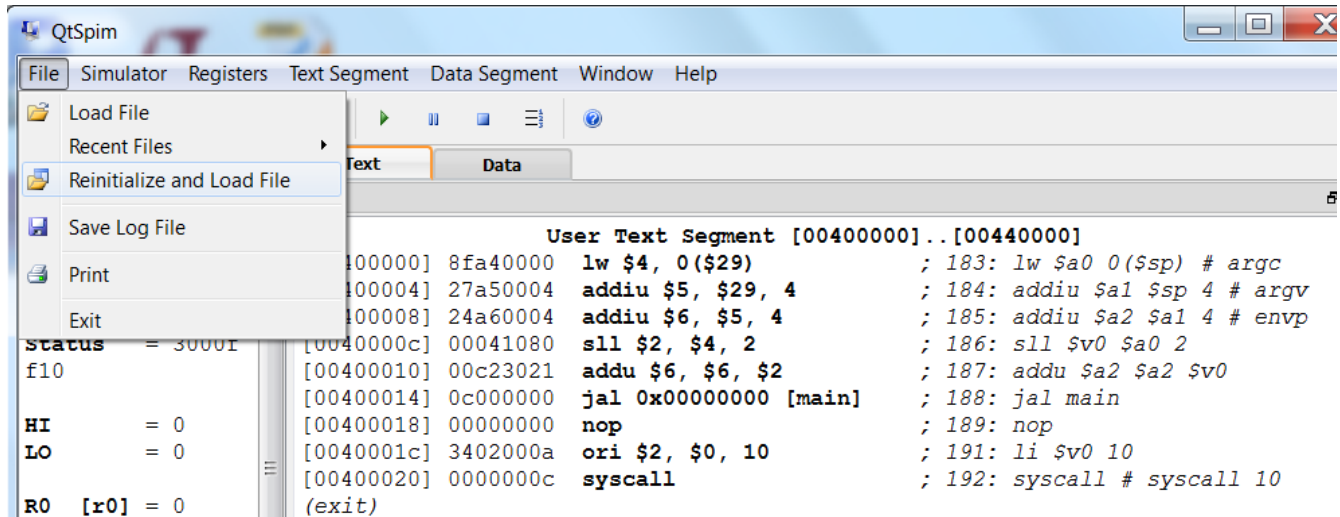


วิธีใช้ QtSpim

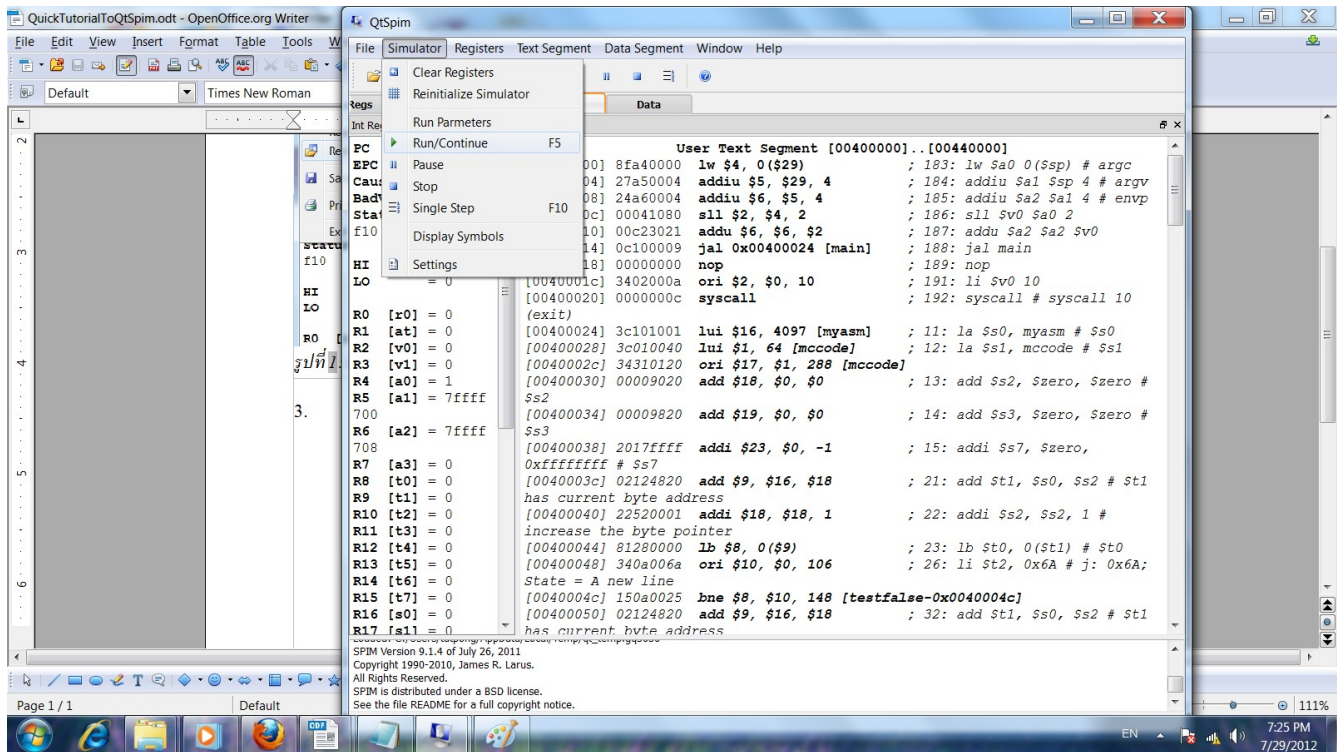
1. เปิดโปรแกรม QtSpim
2. เลือก Reinitialize and Load File (ตามรูปที่ 1) และเลือกไฟล์ .asm (หรือ .s) ที่ต้องการจะรัน



