Procedure Call ใน MIPS Assembly

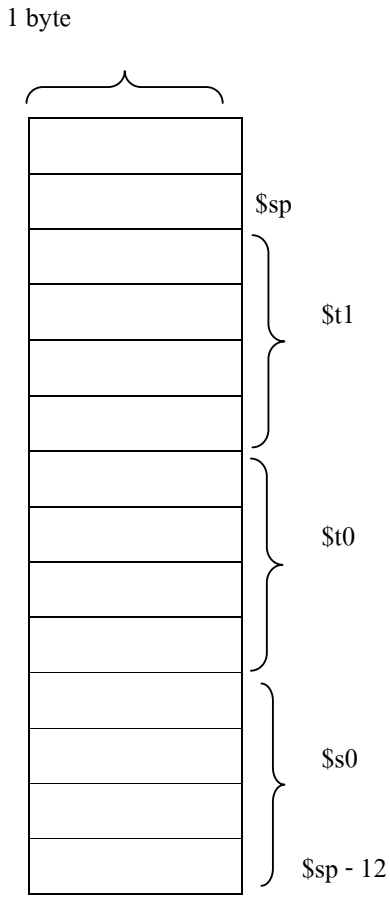
พิจารณาการเปลี่ยนฟังก์ชั่นต่อไปนี้ให้อยู่ในรูป MIPS assembly

```
int leaf_example(int g, int h,
             int i, int j)
{                                   add $t0, $a0, $a1
   f = (g + h) - (i + j);           add $t1, $a2, $a3
                                    sub $s0, $t0, $t1
   return f;                        addu   $v0, $s0, $0
}
```

ให้ parameters ทั้ง 4 คือ g h i และ j อยู่ที่ $a0 $a1 $a2 และ $a3 ตามลำดับ และให้ $v0 ใช้เก็บค่าผลลัพธ์จากการ return ของ function

การแปลงแบบง่ายๆตรงไปตรงมาดังแสดงด้านบนไม่ถูกต้อง เพราะการเขียนทับ $t0 $t1 และ $s0 อาจจะทำให้เกิดความผิดพลาด ณ บริเวณส่วนของโปรแกรมที่เรียกใช้งานฟังก์ชั่น left_example นี้ กล่าวคือโปรแกรมในส่วนดังกล่าว อาจจะมีการใช้ค่ารีจิสเตอร์ $t0 $t1 หรือ $s0 อยู่ ดังนั้นเราจะต้องทำการบันทึก (save) ค่ารีจิสเตอร์เหล่านี้ก่อนการใช้งานในฟังก์ชั่น โดยที่เราจะทำการบันทึกลงบนหน่วยความจำในส่วนที่เราเรียกว่า stack ซึ่งมีส่วนบนสุด (top of stack) อยู่ที่ address ที่มีค่าเก็บอยู่ในรีจิสเตอร์ $sp สำหรับ MIPS stack จะเติบโตจาก address สูงไป address ต่ำ

รูปด้านล่างแสดงกระบวนการบันทึกค่ารีจิสเตอร์ลงบน stack (โค๊ดไฮไลท์สีเหลือง) และเมื่อเราจะ return จากฟังก์ชั่นนี้ เราก็จะคืนค่า (restore) เดิมของรีจิสเตอร์เหล่านี้กลับไป (โค๊ดไฮไลท์สีเขียว) พร้อมกับปรับให้ $sp กลับไปชี้ที่ address เดิมก่อนหน้าที่จะมีการเรียกใช้งาน leaf_example คำสั่งสุดท้ายก่อนจบการทำงานคือคำสั่ง jr $ra ซึ่งเป็นคำสั่งที่บอกให้โปรแกรมกระโดดไปทำงานต่อที่ address ที่ระบุไว้ในค่า $ra (นั่นคือเปลี่ยนค่า PC ไปที่ตำแหน่งที่ระบุไว้ใน $ra) ซึ่งเราจะมาทำความเข้าใจกันในขั้นต่อไปเมื่อเราพูดถึงการเขียนโค๊ดในส่วนการเรียกใช้งาน leaf_example (ส่วน caller)




```
addi   $sp, $sp, -12
sw     $t1, 8 ($sp)
sw     $t0, 4 ($sp)
sw     $s0, 0 ($sp)

add    $t0, $a0, $a1
add    $t1, $a2, $a3
sub    $s0, $t0, $t1
addu   $v0, $s0, $0

lw     $s0, 0 ($sp)
lw     $t0, 4 ($sp)
lw     $t1, 8 ($sp)
addiu  $sp, $sp, 12

jr     $ra
```

ที่ผ่านมาเราได้พิจารณาในส่วนของโปรแกรมที่เกี่ยวข้องกับตัวฟังก์ชั่นโดยตรง (ส่วน callee) ต่อไปเราจะมาดูในส่วนของโปรแกรมที่เรียกใช้งาน leaf_example (ส่วน caller) บ้าง เช่นในกรณีเราเรียกใช้งาน leaf_example ดังแสดงด้านล่าง

```
a++;
leaf_example (0, 2, 5, 6);
a--;
```

การเรียกใช้งานฟังก์ชั่น จะต้องมีการส่งผ่าน parameter ไปยังฟังก์ชั่นนั้น โดยเราจะใช้ stack เป็นตัวส่งผ่าน parameter โดยจะ push paremeter ลง stack จากขวาสุดมาซ้ายสุด เช่นกรณีดังตัวอย่าง จะ push 6 ตัวแรกที่สุด ตามด้วย 5 และ 2 และ 0 หลังจากเสร็จสิ้นการส่งผ่านนี้แล้ว โปรแกรมก็จะกระโดดไปทำงานที่ leaf_example จนกระทั่ง return กลับออกมา ก็จะมีการทำการคืนค่า $sp กลับไปเป็นค่าเดิมก่อนที่จะมีการเรียกใช้งาน leaf_example (ตัวอย่างการแปลงโค๊ดในส่วนนี้แสดงด้านล่างนี้)

```
addiu  $t0, $t0, 1 // a++;

addiu  $sp, $sp, -4
sw  $t0, 0($sp) //เก็บค่า $t0 ลงบน stack ไว้ก่อน เพราะอาจมีการใช้งาน $t0 ใน leaf_example

addiu  $sp, $sp, -16
li     $t4, 6
sw     $t4, 12 ($sp)
li     $t4, 5
sw     $t4, 8 ($sp)
li     $t4, 2
sw     $t4, 4 ($sp)
li     $t4, 0
sw     $t4, 0 ($sp)

jal    leaf_example

addiu  $sp, $sp, 16

lw     $t0, 0($sp)
addiu  $sp, $sp, 4

addiu  $t0, $t0, -1 // a--;
```

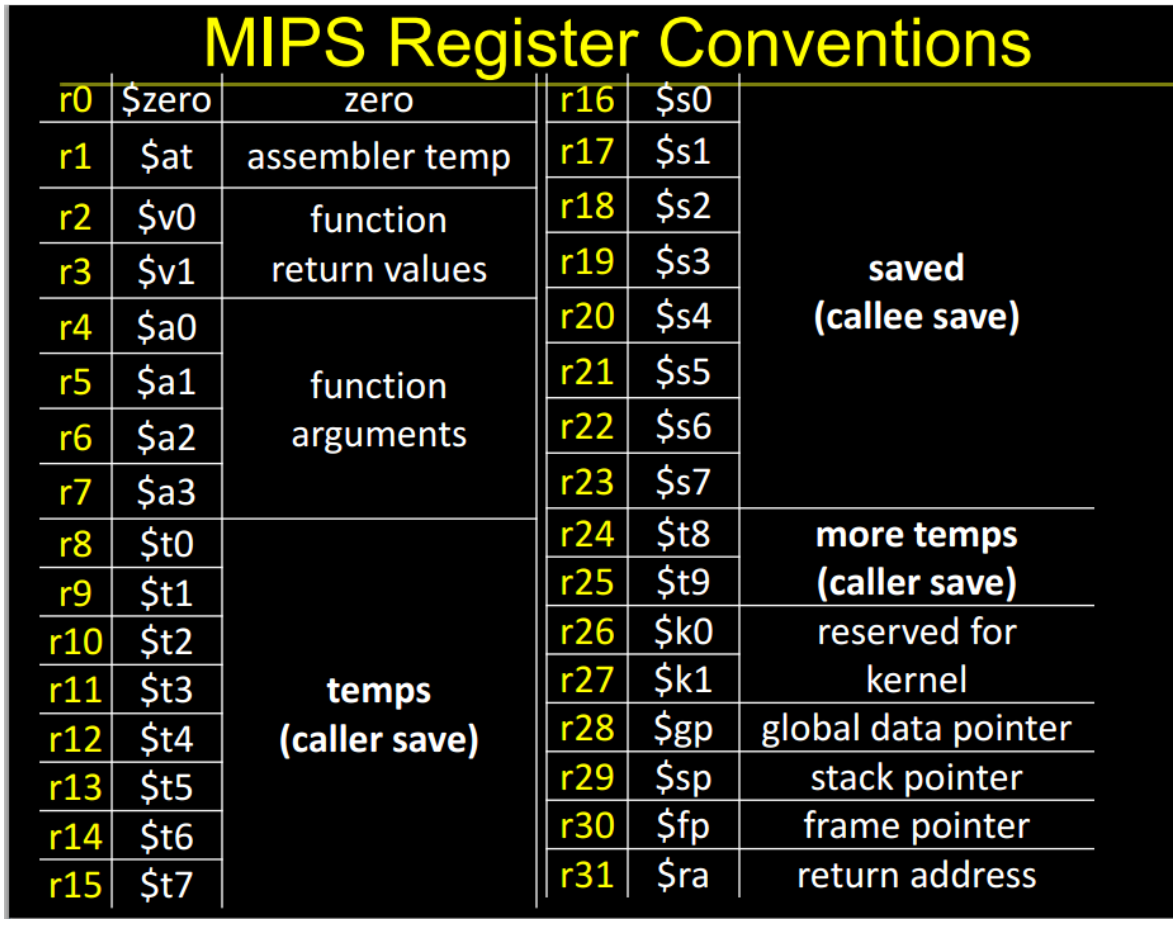
การเรียกใช้งาน leaf_example ใช้คำสั่ง jal (jump and link) ซึ่งนอกจากจะทำให้โปรแกรมกระโดดไปทำงานที่ address ของคำสั่งแรกในฟังก์ชั่น leaf_example แล้ว ยังจะทำการเก็บค่า address ของคำสั่งที่จะทำงานเป็น คำสั่งแรกหลังจาก return จาก leaf_example มา (ในกรณีนี้คือคำสั่ง addiu $sp, $sp, 16) ลงไปไว้ที่รีจิสเตอร์ $ra อีกด้วย

