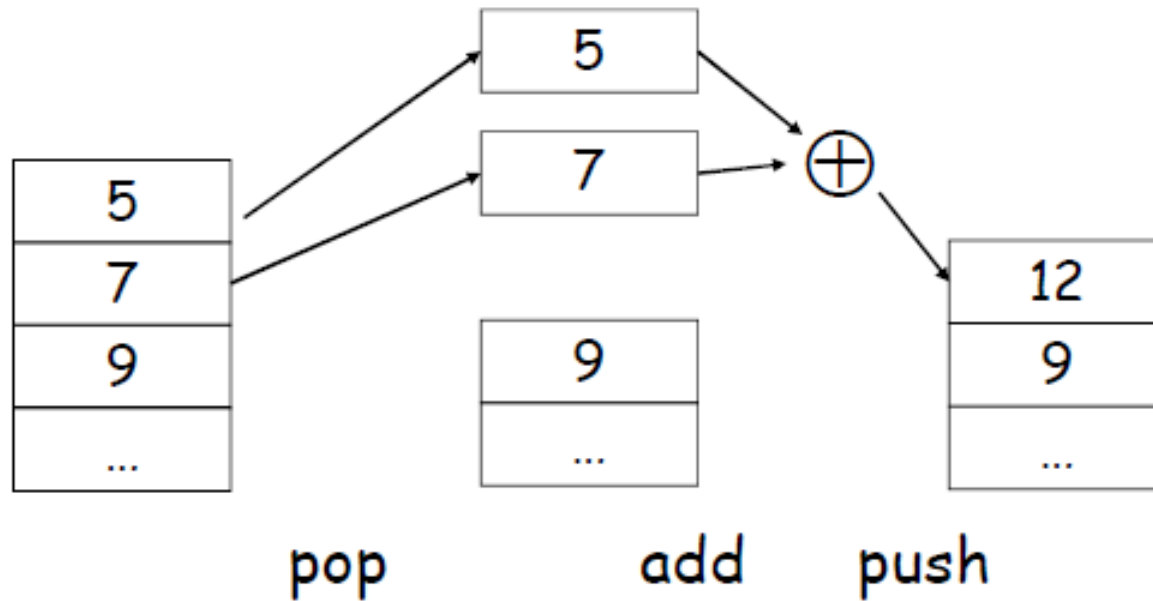


สถาปัตยกรรมแบบ stack และ การผลิตโค้ด

สถาปัตยกรรมแบบ stack

- มี model การประมวลผลที่ง่าย
- ไม่มีตัวแปรหรือรีจิสเตอร์มาเกี่ยวข้อง
- ค่า intermediate result ถูกเก็บอยู่ใน stack
- คำสั่งแต่ละคำสั่งในสถาปัตยกรรมแบบนี้
 - นำ operands มาจากส่วนบนของ stack
 - ดึง operands เหล่านั้นออกจาก stack
 - คำนวณหาผลลัพธ์โดยใช้ตัวปฏิบัติการ (operation) ที่เลือกมา
 - เก็บค่าผลลัพธ์โดยการ push ค่านี้ลงไปที่ stack

การบวก โดยใช้ stack



ตัวอย่าง โปรแกรมของสถาปัตยกรรมแบบ stack

- พิจารณาคำสั่ง 2 รูปแบบ
 - push i : ใส่ค่าของ integer i ไปที่ส่วนบนสุดของ stack (top of stack)
 - add : นำค่าสองค่า (pop) ออกมาจากส่วนบนของ stack และบวกค่าทั้งสองเข้าด้วยกัน จากนั้น push ผลลัพธ์ลงไปที่ stack
- ในการจะบวกเลข $7 + 5$ ทำได้ดังนี้
 - push 7
 - push 5
 - add

ข้อได้เปรียบของ stack machine

- Operation ต่างๆนำ operand มาจาก stack และคำนวณผลลัพธ์ใส่กลับลงไปที่ stack
- Operand และผลลัพธ์มาจากที่เดียวกัน
 - ง่ายต่อการคอมไพล์ ผลิต โค้ดคล้ายๆกัน
 - คอมไพล์ไม่มีความซับซ้อนมาก
- ตำแหน่งที่จะนำ operand มาคำนวณไม่ต้องมีการระบุชัดเจน
 - นำมาจากส่วนบนของ stack เสมอ