รูปที่ 1: เปิดแอสเซมบลีไฟล์: เลือก Reinitialize and Load File

3. ตอนนี้เราสามารถรันโปรแกรมได้แล้ว เราสามารถรันโปรแกรมแบบรวบเดียวจบ (Run) หรือ จะรันทีละบรรทัด (Single Step) ก็ได้

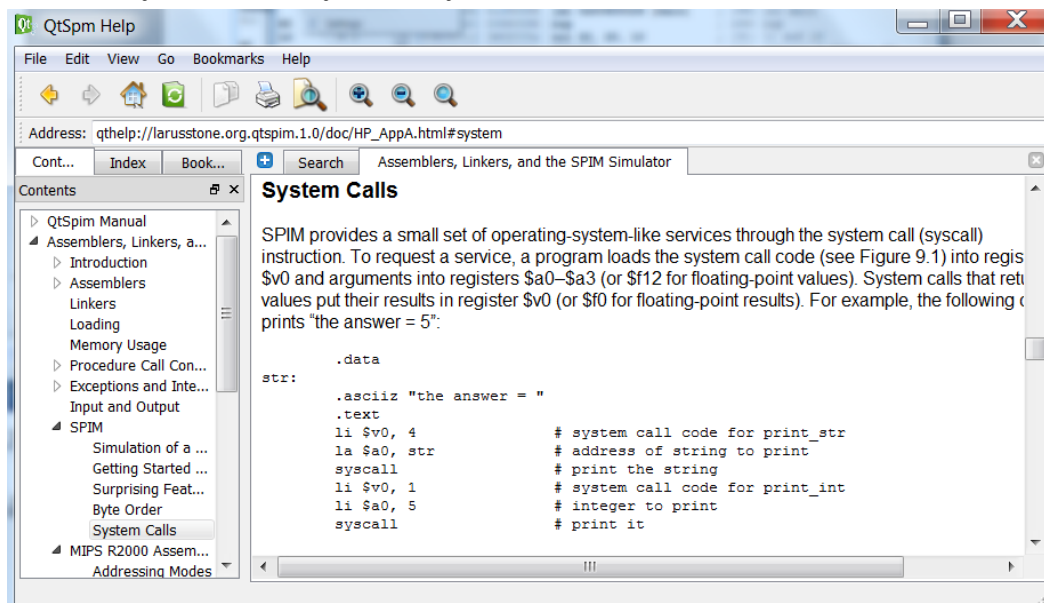
3a. ถ้าต้องการรันรวบเดียวจบ ให้เลือก Run/Continue (ตามรูปที่ 2)



