

## <SIC/XE 어셈블러에서의 Quad Tree 알고리즘 구현>

#중간고사 대체 과제

#2021.04.02-04.18

#소프트웨어학과 201921051 정가희

### ■ 알고리즘에 대한 개요

**\*간략한 개요 설명:** 처음에 num을 입력 받는다. 그리고 나서 num\*num만큼의 0 또는 1을 입력 받아 2차원 배열 arr에 차곡차곡 넣는다. 그 후 quad로 가서 num을 chnum에 복사한 후 chnum에 대하여 check 함수를 통하여 배열 chnum\*chnum 크기 속 숫자들이 모두 동일한 수로 이루어져 있는지 확인하고 그렇지 않다면 recur로 가서 chnum <- chnum/2로 줄인 다음에 chnum\*chnum 크기의 4부분으로 나눈다. 이 4부분은 각각 재귀적으로 check 함수를 call 한다. 이와 같이 각 부분에 해당하는 숫자들이 모두 동일하다면 그 수를 출력하고, 동일하지 않다면 4부분으로 나누어 재귀적으로 check 재귀함수를 call하며 수행한다. 그리고 chnum이 기저조건(1)이 되었을 경우엔 그 수를 곧바로 출력한다.

### \*자세한 개요 설명:

```
main    START    0
        LDA      #gap
        JSUB     stinit
        LDA      #gap_n
        JSUB     stinitn
        LDA      #gap_x
        JSUB     stinitx
```

가장 먼저 JSUB을 위한 L 레지스터 값을 보관할 stack과 재귀를 수행하면서 변하는 입력 값에 대한 stack, 그리고 재귀를 수행하면서 indexed addressing 계산을 위해 기준이 될 X레지스터 값에 대한 stack을 생성한다.

```
loop1   LDA      #0
        TD       INDEV
        JEQ      loop1
        RD       INDEV
        SUB      #48
        STA      num
        MUL      num
        STA      n
        RD       INDEV

rdarr   LDS       Esize
        LDX      #0
        LDA      n
        MUL      Esize
        RMO      A,T
```

**loop1:**  $1 \leq n \leq 8$ 의 범위를 가지는 2의 제곱수 n(필자가 짠 코드에서는 num이라는 변수)을 입력 받는다.

**rdarr:** 아까 입력 받은 n을 이용하여 총 입력 받을 배열의 크기를 구하여 T에 저장한다.

```

loop2 LDA #0
      TD INDEV
      JEQ loop2
      RD INDEV
      COMP #10
      JEQ loop2
      SUB #48
      STA arr, X
      ADDR S, X
      COMP R X, T
      JLT loop2
      JSUB quad
      J ENDD

```

**Loop2:** indexed addressing을 이용하여 X에 3씩 증가시키며 2차원 배열 arr에 입력 받은 것을 넣는다. X가 총 입력 받을 배열 크기 T와 같아질 때까지 계속 입력을 받아 배열 arr에 집어넣는다.

다 입력 받았으면, quad로 JSUB한다.

```

quad LDX #0
     LDA num
     STA chnum

check STL tmpL
     LDA tmpL
     JSUB push
     LDA chnum
     COMP #1
     JEQ PRINT
     LDA arr, X
     COMP #1
     JEQ check_1

```

**quad, check:** 재귀를 수행하면서 입력 받은 num이 변하게 될 수도 있으므로 num을 chnum에 복사한다. L 레지스터의 값을 stack에 push한다. chnum이 재귀의 기저조건인 1인지 확인을 하고 1이라면 바로 print해주고, 1이 아니면 배열에서 값이 연속되어 있는지 확인하고 싶은 첫번째 값을 가져와서, 첫번째 값이 1이라면 check\_1로 가고, 0이라면 check\_0으로 간다.

```

check_0 LDA #0
        STA cnt
        STA cnt2
        STA j1
        STA j2
in_0    LDA arr, X
        COMP #1
        JEQ recur
        ADDR S, X
        LDA cnt
        ADD #1
        STA cnt
        LDA j1
        ADD #1
        STA j1
        COMP chnum
        JLT in_0
        LDA #0
        STA j1
        LDA num
        SUB chnum
        MUL Esize
        STA nnn
        ADDR A, X
        LDA cnt2
        ADD #1
        STA cnt2
        LDA j2
        ADD #1
        STA j2
        COMP chnum
        JLT in_0

