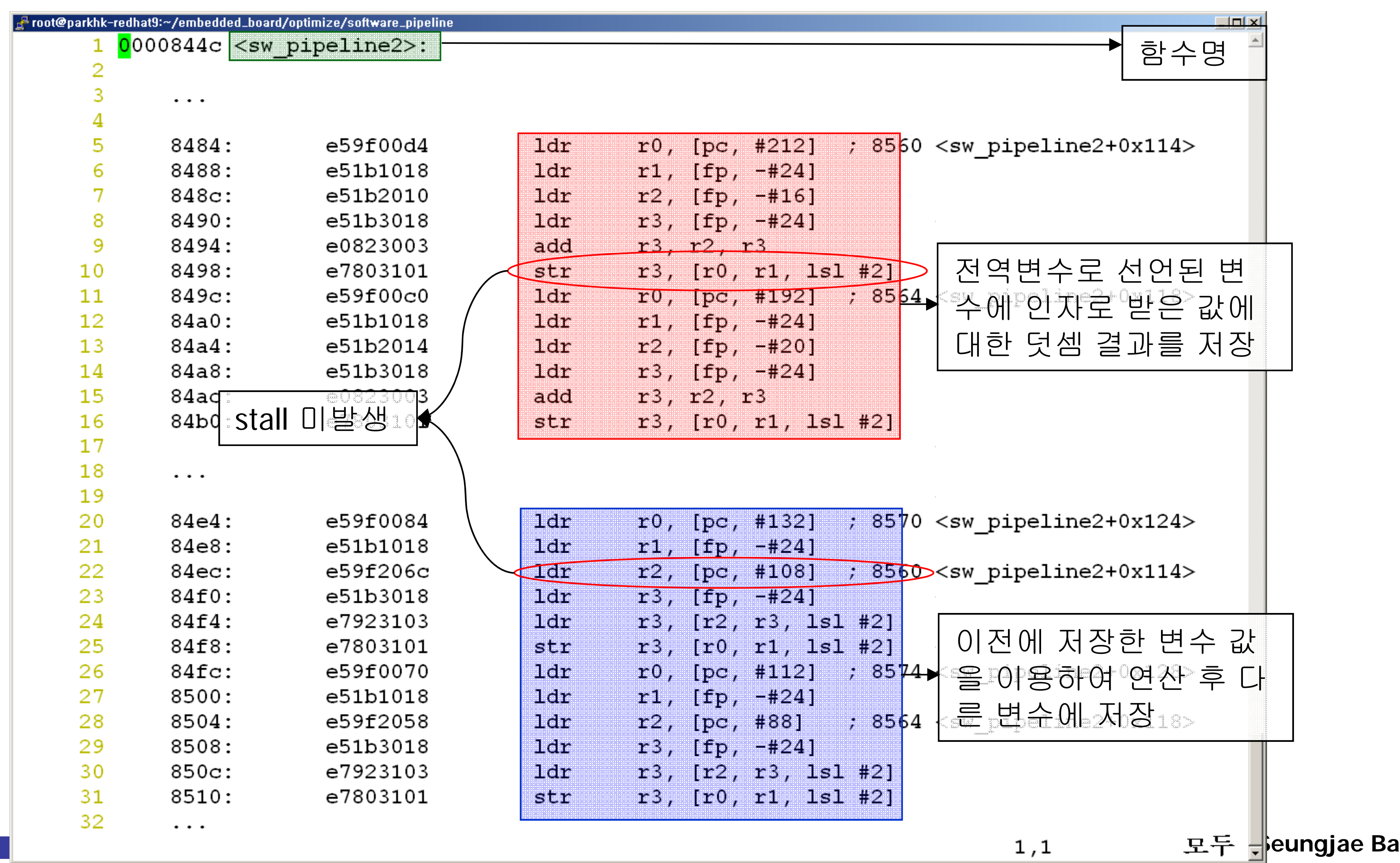
Software Pipelining

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 –lrt –o sw_pipeline1 sw_pipeline1.c



Software Pipelining

- O0옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O0 –lrt –o sw_pipeline2 sw_pipeline2.c



Software Pipelining

- O0옵션을 이용한 컴파일 결과
  - ✓ ▨부분이 인자로 받은 값을 전역 변수(배열)에 저장하는 부분
  - ✓ ▨부분은 저장된 전역변수 값을 연산 후 다른 전역 변수(배열)에 자장
  - ✓ sw_pipeline1()
    - 이전 연산의 결과 값이 종속적인 코드 생성
  - ✓ sw_pipeline2()
    - sw_pipeline1()과 같은 결과를 같지만 이전 연산의 결과 값에 비종속적인 코드 생서

Software Pipelining

- O0옵션을 이용한 컴파일 결과
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 605000nsec, 478000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~
[root@ez-x5 nfs]$ ./sw_pipeline1_O0 ; ./sw_pipeline2_O0

sw_pipeline1()
Interation : 1024  1024
0sec 605000nsec

sw_pipeline2()
Interation : 1024  1024
0sec 478000nsec
[root@ez-x5 nfs]$
```

Software Pipelining

- O3옵션을 이용한 컴파일 결과
  - ✓ armv5l-linux-gcc –O3 -lrt -o sw_pipeline1 sw_pipeline1.c
  - ✓ armv5l-linux-gcc –O3 -lrt -o sw_pipeline2 sw_pipeline2.c
  - ✓ 최적화에 의해 같은 코드 생성
  - ✓ 수행 결과(ez-x5)
    - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 342000nsec, 343000nsec의 시간이 걸렸음(거의 같음)

```
root@parkhk-redhat9:~
[root@ez-x5 nfs]$ ./sw_pipeline1_O3 ; ./sw_pipeline2_O3

sw_pipeline1()
Interation : 1024  1024
0sec 342000nsec

sw_pipeline2()
Interation : 1024  1024
0sec 343000nsec
[root@ez-x5 nfs]$
```

Software Pipelining

- Software Pipelining
  - ✓ 바로 이전 결과값에 종속적인 코드는 data hazard stall을 유발할 수 있음
  - ✓ 최적화 옵션에 따라 instruction moving에 의한 최적화로 stall 발생이 방지될 수 있음