รูปที่ 2: รันโปรแกรมแอสเซมบลีแบบรวดเร็วจาก: เลือก Run/Continue

3b. ถ้าต้องการรันทีละบรรทัด ให้เลือก single step แทน

ต้องการข้อมูลเพิ่มเติม: เปิดดู Help ดังรูปข้างล่าง

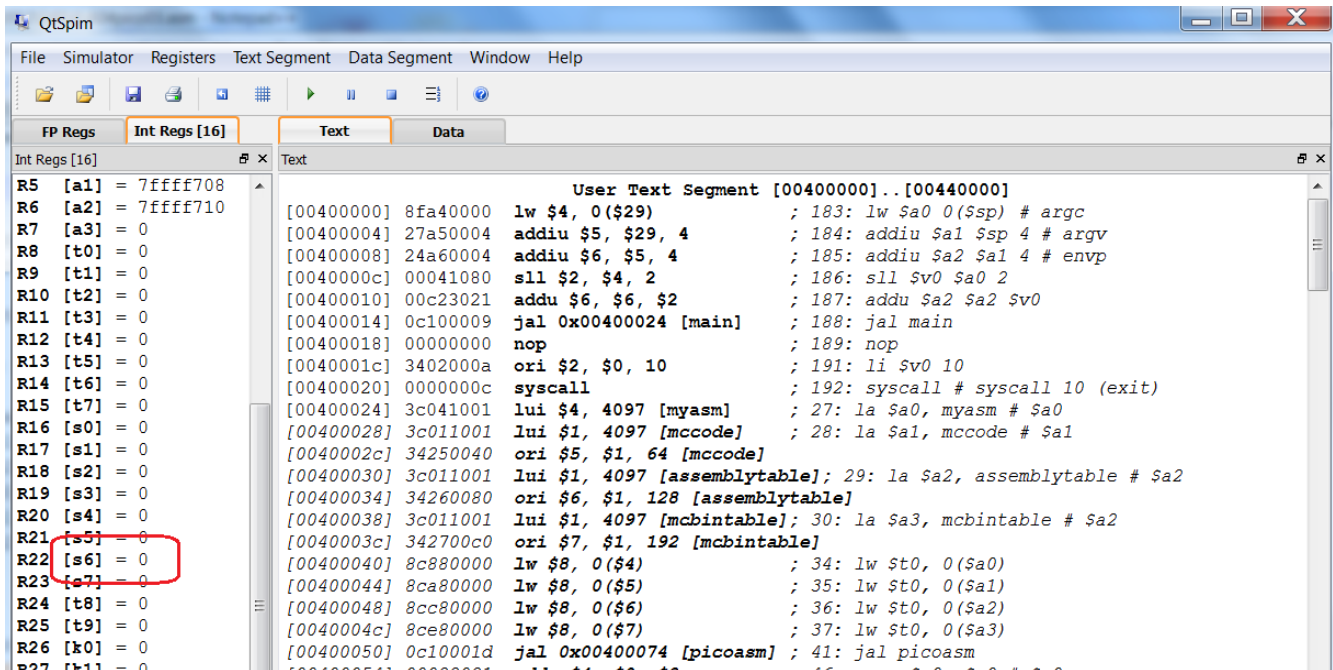


การรัน QtSpim โดยใช้โปรแกรม picoasm.asm เป็นตัวอย่าง พร้อมอธิบายโปรแกรม A04pico03.asm คร่าวๆ

รัชพงศ์ กัตัญญกุล (created 7/29, 2012; last modified 7/30, 2012)

โปรแกรม A04pico.asm เป็น โปรแกรมแอสเซมเบลอรี่่างง่ายทำหน้าที่แปลงคำสั่งแค่คำสั่งเดียว คือ `addi $s6, $s6, 3` ไปเป็นคำสั่งภาษาเครื่องที่ถูกต้อง (ซึ่งคือ 0x22d60003) และแปลงอย่างอื่นเป็น 0x00000000.

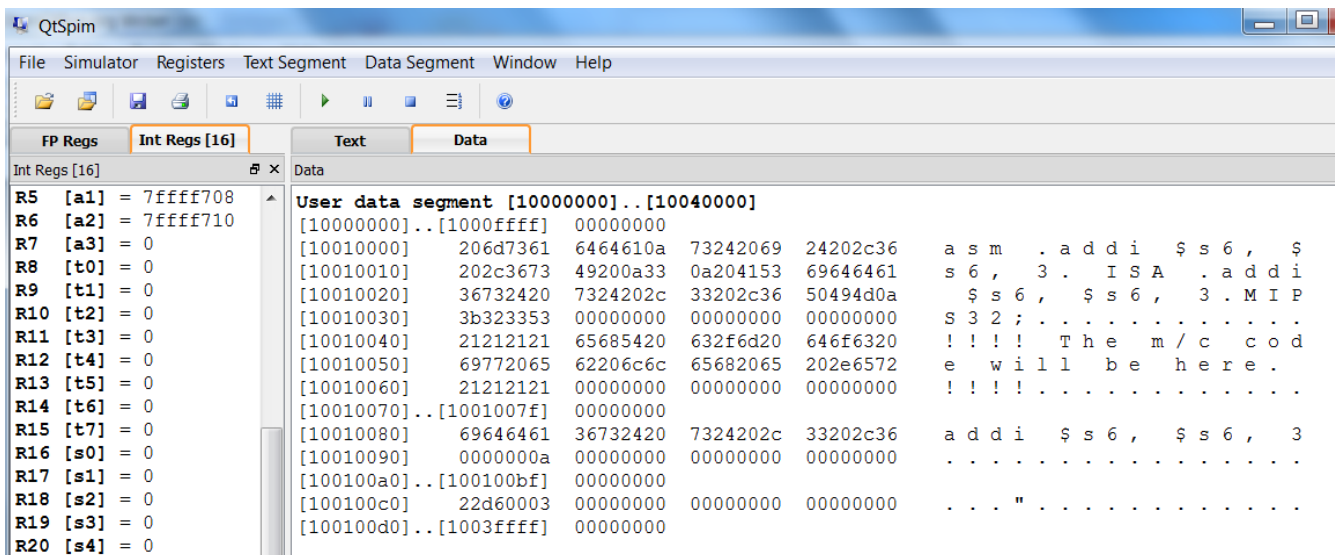
1. เลือก Reinitialize and Load File และเปิดไฟล์ A04pico03.asm ขึ้นมา
2. ตรวจสอบสถานะของ SPIM ก่อนรันโปรแกรม สังเกตว่า ค่ารีจิสเตอร์ `$s6` เป็น 0 (รูปที่ 3) และ ค่าในหน่วยความจำช่วงแอดเดรส 0x10010040 ถึง 0x10010063 เป็นไบนารีที่มีค่าเท่ากับ “!!!! The m/c code will be here. !!!!” (รูปที่ 4)



The screenshot shows the QtSpim MIPS simulator. The 'Int Regs [16]' window is open, displaying the state of registers. Register `$s6` is highlighted with a red box and shows a value of 0. The 'Text' window shows the assembly code for the 'User Text Segment'.