Caller-save & callee-save รีจิสเตอร์

จะเห็นได้ว่าการที่ทั้งส่วน caller และ callee ต้องเก็บค่ารีจิสเตอร์ที่จะใช้งานในทั้งสองฝั่ง อาจทำให้เกิด การเก็บที่ซ้ำซ้อนกันโดยไม่จำเป็น ดังนั้นจึงมีข้อตกลงในการเก็บค่ารีจิสเตอร์โดยแบ่งรีจิสเตอร์เป็นสองส่วนดังต่อไปนี้

- caller-save = ความรับผิดชอบในการ save (จัดเก็บค่า) อยู่ที่ code ที่จะเรียกใช้งาน function
- callee-save = ความรับผิดชอบในการ save อยู่ที่ code ในฟังก์ชันที่ถูกเรียก

ภาพด้านล่างต่อไปนี้แสดงการใช้งานรีจิสเตอร์ของ MIPS ที่ได้มีการตกลงกันไว้และเป็นที่ยอมรับโดยทั่วไป



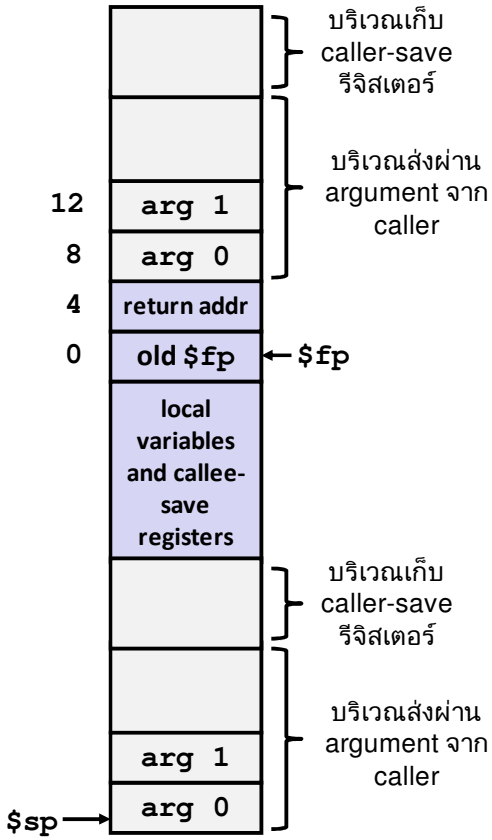
ถ้าเราทำตามข้อตกลงข้างต้นนี้แล้ว ส่วนโค๊ดของ callee สามารถลดจำนวนคำสั่งลงได้ 4 คำสั่ง

```
addi   $sp, $sp, -4
sw     $t1, 8 ($sp)
sw     $t0, 4 ($sp)
sw     $s0, 0 ($sp)

add    $t0, $a0, $a1
add    $t1, $a2, $a3
sub    $s0, $t0, $t1
addu   $v0, $s0, $0

lw     $s0, 0 ($sp)
lw     $t0, 4 ($sp)
lw     $t1, 8 ($sp)
addiu $sp, $sp, 4

jr     $ra
```

Stack คืออะไรและเอาไว้ใช้งานอะไร

- ส่วนของ memory ที่ใช้เก็บข้อมูลที่จำเป็นเวลา function / procedure ถูกเรียกใช้งาน
- แต่ละ function / procedure มี frame ของตัวเอง
- ขอบเขตของ frame กำหนดด้วย frame pointer $fp กับ stack pointer $sp
- Caller (โปรแกรมส่วนที่เรียกใช้งาน function)
    - save ค่า register ที่อยู่ใน set caller save
    - ส่งผ่าน arguments ลงบน stack
- Callee (function ที่ถูกเรียกใช้งาน)
    - save ค่า return address ลงบน stack
    - เก็บค่า frame pointer เดิมลงบน stack
    - จองที่สำหรับ local variables
    - save ค่า register ที่อยู่ใน callee save

รูปแบบของ stack frame ที่เราจะใช้ในการโปรแกรม assembly ในวิชานี้เป็นไปตามภาพด้านล่างนี้ นิสิตอาจจะเห็นได้ว่า stack frame ที่อาจารย์ให้ไว้นี้ ไม่เป็นไปตาม stack frame ที่ MIPS เสนอให้ใช้ แต่อาจารย์เลือกมาใช้เพราะจะทำให้นิสิตเข้าใจได้ง่ายมากกว่า stack frame แท้ๆของ MIPS ที่ค่อนข้างจะยุ่งยากกว่า สำหรับโปรแกรมเมอร์ภาษา assembly มือใหม่ ตามภาพด้านล่างนี้ offset จาก $fp และ $sp จะเพิ่มหรือลดทีละ 4 bytes



frame pointer สำหรับแต่ละเฟรมจะชี้ไปที่ตำแหน่งที่เก็บค่า frame pointer -ของ frame เก่าเสมอ ดังนั้นค่า offset ไปที่ตำแหน่ง (address) ของ argument แต่ละตัวที่ส่งผ่านเข้ามาจาก caller จะเป็นค่าที่คงที่ ตายตัว เสมอในทุกๆ เฟรม ถ้าทำตาม stack frame ที่ตกลงกันนี้ เราสามารถเขียน assembly ของ leaf_example ได้ดังแสดงด้านล่าง ส่วนที่เพิ่มเติมขึ้นมาก็คือ

- การจัดเตรียม frame ของ callee โดยการเก็บค่า return address และ $fp ของ frame ก่อนหน้านี้ลงบน stack
- การนำค่า argument ที่ส่งผ่านจาก caller เข้ามาทาง stack มาเก็บไว้ที่ $a0 $a1 $a2 และ $a3 ตามลำดับ เพื่อทำการประมวลผลในขั้นตอนต่อไป

```
addiu  $sp, $sp, -4
sw     $ra, 0($sp)
addiu  $sp, $sp, -4
sw     $fp, 0($sp)
addu   $fp, $sp, $0 // ปรับ $fp ใหม่ให้เท่ากับ $sp ล่าสุด
```

```
addiu  $sp, $sp, -4
sw  $t1, 8 ($sp)
sw  $t0, 4 ($sp)
sw  $s0, 0 ($sp)

lw $a0, 8($fp)
lw $a1, 12($fp)
lw $a2, 16($fp)
lw $a3, 20($fp)

add    $t0, $a0, $a1
add    $t1, $a2, $a3
sub    $s0, $t0, $t1
addu   $v0, $s0, $0

lw  $s0, 0 ($sp)
lw  $t0, 4 ($sp)
lw  $t1, 8 ($sp)
addiu $sp, $sp, 4
lw     $fp, 0($sp)
addiu $sp, $sp, 4
lw     $ra, 0($sp)
addiu $sp, $sp, 4

jr $ra
```

ต่อไปเราจะมาดูตัวอย่างการเขียนโปรแกรม assembly สำหรับฟังก์ชั่นที่คำนวณค่า factorial ของ n โดยในภาษา C เราสามารถเขียนโดยใช้ recursion ได้ดังนี้

```
int fact (int n){
   if (n<1) return 1;
   else return fact(n-1)*n;
}
```

[ไปที่สไลด์เรื่อง stack และ recursion]