```

**check\_0, in\_0:** 아까 값이 연속되어 있는지 확인하고 싶은 첫번째 값부터 시작해서 chnum 수만큼 X 값을 3씩 증가시켜가면서 첫번째 값인 0이랑 같은 지 다른 지 체크(열을 증가시키며 체크)를 하고, 같지 않으면 recur로 jump한다. Chnum 수만큼 돌면서 값을 체크하고나서 2차배열이므로 chnum 수만큼 행을 바꿔준 후에 열을 또 체크해줘야 한다.

한마디로 in\_0은 이중루프를 구현한 것이다.

열 증가는 X에 +3을 해주어 수행하였고, 행 증가는 X에  $+(num - chnum) * 3$  더해주어 수행하였다.

(num: 가장 초기에 들어온 입력 n, chnum: 재귀의 입력으로 들어온 n값)

이렇게 총  $chnum * chnum$  개수만큼 체크를 하며 in\_0루프를 다 돌면 이 구간은 전부 동일한 값(0)이라고 볼 수 있으므로, 0을 print해준다.

```

JSUB pop
STA tmpL
LDL tmpL
RSUB

```

그 후 L레지스터값을 stack에서 가져와서 돌아간다.

**\*check\_1, in\_1**도 이와 동일한 알고리즘이다.

```

recur  TD      OUTDEV
      JEQ      recur
      LDA      #40
      WD      OUTDEV
      LDA      chnum
      JSUB     push_n
      LDA      chnum
      DIV      #2
      STA      chnum
      COMP     #1
      JEQ      recur1
      LDA      cnt
      MUL      Esize
      SUBR     A,X
      LDA      nnn
      MUL      cnt2
      SUBR     A,X
      STX      xval2
      LDA      xval2
      JSUB     push_x

```

**recur:** 값이 동일하게 연속되는지 check에서 체크하다가 연속되지 않으면, recur로 와서 (를 출력하고, chnum값을 1/2로 다음에, 2차배열의 구간을 4부분으로 나누어 이를 입력으로 하는 **check**를 **recursive**하게 call한다.

이때 재귀를 수행하고 다시 돌아왔을 때를 대비하여 이전 chnum을 stack에 push해두고 재귀를 다 수행하고 왔을 때, 이전 chnum을 pop하여 가져온다. (addressing을 위한 X 레지스터 계산에 필요한 xvalue2도 동일하게 stack에 push)

만약 재귀의 입력으로 들어온 chnum이 1이라면, recur1에서 check 재귀 함수를 call한다.

```

return TD      OUTDEV
      JEQ      return
      LDA      #41
      WD      OUTDEV
      JSUB     pop_n
      STA      chnum
      JSUB     pop_x
      STA      xval2
      JSUB     pop_x
      STA      xval2
      JSUB     pop
      STA      tmpL
      LDL      tmpL
      J        exit

```

recursive call을 수행하고 나면 )를 출력하여 닫아주고, 이전 chnum 값과 이전 x레지스터 계산에 필요한 xvals2 값, 이전 L레지스터값을 pop하여 가져와서 되돌아간다.

## ■ 구현 코드 내에 Indexed addressing, indirect addressing, relative addressing, direct addressing에 대한 활용 사례 설명

### - Indexed addressing

: loop2에서 N\*N 입력을 받을 때, element size씩 X 값을 증가시키면서 배열 arr에 indexed addressing을 하여 입력을 하나씩 넣어주었다. 한마디로 arr주소에서 X 레지스터 안의 값만큼 더한 곳에 접근하는 indexed addressing을 하였다.