FP Regs	Int Regs [16]	Text	Data
		User Text Segment [00400000]..[00440000]	
		[00400000] 8fa40000 lw \$4, 0(\$29) ; 183: lw \$a0 0(\$sp) # argc	
		[00400004] 27a50004 addiu \$5, \$29, 4 ; 184: addiu \$a1 \$sp 4 # argv	
		[00400008] 24a60004 addiu \$6, \$5, 4 ; 185: addiu \$a2 \$a1 4 # envp	
		[0040000c] 00041080 sll \$2, \$4, 2 ; 186: sll \$v0 \$a0 2	
		[00400010] 00c23021 addu \$6, \$6, \$2 ; 187: addu \$a2 \$a2 \$v0	
		[00400014] 0c100009 jal 0x00400024 [main] ; 188: jal main	
		[00400018] 00000000 nop ; 189: nop	
		[0040001c] 3402000a ori \$2, \$0, 10 ; 191: li \$v0 10	
		[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)	
		[00400024] 3c041001 lui \$4, 4097 [myasm] ; 27: la \$a0, myasm # \$a0	
		[00400028] 3c011001 lui \$1, 4097 [mccode] ; 28: la \$a1, mccode # \$a1	
		[0040002c] 34250040 ori \$5, \$1, 64 [mccode]	
		[00400030] 3c011001 lui \$1, 4097 [assemblytable]; 29: la \$a2, assemblytable # \$a2	
		[00400034] 34260080 ori \$6, \$1, 128 [assemblytable]	
		[00400038] 3c011001 lui \$1, 4097 [mcbintable]; 30: la \$a3, mcbintable # \$a2	
		[0040003c] 342700c0 ori \$7, \$1, 192 [mcbintable]	
		[00400040] 8c880000 lw \$8, 0(\$4) ; 34: lw \$t0, 0(\$a0)	
		[00400044] 8ca80000 lw \$8, 0(\$5) ; 35: lw \$t0, 0(\$a1)	
		[00400048] 8cc80000 lw \$8, 0(\$6) ; 36: lw \$t0, 0(\$a2)	
		[0040004c] 8ce80000 lw \$8, 0(\$7) ; 37: lw \$t0, 0(\$a3)	
		[00400050] 0c10001d jal 0x00400074 [picoasm] ; 41: jal picoasm	

รูปที่ 3: สถานะของรีจิสเตอร์ `s6` ก่อนรันโปรแกรม



รูปที่ 4: สถานะของหน่วยความจำก่อนรันโปรแกรม

3. รันโปรแกรม และ ตรวจสอบสถานะของ SPIM หลังรันโปรแกรม

รูปที่ 5 และ 6 แสดงสถานะของ SPIM หลังรันโปรแกรม ซึ่งรูปที่ 5

โปรแกรม

```
asm \naddi $s6, $s6, 3\n ISA \naddi $s6, $s6, 3\nMIPS32;
```

ที่เริ่มต้นที่แอสแอดเรส 0x1001000 ได้ถูกแปลงเป็นไบนารีภาษาเครื่องโดย

asm \n ถูกแปลงเป็น 0x00000000 มีค่าเทียบเท่าคำสั่ง nop.

addi \$s6, \$s6, 3\n ถูกแปลงเป็น 0x22d60003 มีค่าเทียบเท่าคำสั่ง addi \$s6, \$s6, 3.

ISA \n ถูกแปลงเป็น 0x00000000 มีค่าเทียบเท่าคำสั่ง nop.

addi \$s6, \$s6, 3\n ถูกแปลงเป็น 0x22d60003 มีค่าเทียบเท่าคำสั่ง addi \$s6, \$s6, 3.

และ MIPS32; จะจบท้ายด้วย ; ซึ่งเป็นรหัสจบโปรแกรมที่กำหนดไว้ บรรทัดนี้จึงไม่ถูก

แปลง

หมายเหตุ “\n” จะแทนรหัสขึ้นบรรทัดใหม่

การแปลงตัวหนังสือ (เช่น asm \naddi \$s6, \$s6, 3\n ISA \naddi \$s6, \$s6,

3\nMIPS32;) ไปเป็นไบนารี นี้จะทำโดย ฟังก์ชัน picoasm. โดยเริ่มที่แอสแอดเรส 0x10010040 เป็นต้นไป

และ จะให้ค่าจำนวนคำสั่ง (บรรทัดที่แปลง) ออกมาผ่านรีจิสเตอร์ \$v0.

Text	Data
Data	
User data segment [10000000]..[10040000]	
[10000000]..[1000ffff]	00000000
[10010000]	206d7361 6464610a 73242069 24202c36 a s m . a d d i \$ s 6 , \$
[10010010]	202c3673 49200a33 0a204153 69646461 s 6 , 3 . I S A . a d d i
[10010020]	36732420 7324202c 33202c36 50494d0a \$ s 6 , \$ s 6 , 3 . M I P
[10010030]	3b323353 00000000 00000000 00000000 S 3 2 ;
[10010040]	00000000 22d60003 00000000 22d60003 " "
[10010050]	69772065 62206c6c 65682065 202e6572 e w i l l b e h e r e .
[10010060]	21212121 00000000 00000000 00000000 ! ! ! !
[10010070]..[1001007f]	00000000
[10010080]	69646461 36732420 7324202c 33202c36 a d d i \$ s 6 , \$ s 6 , 3
[10010090]	0000000a 00000000 00000000 00000000
[100100a0]..[100100bf]	00000000
[100100c0]	22d60003 00000000 00000000 00000000 . . . "
[100100d0]..[1003ffff]	00000000

รูปที่ 5: สถานะของหน่วยความจำหลังรันโปรแกรม

หลังจากที่ข้อความภาษาแอสเซมบลีถูกแปลงแล้ว โปรแกรม A04pico03.asm จะเรียก ฟังก์ชัน picoloader ขึ้นมาเพื่อโหลดโปรแกรมจากพื้นที่ใน data segment ไปใส่ใน program segment (MIPS32 เรียก text) และ เปลี่ยน Program Counter (PC) ไปรันโค้ดที่แปลงมา