ข้อได้เปรียบของ stack machine

- คำสั่งไม่ต้องการระบุ operand
- คำสั่งไม่ต้องการระบุตำแหน่งที่จะใช้เก็บผลลัพธ์
- ตัวอย่างเช่นการใช้คำสั่ง add แทนที่จะต้องเป็น
add \$1, \$2, \$3
 - คำสั่งแต่ละคำสั่ง ใช้จำนวนบิตในการ encode น้อย
 - โปรแกรม binary มีขนาดเล็ก
- ตัวอย่างของชุดคำสั่งแบบ stack machine เช่น Java Bytecodes

ตัวอย่างการประมวลผลโดยใช้ stack

$3 + (5 + 7)$	$3 + 5 + 7$	จาก infix expression เราจะ รู้ลำดับก่อนหลังของการ push กับ add เวลาสร้าง โค้ดประมวลผลแบบ stack ได้อย่างไร?
push 3	push 3	
push 5	push 5	
push 7	add	
add	push 7	
add	add	ใช้ expression tree ช่วย

$$3 + 5 + 7 * 8 / 9 \% 10$$

นิสิตสร้าง expression tree และเขียนโค้ด
ประมวลผลแบบ stack ได้หรือไม่?

$$3 + 5 + 7 * 8 / 9 \% 10$$

push 3

push 5

add

push 7

push 8

mul

push 9

div

push 10

mod

add

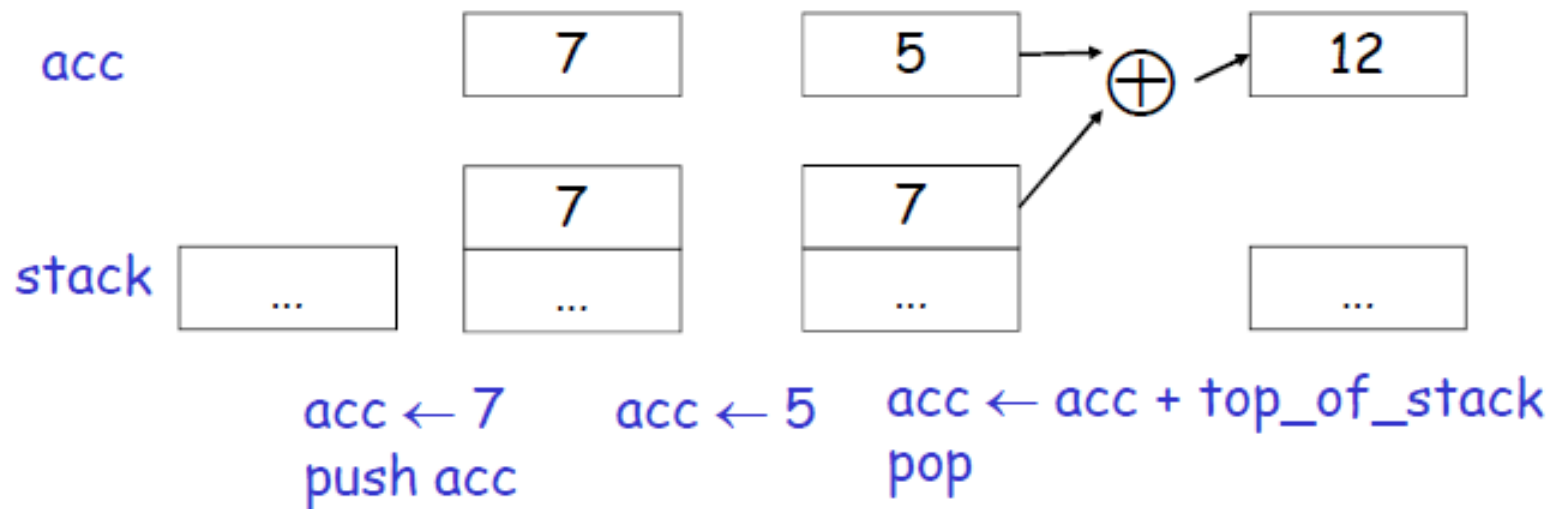
N-Register Stack Machine

- ลูกผสมระหว่าง register machine และ stack machine
- ตำแหน่ง n ตำแหน่งที่อยู่ส่วนบนของ stack จะถูกเก็บไว้ใน register
- เราจะสนใจและพิจารณา 1-register stack machine
 - โดย register หนึ่งเดียวนี้จะเรียกว่า accumulator