또한 check\_0, check\_1에서 배열 arr에서 특정구간에 들어있는 값들이 동일한 값인지 체크하기 위해 arr에 접근할 때 X를 이용하여 arr주소에서 X만큼 더하여 주소를 지정하는 indexed addressing을 하였다. (bit x=1 이므로 indexed addressing임을 확인할 수 있다.)

```
LDA arr, X
COMP #1
JEQ recur
ADDR S, X
```

in_1	LDA	(PC)+1427,X
	COMP	#0
	JEQ	(PC)+160

Bits nixbpe: nix-p-

[indexed addressing]

## - Indirect addressing

: stack에서 push를 하고 pop을 할 때, @stackp(indirect addressing)을 하였다. Indirect addressing이므로 stackp가 가리키고 있는 주소로 가서 또 다른 주소 값을 인출하고, 인출한 주소 값으로 또 이동하여 그 안의 값을 push하거나 pop한다. (n=1, i=0이므로 indirect addressing임을 알 수 있다.)

```
push STA @stackp
LDA stackp
ADD #3
STA stackp
RSUB
```

push	STA	@(PC)+792
------	-----	-----------

Bits nixbpe: n---p-

[indirect addressing]

## - relative addressing

: relative addressing을 사용한 사례 중 하나를 들자면, quad 안의 LDA num과 STA chnum을 들 수 있다. 먼저 LDA num은 pc를 기준으로 676만큼 떨어진 곳에서 값을 가져와 A 레지스터에 load하는 pc relative addressing이다. 그리고 STA chnum는 pc를 기준으로 670만큼 더한 곳에 값을 저장하는 pc relative addressing이다. (bit p=1이므로 pc relative addressing임을 알 수 있다.)

```
quad LDX #0
LDA num
STA chnum
```

quad	LDX	#0
	LDA	(PC)+676
	STA	(PC)+670

Bits nixbpe: ni--p-

[relative addressing]

## - direct addressing

: LDX #0 <- 와 같이 direct addressing을 이용하여 X 레지스터에 값을 0 그대로 direct하게 집어넣었다. 이 어셈블리어는 direct addressing을 사용했으므로 b=p=0임을 확인할 수 있다.

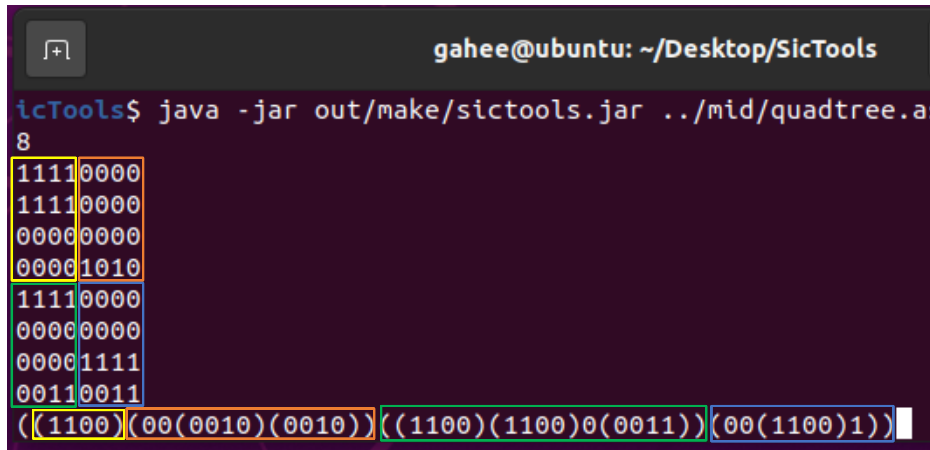
```
rdarr LDS Esize
LDX #0
LDA n
```

rdarr	LDS	(PC)+844
	LDX	#0
	LDA	(PC)+835

Bits nixbpe: -i----

[direct addressing]

