SMC 5번

Refactor : Runtime code generation

런타임 중 코드 생성

코드 흐름

프로그램 실행 중 loop문을 돌며 template code인 tpl의 워드 주소를 참조하면서
gen 코드를 생성함 ->그 후 gen실행

Data declaration

```
        .data
vec1:   .word 22, 0, 25
vec2:   .word 7, 429, 6
result: .word 0
```

Data declaration section

워드 단위 배열 {22, 0, 25}

워드 단위 배열 {7, 429, 6}

초기값 0인 변수

main

카운터 ->

$4	3
$8	0
$9	[4+gen]
$11	tpl
$12	[0+tpl]

```
        .text           # Code section
main:
    li $4, 3            $4 = 3
    li $8, 0            $8 = 0
    la $9, gen          $9 = gen의 주소
    la $11, tpl         $11 = tpl의 주소
    lw $12, 0($11)      $12 = [0+$11] 즉 tpl의 1번째 명령어를 뜻함
    sw $12, 0($9)       $9 = [0+$9] 즉 tpl의 1번째 명령을 gen의 1번째 명령어로 생성
    addi $9, $9, 4      $9 = gen의 주소 + 4 즉 [4+$9](gen의 2번째 명령어)의 주소
```

생성된 코드
gen: li $2, 0

loop(첫번째 루프 $8=0)

$4	3
$8	0
$9	[4+gen]
$10	22
$11	tpl
$12	[8+tpl]
$13	0

인덱스에 따라 불러온 vec1의 값→

vec의 인덱스로 사용 →

loop:

beq $8, $4, post	$8==$4라면 post로 branch. false
li $13, 4	$13 = 4
mul $13, $13, $8	$13 = $13(4)*$8(0) = 0
lw $10, vec1($13)	$10 = vec1의 0번째 값 (22)
beqz $10, next	$10==zero면 next로 점프. false
lw $12, 4($11)	$12 = [4+$11] 즉 tpl의 2번째 명령어
add $12, $12, $13	$12 = [4+$11]+0
sw $12, 0($9)	$9에 [4+tpl] 생성
lw $12, 8($11)	$12 = [8+$11] 즉 tpl의 3번째 명령어

0어브로 그대로 생성

생성된 코드

gen: li $2, 0

lw $13, 0($4)

여기서 메모리 주소값

0x240c0000 (604766208) ->

0x240c0016 (604766230)

loop(첫번째 루프 $8=0) 이어서

$4	3
$8	0
$9	[20+gen]
$10	22
$11	tpl
$12	[16+tpl]
$13	0

원래 [8+$11]의 코드는
li $12, 0 이지만
li $12, 22로 생성됨

인덱스에 따라 불러온 vec1의 값→

vec의 인덱스로 사용 →

```
add $12, $12, $10   $12 = [8+$11]+22
sw $12, 4($9)       [4+$9] = $12 [4+$9]위치에서 위 명령을 생성
lw $12, 12($11)     $12 = [12+$11] tpl의 4번째 명령
sw $12, 8($9)       [8+$9] = $12 위 명령을 생성
lw $12, 16($11)     $12 = [16+$11] tpl의 5번째 명령
sw $12, 12($9)      [12+$9] = $12 위 명령을 생성
addi $9, $9, 16     $9 = [16+$9], 9는 20+gen의 위치를 가리킴
```

생성된 코드

```
gen:    li $2, 0
        lw $13, 0($4)
        li $12, 22
        mul $12, $12, $13
        add $2, $2, $12
```

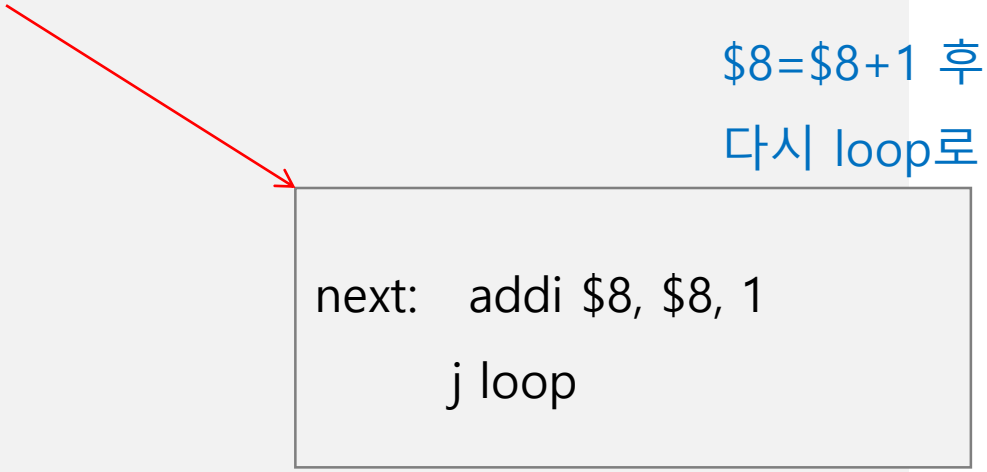
```
next:   addi $8, $8, 1
        j loop
```

loop가 끝나고 $8=$8+1
후 다시 loop로 점프

loop(두번째 루프 $8=1)