ตัวอย่างข้างล่าง แสดงแนวคิดของ picoloader:

# a0 : number of instructions	## a0 เก็บจำนวนคำสั่ง
# a1 : base address of mcode	## a1 เก็บค่าแอสแอสเริ่มต้นของไบนารีภาษาเครื่อง
# a2 : base address of loading area	## a2 เก็บค่าแอสแอสเริ่มต้นของตำแหน่งที่จะโหลดไป
# Check if there is sth to load	
loading:	
beqz \$a0, loadingdone	## ถ้า a0 เป็น 0 แปลว่าไม่มีอะไรให้โหลดแล้ว
lw \$t0, 0(\$a1)	## a0 > 0 โหลด
sw \$t0, 0(\$a2)	
addi \$a0, \$a0, -1	## ลดจำนวนคำสั่งที่จะโหลดลงหนึ่ง
addi \$a1, \$a1, 4	## ขยับแอสแอสของคำสั่งที่จะโหลดไปคำสั่งต่อไป
addi \$a2, \$a2, 4	## ขยับแอสแอสของตำแหน่งที่จะโหลดไปตำแหน่งต่อไป
j loading	## โหลดคำสั่งต่อไป
loadingdone:	## เมื่อโหลดเสร็จแล้ว

```
li $t4, 0x03E00008
sw $t4, 0($a2)
```

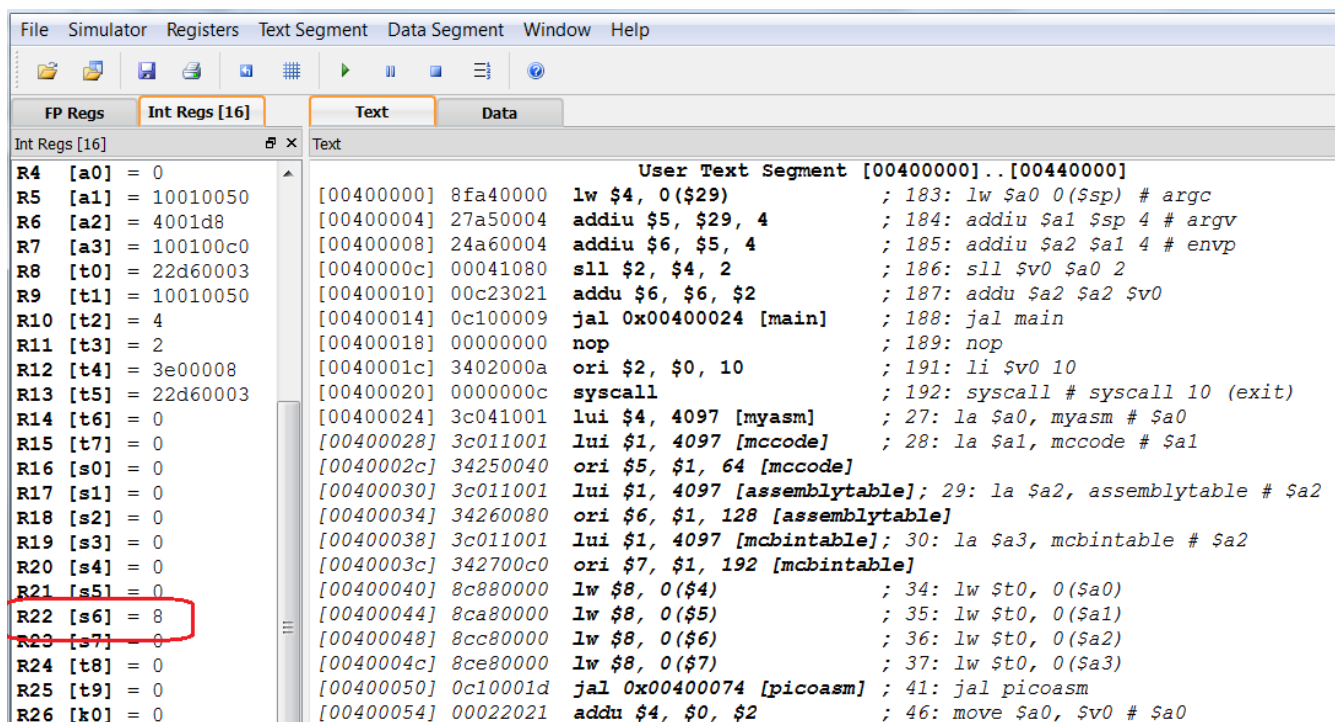
ใส่คำสั่ง "jr \$ra" ไว้ท้ายคำสั่งด้วย

```
# run loaded program
jal loadingarea
```

เรียกรันโปรแกรมที่เพิ่งโหลดไป

และสุดท้ายก็กำหนดตำแหน่งที่จะโหลดเอาไว้

สังเกตว่า picoloader เรียกรันโปรแกรมที่โหลด เหมือนกับเป็น ฟังก์ชัน และ ใส่คำสั่ง "jr \$ra" (ซึ่งมีไบนารีภาษาเครื่องคือ 0x03E00008) เพื่อที่เมื่อหลังจาก CPU ทำคำสั่งที่โหลดเสร็จแล้วจะกลับไปที่ picoloader อีกครั้ง