## ■ 실행 결과 스크린샷 및 결과에 대한 설명



```
gahee@ubuntu: ~/Desktop/SicTools
icTools$ java -jar out/make/sictools.jar ../mid/quadtree.as
8
11110000
11110000
00000000
00001010
11110000
00000000
00001111
00110011
((1100)(00(0010)(0010))((1100)(1100)0(0011))(00(1100)1))
```

전체 8\*8이 0이나 1로 동일하지 않으므로 4부분으로 나누어서 재귀적으로 각 부분마다 숫자가 동일한지 체크한다. 4부분으로 나눈 부분은 위의 네모로 표시해 두었다. 동일하지 않으면 또 4부분으로 나누어서 계속 recursive하게 call하며 체크해 나가면서 실행되면 위와 같은 결과가 나온다.

스크린샷에서 표시한 부분 별로 결과를 설명하겠다.

**노란색 파트:** 노란색 안의 숫자들은 모두 동일하지 않으므로 4부분으로 나누고 그 각 부분마다 1111, 1111, 0000, 0000으로 같은 숫자끼리 존재하므로 (1100)이 출력된다.

**주황색 파트:** 주황색 안의 숫자들은 모두 동일하지 않으므로 4부분(2\*2)로 나눈다. 각 부분은 0000, 0000, 0010, 0010인데 0000과 0000은 동일한 숫자로 구성되어 있으므로 00으로 출력되고 나머지 두 파트 0010과 0010은 또 4부분(1\*1)으로 나눈다 그럼 각 부분마다 숫자가 하나씩만 존재하므로 그대로 출력한다. 그럼 주황색 파트는 (00(0010)(0010))으로 출력된다.

**초록색 파트:** 초록색 안의 숫자들은 모두 동일하지 않으므로 4부분(2\*2)로 나누고, 각 부분마다 1100, 1100, 0000, 0011로 구성된다. 1100, 1100, 0011은 동일한 숫자로 이루어져 있지 않으므로 각각 또 4부분(1\*1)으로 나눈다. 그럼 숫자가 하나씩만 존재하므로 각 숫자 그대로 출력한다. 그리고 0000은 동일한 숫자로 이루어져 있으므로 0으로 출력된다. 따라서 초록색 파트는 ((1100)(1100)0(0011))이 출력된다.

**파란색 파트:** 파란색 안의 숫자들은 모두 동일하지 않으므로 4부분(2\*2)로 나누고, 각 부분마다 0000, 0000, 1100, 1111이 들어가 있다. 0000, 0000, 1111은 각 부분이 같은 숫자로 구성되어 있으므로 0, 0, 1이 출력이 되고, 1100은 같은 숫자로 구성되어 있지 않으므로 4부분(1\*1)로 나눈다. 그럼 숫자가 하나씩만 존재하게 되므로 각 숫자 그대로 출력된다. 따라서 파란색 파트는 (00(1100)1)이 출력된다.

각 파트를 합쳐서 ((1100)(00(0010)(0010))((1100)(1100)0(0011))(00(1100)1))이라는 출력 결과가 나오게 된다.

## ■ 구현사항/미구현사항/개선점

### <구현사항>

- **알고리즘 구현:** 이 레포트 가장 위에 알고리즘에 대한 개요를 보면 알고리즘을 구현했음을 알 수 있다.
- **입출력 서브루틴 구현**
  - ① 입력에 대한 서브루틴 loop1과 loop2을 구현하였다.
  - ② 숫자출력에 대한 서브루틴 PRINT(PRINT\_0, PRINT\_1)를 구현하였다.

