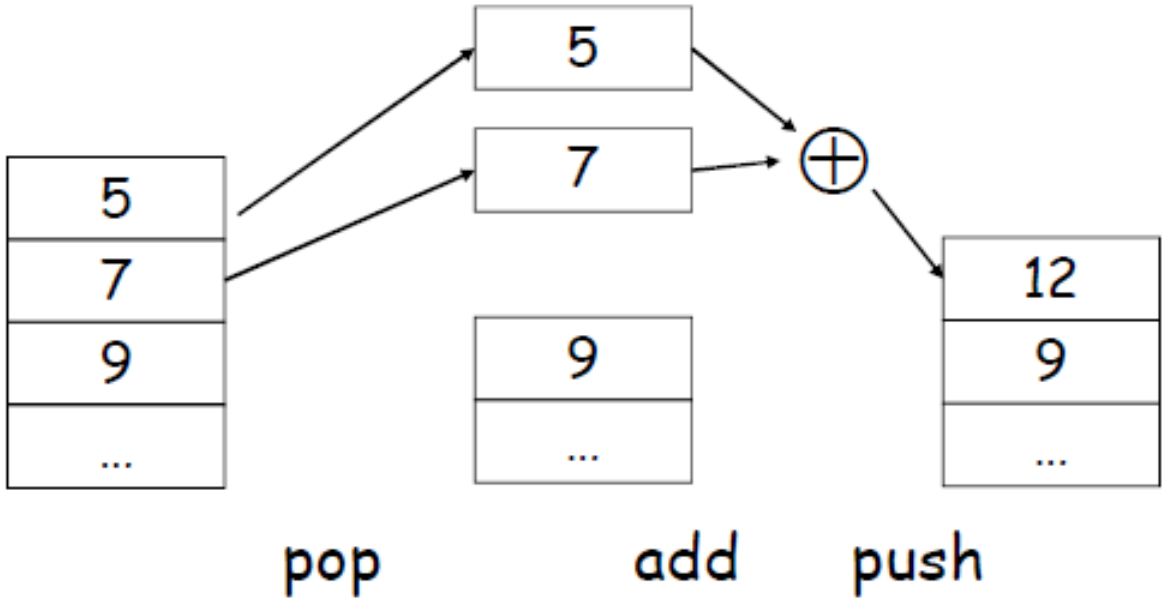
สถาปัตยกรรมแบบ stack และ การผลิตโค๊ด

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

การบวกโดยใช้ stack



ตัวอย่างโปรแกรมของสถาปัตยกรรมแบบ stack

- พิจารณาคำสั่ง 2 รูปแบบ
  - push i : ใสค่าของ 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 ผลิตโค๊ดคล้ายๆกัน
  - คอมไพล์ไม่มีความซับซ้อนมาก
- ตำแหน่งที่จะนำ operand มาคำนวณไม่ต้องมีการระบุชัดเจน
  - นำมาจากส่วนบนของ stack เสมอ

ข้อได้เปรียบของ stack machine

- คำสั่งไม่ต้องมีการระบุ operand
- คำสั่งไม่ต้องมีการระบุตำแหน่งที่จะใช้เก็บผลลัพธ์
- ตัวอย่างเช่นการใช้คำสั่ง add แทนที่จะต้องเป็น add $1, $2, $3
  - คำสั่งแต่ละคำสั่งใช้จำนวนบิทในการ encode น้อย
  - โปรแกรม binary มีขนาดเล็ก
- ตัวอย่างของชุดคำสั่งแบบ stack machine เช่น Java Bytecodes

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

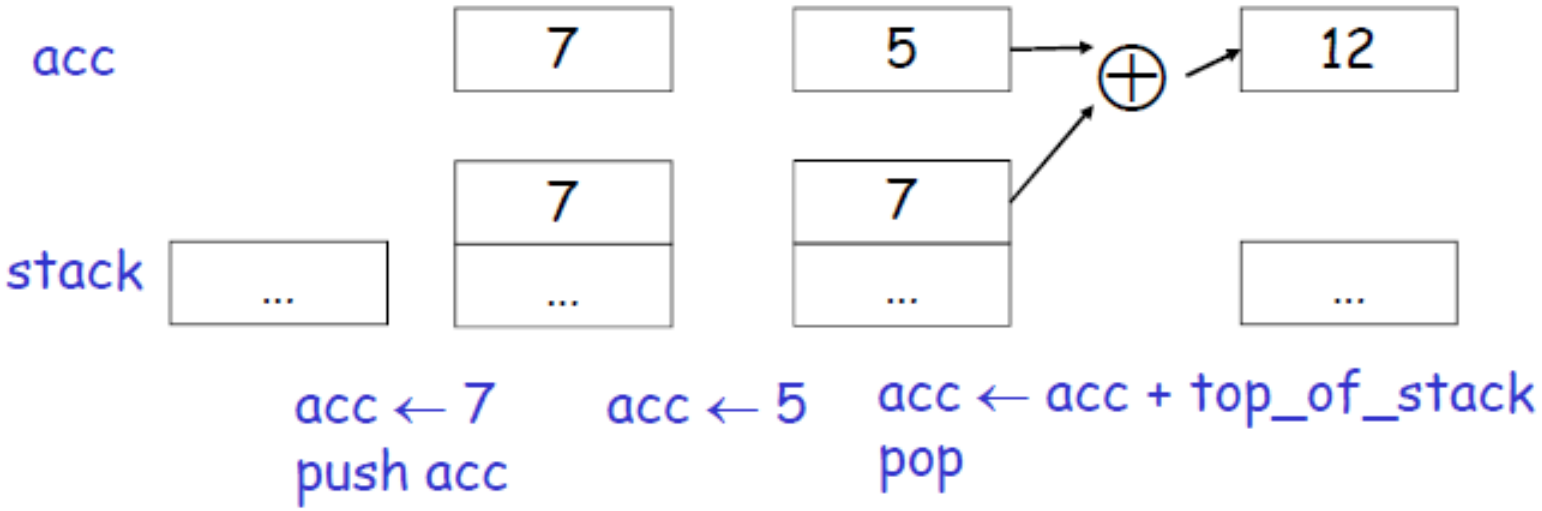
- Stack machine เดิมต้องติดต่อกับ memory ถึง 3 ครั้ง
  - อ่านค่าสองค่าจาก stack
  - เขียนค่าผลลัพธ์ลง stack
- ถ้ามี accumulator (acc) มาช่วย:

  acc <- acc + top_of_stack
- ติดต่อกับ memory ครั้งเดียว นอกนั้นติดต่อผ่าน acc ที่เป็น register ซึ่งการอ่านและเขียนทำได้เร็วกว่า memory มาก

การคำนวณผ่าน stack machine

- พิจารณา expression op(e_1, … , e_n) ที่มี operation คือ op และมี operand n ตัว
  - e_1, …, e_n เป็น sub-expression
- สำหรับแต่ละ e_i (0 < i < n)
  - คำนวณหา e_i
  - ได้ผลลัพธ์เก็บไว้ที่ acc และ push ผลลัพธ์ลงบน stack
  - คำนวณหา e_n ใส่ค่าผลลัพธ์เข้าที่ acc แต่ไม่ต้อง push ลง stack
- ค่อยๆไล่ pop ทั้ง n-1 ค่าออกจาก stack เพื่อจะคำนวณหา op(e_1, … e_{n-1}, acc)
- acc ← op(e_1, … e_{n-1}, acc)

ตัวอย่างการคำนวณ



ตัวอย่างการคำนวณ

3 + (7 + 5)

Code	Acc	Stack
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

คุณสมบัติในการคำนวณโดย stack machine

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

MIPS Assembly กับ Stack Machine

- สร้างคอมไพเลอร์ที่ผลิตโค๊ด MIPS assembly สำหรับ 1-register stack machine
- รันโค๊ดใน 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

ทบทวน MIPS ISA

[ตัดไปที่สไลด์เรื่อง MIPS ISA ที่เราเคยเรียนกันในวิชาสถาปัตยกรรมคอมพิวเตอร์]

โค๊ดสำหรับ 7 + 5

Stack machine assembly

```
acc ← 7
push acc

acc ← 5
acc ← acc + top_of_stack

pop
```

MIPS assembly (จำลองการทำงานถ้ารันบน 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
```

แนวทางการผลิตโค๊ด

- สำหรับ expression e เราจะผลิตโค๊ด MIPS ที่:
  - ใส่ผลลัพธ์ของ e ไว้ที่ $a0
  - ทุกๆครั้งที่คำนวณ e เราจะต้อง preserve $sp และ stack (นั่นคือสถานะของ $sp และ stack หลังจากคำนวณ e จะต้องเหมือนกับตอนก่อนคำนวณ e)
- ให้ฟังก์ชั่น cgen(e) ให้ผลลัพธ์เป็นโค๊ดที่ผลิตจาก input ที่เป็น expression e

โค๊ดสำหรับค่าคงที่

- ทำได้ตรงไปตรงมาโดยที่ก๊อปปี้ค่าคงที่นั้นไปไว้ที่ accumulator:

cgen(i) = li $a0 i

- เราจะใช้สีแดงแทนส่วนของโค๊ดที่อยู่ในช่วงเวลาคอมไพล์
- เราจะใช้สีน้ำเงินแทนส่วนของโค๊ดที่จะนำไปรัน ณ เวลาจริง

โค๊ดสำหรับการบวก

```
cgen(e1 + e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        add $a0 $t1 $a0
        addiu $sp $sp 4
```

```
cgen(e1 + e2) =
    cgen(e1)
    print "sw $a0 0($sp)"
    print "addiu $sp $sp -4"
    cgen(e2)
    print "lw $t1 4($sp)"
    print "add $a0 $t1 $a0"
    print "addiu $sp $sp 4"
```

ความผิดพลาดในการพยายาม optimize

- โค๊ดด้านล่างก๊อปปี้ผลลัพธ์จาก e_1 เข้าไปไว้ที่ $t1
- คำนวณ e_2 แล้วนำผลลัพธ์มาบวกเข้ากับค่าใน $t1
- นำผลลัพธ์สุดท้ายเก็บไว้ใน accumulator
- นิสิตหาข้อผิดพลาดของการแปลแบบนี้ได้หรือไม่

```
cgen(e1 + e2) =
        cgen(e1)
        move $t1 $a0
        cgen(e2)
        add $a0 $t1 $a0
```

คุณสมบัติสำคัญในการผลิตโค๊ดแบบนี้

- มีลักษณะเหมือนการเติม template ที่ส่วนของโค๊ด e_1 และ e_2 จะต้องถูกเติมเพิ่มเข้ามา
- การผลิตโค๊ดสำหรับ stack machine ในลักษณะนี้จึงเป็นแบบ recursive
- ดังนั้นเราสามารถทำการผลิตโค๊ดได้โดย traverse AST แบบ top-down และใช้เทคนิค recursive-descent

โค๊ดสำหรับการลบ

cgen(e_1 - e_2) =

```
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    sub $a0 $t1 $a0
    addiu $sp $sp 4
```

โค๊ดสำหรับ control flow

cgen(if e_1 = e_2 then e_3 else e_4) =

```
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch
```

```
false_branch:
  cgen(e4)
  b end_if
true_branch:
  cgen(e3)
end_if:
```