รูปที่ 6: สถานะของรีจิสเตอร์ s6 หลังรันโปรแกรม

เนื่องจาก picoloader นี้ผมใส่ค่าเริ่มต้นให้ s6 เป็น 2 (เพื่อ debug โปรแกรม) และหลังจากที่ CPU ทำคำสั่งที่โหลด (ซึ่งคือ `nop; addi $s6, $s6, 3; nop; addi $s6, $s6, 3` ที่ได้จาก `picoasm`) แล้ว จึงได้ค่า s6 = 8 เป็นสถานะสุดท้าย

เพื่อทำความเข้าใจโปรแกรม อาจจะลองรันทีละคำสั่งดูได้ (รูปที่ 7) การรันทีละคำสั่งจะช่วยให้เราสามารถตรวจสอบการทำงานของ CPU ในแต่ละคำสั่งว่าเป็นไปตามที่เรากำหนด หรือ คาดหมายไว้หรือไม่

FP Regs	Int Regs [16]	Text	Data
Int Regs [16]		Text	
PC	= 400040	User Text Segment [00400000]..[00440000]	
EPC	= 0	[00400000] 8fa40000 lw \$4, 0(\$29)	; 183: lw \$a0 0(\$sp) # argc
Cause	= 0	[00400004] 27a50004 addiu \$5, \$29, 4	; 184: addiu \$a1 \$sp 4 # argv
BadVAddr	= 0	[00400008] 24a60004 addiu \$6, \$5, 4	; 185: addiu \$a2 \$a1 4 # envp
Status	= 3000ff10	[0040000c] 00041080 sll \$2, \$4, 2	; 186: sll \$v0 \$a0 2
HI	= 0	[00400010] 00c23021 addu \$6, \$6, \$2	; 187: addu \$a2 \$a2 \$v0
LO	= 0	[00400014] 0c100009 jal 0x00400024 [main]	; 188: jal main
R0 [r0]	= 0	[00400018] 00000000 nop	; 189: nop
R1 [at]	= 10010000	[0040001c] 3402000a ori \$2, \$0, 10	; 191: li \$v0 10
R2 [v0]	= 4	[00400020] 0000000c syscall	; 192: syscall # syscall 10 (exit)
R3 [v1]	= 0	[00400024] 3c041001 lui \$4, 4097 [myasm]	; 27: la \$a0, myasm # \$a0
R4 [a0]	= 10010000	[00400028] 3c011001 lui \$1, 4097 [mccode]	; 28: la \$a1, mccode # \$a1
R5 [a1]	= 10010040	[0040002c] 34250040 ori \$5, \$1, 64 [mccode]	
R6 [a2]	= 10010080	[00400030] 3c011001 lui \$1, 4097 [assemblytable]; 29: la \$a2, assemblytable # \$a2	
R7 [a3]	= 100100c0	[00400034] 34260080 ori \$6, \$1, 128 [assemblytable]	
R8 [t0]	= 0	[00400038] 3c011001 lui \$1, 4097 [mcbintable]; 30: la \$a3, mcbintable # \$a2	
R9 [t1]	= 0	[0040003c] 342700c0 ori \$7, \$1, 192 [mcbintable]	
		[00400040] 8c880000 lw \$8, 0(\$4)	; 34: lw \$t0, 0(\$a0)
		[00400044] 8ca80000 lw \$8, 0(\$5)	; 35: lw \$t0, 0(\$a1)

รูปที่ 7: การรันทีละคำสั่ง

โปรแกรม A04Pico03.asm จะเริ่มด้วย การกำหนดตำแหน่งต่างๆในหน่วยความจำ:

```
.data 0x10010000
myasm: .asciiz      "asm \naddi $s6, $s6, 3\n ISA \naddi $s6, $s6, 3\nMIPS32;"

.data 0x10010040
mccode: .asciiz     "!!!! The m/c code will be here. !!!!!"

.data 0x10010080
assemblytable:     .asciiz      "addi $s6, $s6, 3\n"

.data 0x100100C0
mcbintable:        .word 0x22d60003
```

ซึ่ง .data 0x10010000 จะบอก SPIM ว่า กำหนดค่าในหน่วยความจำตั้งแต่แอดเดรส 0x10010000 ด้วยค่า ascii "asm \naddi \$s6, \$s6, 3\n ISA \naddi \$s6, \$s6, 3\nMIPS32;".

ฉลาก myasm ระบุตำแหน่งของข้อความภาษาแอสเซมบลีที่ต้องการจะแปล

ฉลาก mccode ระบุตำแหน่งของไบนารีภาษาเครื่องเมื่อแปลเสร็จ (ตอนเริ่มต้นเราใส่ค่า "!!!! The m/c code will be here. !!!!!" ไปก่อนเพื่อง่ายต่อการ debug)

ฉลาก assemblytable และ mcbintable ใช้ระบุตำแหน่งของพจนานุกรมภาษาแอสเซมบลี และค่าแปล (ไบนารีภาษาเครื่อง) ตามลำดับ: เราต้องการแปล addi \$s6, \$s6, 3\n เป็น 0x22d60003.