loop1	LDA	#0	PRINT_0	TD	OUTDEV
	TD	INDEV		JEQ	PRINT_0
	JEQ	loop1		LDA	#0
	RD	INDEV		ADD	#48
				WD	OUTDEV

- **2차원 배열 구현:** 각 입력을 indexed addressing을 이용하여 2차원 배열(arr)에 집어넣고, indexed addressing을 이용하여 2차원 배열(arr)에서 값을 읽어왔다.

```
LDA    arr, X
COMP   #0
JEQ    recur
ADDR   S, X
```

### - 3개의 STACK 구현

- ① JSUB을 위한 L 레지스터 값을 보관할 stack
- ② 재귀에서 필요한 입력 값에 대한 stack
- ③ 재귀에서 indexed addressing을 위한 stack

```
main    START    0
        LDA      #gap
        JSUB     stinit
        LDA      #gap_n
        JSUB     stinitn
        LDA      #gap_x
        JSUB     stinitx
```

```
stinit  STA      stackp
        RSUB
push     STA      @stackp
        LDA      stackp
        ADD      #3
        STA      stackp
        RSUB
pop      LDA      stackp
        SUB      #3
        STA      stackp
        LDA      @stackp
        RSUB
```

- **재귀함수 구현:** 값이 동일하게 연속되는지 check함수에서 체크하다가 연속되지 않으면, recur로 와서 (를 출력하고, chnum값을 1/2로 다음에, 2차배열의 구간을 4부분으로 나누어 이를 입력으로 하는 **check**를 **recursive**하게 call하는 방식으로 check 재귀함수를 구현하였다.

### <미구현사항>

- 만약에 입력을 0과 1이 아닌 수나, 입력해야 하는 수보다 더 많거나 적게 입력하였을 때, 에러 메시지가 뜨게 하는 코드는 구현하지 않았다.

### <개선점>

- 항상 똑바른 입력이 들어오지 않을 수도 있으므로 그에 대한 대처를 할 수 있는 코드가 추가하여 개선하면 좋을 것 같다.

### ■ 고찰

재귀함수를 구현하기 위해서 재귀함수를 수행하고 나서 되돌려 놓아야 하는 값이 무엇이 있는지 알아보고, 그 값들은 재귀함수를 호출하기 전에 따로 stack에 저장해 두었다. 재귀함수가 성공적으로 끝나고 돌아오면 다시 되돌려 놓아야 하는 값들을 원상복귀 시켜주었다.

그리고 재귀를 수행하면서 서브루틴으로 반복해서 들어갈 수 있으므로, L 레지스터에 저장된 값이 덮어 쓰여지는 문제를 해결하기 위해, JSUB을 하게 되면 L 레지스터 값을 stack에 넣어 둔다.

Quadtree를 구현하면서 가장 많은 고민을 쏟아부었던 것은 '2차원 배열의 원하는 곳에 어떻게 접근할 수 있을까'였다. 이는 각 재귀함수를 호출 시 num(가장 처음에 입력 받은 값)의 값과 chnum(재귀호출 시 변화한 num의 값)을 이용하여 접근하는 방법을 찾아냈다. 예를 들어  $X = X + ((num - (num - chnum)) * 3 * num)$  이런 식으로 계산을 하여 X register 값을 바꾸어 원하는 배열의 위치에 접근할 수 있다.

그리고 check재귀함수 안에서 숫자가 동일한지 체크를 하는 도중에 동일하지 않은 숫자가 발견되어 recur로 가서 4부분으로 또 나누어 재귀함수를 다시 불러야 한다면, 배열에서 비교하고 싶은 각 부분의 처음 시작 주소는 어떻게 지정할 수 있을지에 대하여 고민을 해보았다. 이에 대해서는, 이전 재귀 함수 check에서 X값을 몇 번 그리고 얼마나 증가시켰는지를 cnt와 cnt2에 저장해 두어서 해결하려고 하였다. 재귀 함수에서 빠져나오고 나서 저장해둔 값을 이용해 계산해서 계산한 값들을, 배열에서 비교하고 싶은 각 부분의 처음 시작 주소로 지정하여 해결하였다.