$4	3
$8	1
$9	[20+gen]
$10	0
$11	tpl
$12	[16+tpl]
$13	4

```
loop:
        beq $8, $4, post        $8==$4라면 post로 branch. false
        li $13, 4               $13 = 4
        mul $13, $13, $8        $13 = $13(4)*$8(1) = 4
        lw $10, vec1($13)       $10 = vec1의 1번째 값 (0)
        beqz $10, next          $10==zero면 next로 점프. true
        lw $12, 4($11)
        add $12, $12, $13
        sw $12, 0($9)
        lw $12, 8($11)
        add $12, $12, $10
```



$8=$8+1 후

다시 loop로 점프

loop(세번째 루프 $8=2)

$4	3
$8	2
$9	[20+gen]
$10	25
$11	tpl
$12	[8+tpl]
$13	8

인덱스에 따라 불러온 vec1의 값→

vec의 인덱스로 사용 →

```
loop:
    beq $8, $4, post        $8==$4라면 post로 branch. false
    li $13, 4               $13 = 4
    mul $13, $13, $8        $13 = $13(4)*$8(2) = 8
    lw $10, vec1($13)       $10 = vec1의 2번째 인덱스 값 (25)
    beqz $10, next          $10==zero면 next로 점프. false
    lw $12, 4($11)          $12 = [4+$11] tpl의 2번째 명령어
    add $12, $12, $13       $12 = [4+$11]+8
    sw $12, 0($9)           $9에 [4+tpl] 생성
    lw $12, 8($11)          $12 = [8+$11] tpl의 3번째 명령어
```

원래 [4+$tpl]의 코드는
lw $13, 0($4) 이지만
lw $13, 8($4) 로 생성됨

생성된 코드

```
gen:    li $2, 0
        lw $13, 0($4)
        li $12, 22
        mul $12, $12, $13
        add $2, $2, $12
        lw $13, 8($4)
```

loop(세번째 루프 $8=2) 이어서

$4	
$8	
$9	
$10	
$11	
$12	
$13	

인덱스에 따라 불러온 vec1의 값→

vec의 인덱스로 사용 →

```
add $12, $12, $10   $12 = [8+$11]+25
sw $12, 4($9)       [4+$9] = $12 [4+$9]위치에서 위 명령을 생성
lw $12, 12($11)     $12 = [12+$11] tpl의 4번째 명령
sw $12, 8($9)       [8+$9] = $12 위 명령을 생성
lw $12, 16($11)     $12 = [16+$11] tpl의 5번째 명령
sw $12, 12($9)      [12+$9] = $12 위 명령을 생성
addi $9, $9, 16
```

생성된 코드

```
gen:    li $2, 0
        lw $13, 0($4)
        li $12, 22
        mul $12, $12, $13
        add $2, $2, $12
        lw $13, 8($4)
        li $12, 25
        mul $12, $12, $13
        add $2, $2, $12
```

```
next:   addi $8, $8, 1
        j loop
```

loop가 끝나고 $8=$8+1 후 다시 loop로 점프

loop(네번째 루프 $8=3)

```
loop:
        beq $8, $4, post      $8==$4라면 post로 branch. true
        li $13, 4
        mul $13, $13, $8
        lw $10, vec1($13)
        beqz $10, next
        lw $12, 4($11)
        add $12, $12, $13
        sw $12, 0($9)
        lw $12, 8($11)
        add $12, $12, $10
```

post

```
post:   lw $12, 20($11)
        sw $12, 0($9)
        la $4, vec2
        jal gen          #jump and link (다음 명령어의 주소를 $ra(서브루틴시 반
        sw $2, result    환주소 저장하는것)에 저장하고 gen으로 점프?)
        j main           # main으로 돌아가서 반복
```

????

생성된 코드

```
gen:    li $2, 0
        lw $13, 0($4)
        li $12, 22
        mul $12, $12, $13
        add $2, $2, $12
        lw $13, 8($4)
        li $12, 25
        mul $12, $12, $13
        add $2, $2, $12
        jr $31
```

gen

```
vec1:   .word 22, 0, 25        워드 단위 배열 {22, 0, 25}
vec2:   .word 7, 429, 6        워드 단위 배열 {7, 429, 6}
```

#gen은 $2+=22*7, $2+=25*6 ($2는 result)를 실행

#따라서 gen은 vec1과 ve2의 같은 인덱스 원소를 곱해서 각각의 결과를 result에 더하는 것 (배열 원소가 0인 경우 skip)

```
gen:
        li $2, 0            # int gen(int *v)
        lw $13, 0($4)       # {
        li $12, 22          # int res = 0;
        mul $12, $12, $13   # res += 22 * v[0];
        add $2, $2, $12     # res += 25 * v[2];
        lw $13, 8($4)       # return res;
        li $12, 25          # }
        mul $12, $12, $13
        add $2, $2, $12
        jr $31                      #jr (jump reg) : go to $ra
                                    다시 리턴주소(post)로 돌아감
```

- 명령어 add와 addi의 차이
  - add => adds the value in two registers
  - addi => adds an immediate value (constant) to the register