การบวกใน 1-register stack machine

- Stack machine เดิมต้องติดต่อกับ memory ถึง 3 ครั้ง
 - อ่านค่าสองค่าจาก stack
 - เขียนค่าผลลัพธ์ลง stack
- ถ้ามี accumulator (acc) มาช่วย:
$$\text{acc} \leftarrow \text{acc} + \text{top_of_stack}$$
- ติดต่อกับ memory ครั้งเดียว นอกนั้นติดต่อผ่าน acc ที่เป็น register ซึ่งการอ่านและเขียนทำได้เร็วกว่า memory มาก

ตัวอย่างการคำนวณ



ตัวอย่างการคำนวณ

$$3 + (7 + 5)$$

Code	Acc	Stack
$\text{acc} \leftarrow 3$	3	<init>
push acc	3	3, <init>
$\text{acc} \leftarrow 7$	7	3, <init>
push acc	7	7, 3, <init>
$\text{acc} \leftarrow 5$	5	7, 3, <init>
$\text{acc} \leftarrow \text{acc} + \text{top_of_stack}$	12	7, 3, <init>
pop	12	3, <init>
$\text{acc} \leftarrow \text{acc} + \text{top_of_stack}$	15	3, <init>
pop	15	<init>

คุณสมบัติในการคำนวณ โดย stack machine

- การคำนวณ expression ทุกชนิดจะต้องไม่ทำให้ stack เดิมที่มีมาก่อนการคำนวณนี้เปลี่ยนแปลง
- นั่นคือการคำนวณ sub-expression ทั้งหลายจะต้องคงสถานะของ stack เดิมไว้หลังจากการคำนวณเสร็จสิ้น
 - เรียกว่าการคำนวณหาผลลัพธ์ของ expression ใดๆ จะต้อง preserve stack

MIPS Assembly กับ Stack Machine

- สร้างคอมไพเลอร์ที่ผลิตโค้ด MIPS assembly สำหรับ 1-register stack machine
- รันโค้ดใน SPIM ซึ่งเป็น simulator ของ MIPS ISA
- ให้ accumulator อยู่ใน \$a0
- Stack อยู่ในหน่วยความจำ
 - เติบโตจาก high ไปที่ low address
 - เป็นไปตาม convention ของสถาปัตยกรรมแบบ MIPS
- \$sp เก็บตำแหน่งถัดไปจาก top of stack (tos)
 - tos อยู่ที่ตำแหน่ง $\$sp + 4$

โค้ดสำหรับ $7 + 5$

Stack machine
assembly

```
acc ← 7  
push acc  
  
acc ← 5  
acc ← acc + top_of_stack  
  
pop
```

MIPS assembly (จำลอง
การทำงานถ้ารันบน stack
machine)

```
li $a0 7  
sw $a0 0($sp)  
addiu $sp $sp -4  
li $a0 5  
lw $t1 4($sp)  
add $a0 $a0 $t1  
addiu $sp $sp 4
```


โค้ดสำหรับ $3 + (7 + 5)$

Stack machine
assembly

```
acc ← 3  
push acc  
acc ← 7  
push acc  
acc ← 5  
acc ← acc + top_of_stack  
pop  
acc ← acc + top_of_stack  
pop
```

MIPS assembly (จำลอง
การทำงานถ้ารันบน stack
machine)

นิสิตลองทำส่วนนี้ดู

โค้ดสำหรับ $3 + (7 + 5)$

Stack machine assembly

```
acc ← 3
push acc
acc ← 7
push acc
acc ← 5
acc ← acc + top_of_stack
pop
acc ← acc + top_of_stack
pop
```