การกำหนดค่าในหน่วยความจำนี้จะทำทันทีที่โหลดไฟล์ A04pico03.asm เข้า SPIM

การทำงานหลักๆของ A04pico03.asm จะมีแค่ การเรียก compiler (ฟังก์ชัน picoasm) และ ตามด้วย loader

(ฟังก์ชัน picoloader) ดังนี้:

```
.globl main
.text
main:

# call procedure picoassembler

    # passing parameter values
    la $a0, myasm           # $a0 <- base address of source assembly
    la $a1, mccode          # $a1 <- base address of target m/c code
    la $a2, assemblytable   # $a2 <- base address of mapping assembly table
    la $a3, mcbintable      # $a2 <- base address of mapping m/c binary table

    # call procedure
    jal picoasm

# call procedure picoloader

    # passing parameter values
    move $a0, $v0           # $a0 <- $v0 returned number of lines translated
    la $a1, mccode          # $a1 <- base address of target m/c code
    la $a2, loadingarea     # $a2 <- base address of loading area

    # call procedure
    jal picoloader

end:
    li    $v0,10            # System call code to Exit
    syscall                 # Call SPIM-OS to Exit the program
```

ซึ่ง 3 บรรทัดแรกเป็นการประกาศพื้นที่สำหรับโปรแกรม และตั้งชื่อ ฟังก์ชัน หลัก ซึ่งก็คือ main แล้วตามด้วยการกำหนด ตัวแปรที่จะผ่านให้ ฟังก์ชัน picoasm (โดยใช้ รีจิสเตอร์ \$a0, \$a1, \$a2, และ \$a3) และ เรียก jal picoasm และทำเช่นเดียวกันก่อนเรียก jal picoloader ก่อนจะจบโปรแกรม.

ฟังก์ชัน picoasm เป็นฟังก์ชันหลักของงานนี้ การทำงานของมัน มีหลักการตามโค้ดคร่าวๆดังนี้ (ดูรายละเอียดได้จากไฟล์ A04Pico03.asm)

```
picoasm:
    # $a0 : base address of source assembly
    # $a1 : base address of target m/c code
    # $a2 : base address of mapping assembly table
    # $a3 : base address of mapping m/c binary table

    # return $v0 : a number of lines translated
```



```

# initialization

    add $t0, $a0, $zero    # $t0 <- source pointer
    add $t1, $a1, $zero    # $t1 <- target pointer
    add $t2, $zero, $zero  # $t2 counts the number of lines translated.

readeachline:
# call procedure readline

    # arguments
    move $a0, $t0          # source pointer
    # $a2                  # base address of mapping assembly table

    jal readline

    move $t3, $v0          # readline.$v0 : 0 = no match, 1 = match, 2 = terminate
    move $t0, $v1          # readline.$v1 : source pointer after the line

```

picoasm จะเรียกฟังก์ชัน readline มาเพื่ออ่านภาษาแอสเซมบลีต้นฉบับทีละประโยค และเปรียบเทียบกับคำสั่งในพจนานุกรมที่เราระบุไว้ (ตำแหน่ง assemblytable) เมื่อทำงานเสร็จฟังก์ชัน readline จะผ่านค่า \$v0 เพื่อบอกผลว่า ประโยคที่อ่าน ตรงกับ คำสั่งแอสเซมบลีในพจนานุกรมหรือไม่

picoasm จึงนำผลจาก มาตรวจสอบดูว่า ประโยคตรงกับคำสั่งที่ระบุหรือไม่ ถ้าไม่ตรง ก็ใส่ 0x00000000 ในส่วนไบนารีที่แปล แต่ ถ้าตรงก็ใส่ค่าที่ระบุในพจนานุกรม (4 บรรทัดสุดท้ายข้างล่างนี้).

```

# check if it is terminated
    li $t4, 2
    beq $t3, $t4, endpicoasm

# No, not terminated. Keep going
# increment line counter

    addi $t2, $t2, 1

# check if it is matched
    li $t4, 1
    beq $t3, $t4, itsmatched

# No, not matched. Put 0x00000000 into the target
    li $t5, 0x00000000
    sw $t5, 0($t1)
    addi $t1, $t1, 4

    j readeachline

```

```

endpicoasm:
    move $v0, $t2    # $v0 : a number of lines translated
    jr $ra           # return

# It's matched. Put binary m/c into the target
itsmatched:
    lw $t5, 0($a3)
    sw $t5, 0($t1)
    addi $t1, $t1, 4

    j readeachline

```

ฟังก์ชัน `readline` ทำหน้าที่อ่านข้อความทีละบรรทัด (โดยใช้ `'\n'` เป็นตัวระบุ) จากส่วนของข้อความต้นฉบับ และเปรียบเทียบกับ คำสั่งที่อยู่ในพจนานุกรม

การทำงานของ ฟังก์ชัน `readline` ทำอธิบายคร่าวๆ ได้โดย state diagram ในรูปที่ 8 โค้ดข้างล่าง มีคอมเมนต์ตัวหนาระบุ State เทียบกับ state diagram ในรูปที่ 8 หลักการน่าจะพอเข้าใจจากการอ่านโค้ดข้างล่างประกอบ state diagram.

```

readline:
#   $a0 : source text pointer
#   $a2  : base address of a word-to-match
#   return $v0           # 0 = no match, 1 = match, 2 = terminate
#   return $v1           # $v2 = source pointer after readline

readSrc:                # State 1 in Fig. 8
    lb $t0, 0($a0)      # load byte of the source text
    addi $a0, $a0, 1

# check if it is termination code.
    li $t1, 0x3B        # 0x3B represents ';'
    bne $t0, $t1, noterm

# yes, it's terminated.      # State 2 in Fig. 8
terminated:
    li $v0, 2
    move $v1, $a0

    jr $ra              # return

# no, it's not terminated.   # State 3 in Fig. 8
noterm:

# check if it is an end of line.
    li $t1, 0x0A        # 0x0A represents '\n'
    bne $t0, $t1, noendline

# yes, it's an end of line.  # State 4 in Fig. 8
    # read matching word

```

```

        lb $t2, 0($a2)           # load byte of the matching table      # State 6 in Fig. 8
        addi $a2, $a2, 1
# check if src == matching.
        bne $t0, $t2, nomatch
# yes, it's matched.             # State 7 in Fig. 8
        li $v0, 1
        move $v1, $a0

        jr $ra                   # return

# no, it's not matched.          # State 8 in Fig. 8
nomatch:
        li $v0, 0
        move $v1, $a0

        jr $ra                   # return

# no, it's not an end of line.   # State 5 in Fig. 8
noendline:
        # read matching word      # State 9 in Fig. 8
        lb $t2, 0($a2)           # load byte of the matching table
        addi $a2, $a2, 1
# check if src == matching.
        bne $t0, $t2, nomatchnoend
# yes, it's matched. But, not finish yet.
        # next round
        j readSrc

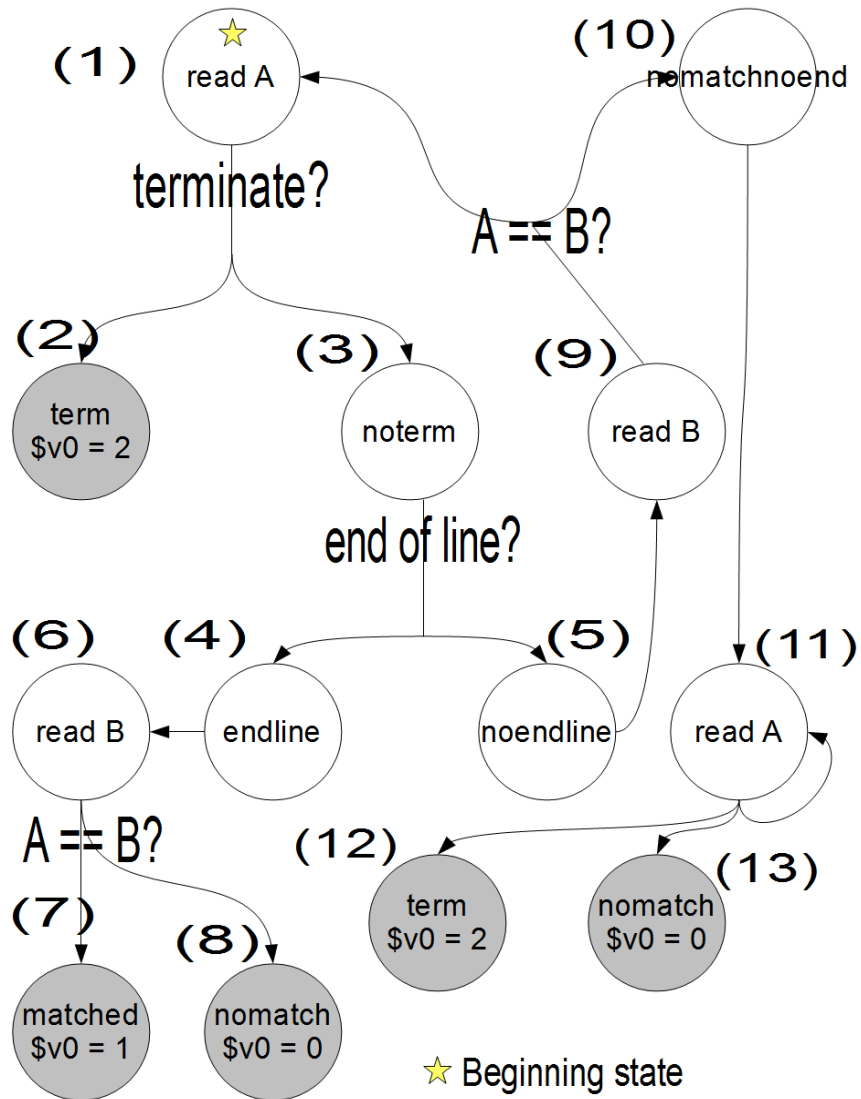
# no, it's not matched nor end.  # State 10 in Fig. 8
nomatchnoend:                   # State 11 in Fig. 8
        lb $t0, 0($a0)           # load byte of the source text
        addi $a0, $a0, 1

# check if it is termination code.
        li $t1, 0x3B             # 0x3B represents ';'
        bne $t0, $t1, nomatchnoterm
# yes, it's terminated.          # State 12 in Fig. 8
        j terminated

# no, it's not terminated nor matched.
nomatchnoterm:
# check if it is an end of line.
        li $t1, 0x0A             # 0x0A represents '\n'
        bne $t0, $t1, nomatchnoend
# yes, it's the end of line, but no match.  # State 13 in Fig. 8
        li $v0, 0
        move $v1, $a0

        jr $ra                   # return

```



รูปที่ 8: State diagram of function readline.