MIPS assembly (จำลองการทำงาน ถ้ารันบน stack machine)

```
li $a0, 3
sw $a0, 0($sp)
addi $sp, $sp, -4
li $a0, 7
sw $a0, 0($sp)
addi $sp, $sp, -4
li $a0, 5
lw $t1, 4($sp)
add $a0, $a0, $t1
addi $sp, $sp, 4
lw $t1, 4($sp)
add $a0, $a0, $t1
addi $sp, $sp, 4
```

แนวทางการผลิต โค้ด

- สำหรับ expression e เราจะผลิต โค้ด MIPS ที่:
 - ใส่ผลลัพธ์ของ e ไว้ที่ $\$a0$
 - ทุกๆครั้งที่คำนวณ e เราจะต้อง preserve $\$sp$ และ stack (นั่นคือสถานะของ $\$sp$ และ stack หลังจากคำนวณ e จะต้องเหมือนกับตอนก่อนคำนวณ e)
- ให้ฟังก์ชัน $cgen(e)$ ให้ผลลัพธ์เป็น โค้ดที่ผลิตจาก input ที่เป็น expression e

โค้ดสำหรับค่าคงที่

- ทำได้ตรงไปตรงมา โดยที่ก๊อปปี้ค่าคงที่นั้นไปไว้ที่ accumulator:

`cgen(i) = li $a0 i`

- เราจะใช้สีแทนแทนส่วนของโค้ดที่อยู่ในช่วงเวลาคอมไพล์
- เราจะใช้สีน้ำเงินแทนส่วนของโค้ดที่จะนำไปรัน ณ เวลาจริง

โค้ดสำหรับการบวก

```
cgen( $e_1 + e_2$ ) =  
  cgen( $e_1$ )  
  sw $a0 0($sp)  
  addiu $sp $sp -4  
  cgen( $e_2$ )  
  lw $t1 4($sp)  
  add $a0 $t1 $a0  
  addiu $sp $sp 4
```

```
cgen( $e_1 + e_2$ ) =  
  cgen( $e_1$ )  
  print "sw $a0 0($sp)"  
  print "addiu $sp $sp -4"  
  cgen( $e_2$ )  
  print "lw $t1 4($sp)"  
  print "add $a0 $t1 $a0"  
  print "addiu $sp $sp 4"
```

ความผิดพลาดในการพยายาม optimize

- โค้ดด้านล่างก็อปปี้ผลลัพธ์จาก e_1 เข้าไปไว้ที่ $\$t1$
- คำนวณ e_2 แล้วนำผลลัพธ์มาบวกเข้ากับค่าใน $\$t1$
- นำผลลัพธ์สุดท้ายเก็บไว้ใน accumulator
- นิเสืหาข้อผิดพลาดของการแปลแบบนี้ได้หรือไม่

```
cgen( $e_1 + e_2$ ) =  
    cgen( $e_1$ )  
    move  $\$t1$   $\$a0$   
    cgen( $e_2$ )  
    add  $\$a0$   $\$t1$   $\$a0$ 
```

คุณสมบัติสำคัญในการผลิตโค้ดแบบนี้

- มีลักษณะเหมือนการเติม template ที่ส่วนของโค้ด e_1 และ e_2 จะต้องถูกเติมเพิ่มเข้ามา
- การผลิตโค้ดสำหรับ stack machine ในลักษณะนี้จึงเป็นแบบ recursive
- ดังนั้นเราสามารถทำการผลิตโค้ดได้โดย traverse AST แบบ top-down และใช้เทคนิค recursive-descent

โค้ดสำหรับการลบ

```
cgen( $e_1 - e_2$ ) =  
    cgen( $e_1$ )  
    sw $a0 0($sp)  
    addiu $sp $sp -4  
    cgen( $e_2$ )  
    lw $t1 4($sp)  
    sub $a0 $t1 $a0  
    addiu $sp $sp 4
```


โค้ดสำหรับ control flow

$\text{cgen}(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$

$\text{cgen}(e_1)$

sw \$a0 0(\$sp)

addiu \$sp \$sp -4

$\text{cgen}(e_2)$

lw \$t1 4(\$sp)

addiu \$sp \$sp 4

beq \$a0 \$t1 true_branch

false_branch:

$\text{cgen}(e_4)$

b end_if

true_branch:

$\text{cgen}(e_3)$

end_if: