รายงาน

Project 1 Computer Architecture

จัดทำโดย

กษิดิ์ยศ หารไพโรจน์	640610621
ธนภัทร สมสิทธิ์	640610639
เจตพล กอบกำ	640612083
ณัฐชนน นันทศรี	640612086
ศรัณย์ กิมานุวัฒน์	640612192

เสนอ

รศ.ดร.ศันสนีย์ เอื้อพันธ์วิริยะกุล

รายงานเล่มนี้เป็นส่วนหนึ่งของกระบวนวิชาสถาปัตยกรรมคอมพิวเตอร์

รหัสวิชา 261304

ภาคการศึกษาที่ 1 ปีการศึกษา 2566

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเชียงใหม่

คำนำ

รายงาน Project Computer Architecture ฉบับนี้เป็นส่วนหนึ่งของวิชา Computer Architecture รหัสกระบวนวิชา 261304 โดยมีวัตถุประสงค์เพื่อศึกษาและทำความเข้าใจเกี่ยวกับภาษาแอสเซมบลี โครงสร้าง ลักษณะการทำงานของโปรแกรม Multiplication, Recursive Combination และ Optimized Exponentiation รวมไปถึงการ ออกแบบ Behavioral Simulator ของหน่วยประมวลผลหนึ่ง

คณะผู้จัดทำหวังเป็นอย่างยิ่งว่ารายงานฉบับนี้จะเป็นประโยชน์อย่างยิ่งแก่ผู้ที่สนใจหากมีข้อเสนอแนะหรือข้อผิดพลาดประการใดทางคณะผู้จัดทำขอน้อมรับไว้มา ณ ที่นี้

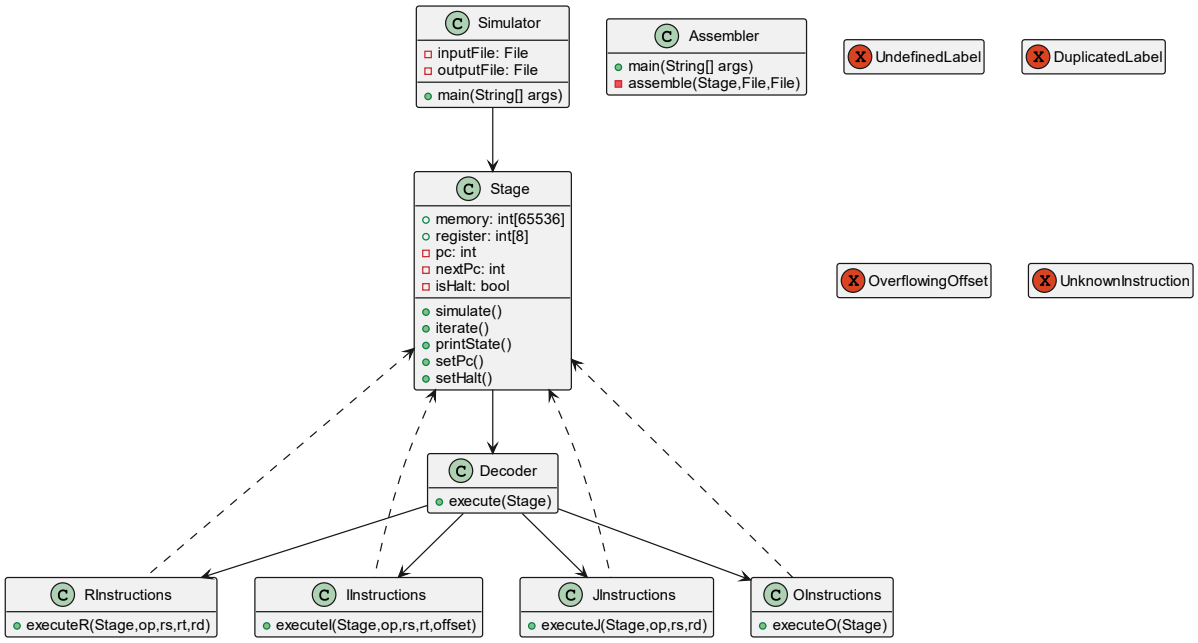
คณะผู้จัดทำ
9 ตุลาคม 2566

สารบัญ

สารบัญตาราง

สารบัญรูปภาพ

การทำงานของของโปรแกรม



แผนภาพคลาสของโปรเจค

ในตัวโปรเจคจะถูกแบ่งออกเป็นสองส่วนใหญ่ได้แก่ Assembler ซึ่งเป็นส่วนที่แปลงโค้ดภาษาแอสเซมบลีกลายเป็นรหัสเครื่อง และ ส่วน Simulator ซึ่งอ่านรหัสเครื่องแต่ละบรรทัดแล้วนำมาเลียนแบบและรายงานสถานะการทำงานของหน่วยประมวลผลสมมุติ

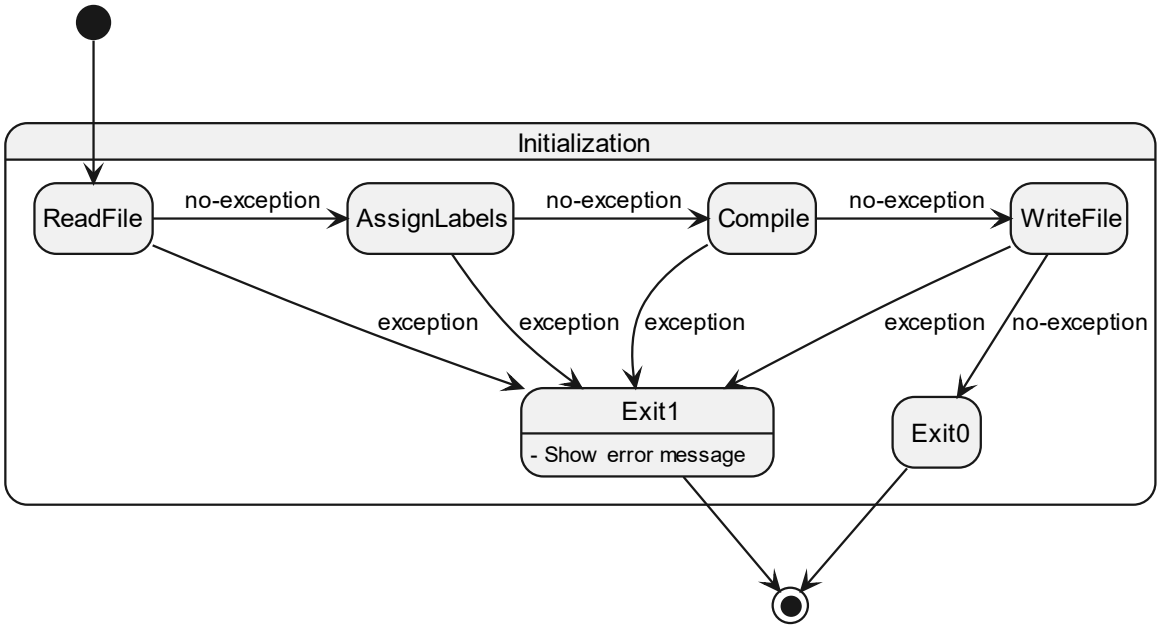
โปรแกรม Assembler

- ***`Assembler.java`*** คือคลาสหลักที่มีหน้าที่แปลงโค้ดภาษาแอสเซมบลีกลายเป็นรหัสเครื่อง ซึ่งคำนึงถึงการรองรับเงื่อนไขที่ผิดพลาดภายในตัวข้อมูลด้วยเช่น การไม่ค้นพบไฟล์ที่จะแปลงภาษา การค้นพบ labels ที่ซ้ำกัน การค้นพบการเรียก label ที่ไม่เคยถูกประกาศ การไม่พบเจอ operator การใช้ตัวเลขที่มีค่าสูงเกินการกำหนด ฯลฯ

โปรแกรม Simulator

- ***`Simulator.java`*** เป็นคลาสหลักของโปรแกรม ซึ่งมีหน้าที่ในการสร้าง Stage สำหรับการประมวลผล โดยอ่านรหัสเครื่องที่ถูกแปลงด้วยตัว Assembler และนำข้อมูลรหัสเครื่องไปใส่ในหน่วยความจำของ Stage หลังจากนั้นจะสั่งให้ Stage เริ่มจำลองการทำงานในที่สุด โดยที่คลาสนี้ต้องคำนึงถึงการรองรับเงื่อนไขที่ผิดพลาดเช่นกัน ได้แก่ การไม่ค้นพบไฟล์ที่นำไปอ่าน และ การค้นพบบรรทัดของรหัสเครื่องที่ไม่สามารถแปลงให้เป็นตัวเลข 32 บิต ได้
- ***`Stage.java`*** คือคลาสที่เก็บข้อมูลสถานะต่าง ๆ ของหน่วยประมวลผลจำลอง เช่น ข้อมูลในหน่วยความจำ หน่วยเรจิสเตอร์ เลขลำดับของคำสั่งที่กำลังทำงาน เลขลำดับของคำสั่งที่จะทำงานถัดไป และอื่น ๆ นอกจากนี้ยังมีหน้าที่รายงานสถานะของหน่วยประมวลผลและใช้ Decoder ในการถอดรหัสชุดคำสั่งเช่นเดียวกัน
- ***`Decoder.java`*** คือคลาสที่ทำการ ประเมินประเภทของคำสั่ง ถอดรหัสคำสั่ง แล้วเลือกใช้คลาส Instruction ที่เหมาะสมในการดำเนินการคำสั่ง
- ***`RInstruction.java`*** เป็นคลาสที่ดำเนินการคำสั่งประเภท R ซึ่งรวมไปด้วยคำสั่ง ADD กับ NAND
- ***`IInstruction.java`*** เป็นคลาสที่ดำเนินการคำสั่งประเภท I ได้แก่ LW, SW, และ BEQ
- ***`JInstruction.java`*** เป็นคลาสที่ดำเนินการคำสั่ง JALR โดยเฉพาะ

- *`OInstruction.java`* เป็นคลาสที่ดำเนินการคำสั่งประเภท O ได้แก่ HALT กับ NOOP

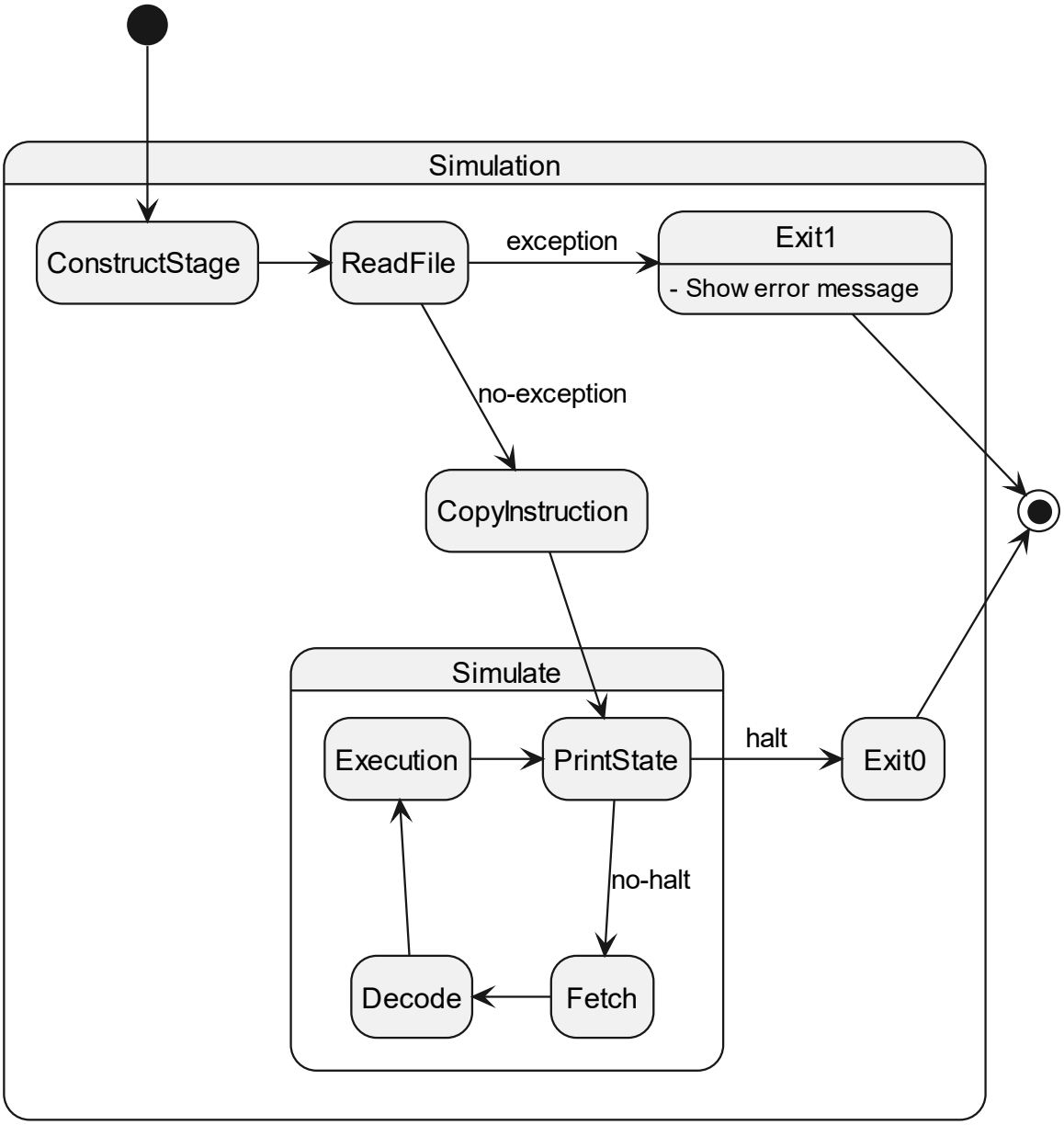


แผนภาพสถานะของโปรแกรม Assembler

ในการทำงานของโปรแกรม Assembler จะถูกควบคุมโดยคลาส Assembler.java ทั้งหมด โดยที่ตัวโปรแกรมมีขั้นตอนการทำงานดังนี้

1. อ่านไฟล์ที่เก็บโค้ดภาษาแอสเซมบลีบลี้ไว้
2. หาการประกาศของ labels ต่าง ๆ ในตัวโค้ด แล้วบันทึกเลขบรรทัดที่อยู่ไว้
3. แปลงโค้ดออกมาให้อยู่ในรูปของรหัสคอมพิวเตอร์
4. เขียนข้อมูลรหัสคอมพิวเตอร์ลงไปในไฟล์ปลายทาง

โดยถ้าโปรแกรมค้นพบเงื่อนไขข้อมูลที่ไม่ถูกต้อง ตัวโปรแกรมจะส่งค่า 1 ออกมา แล้วออกจากการทำงานทันที นอกจากนี้ ถ้าไม่เกิดข้อผิดพลาดใด ๆ โปรแกรมจะส่งค่า 0 แทน



แผนภาพสถานะของโปรแกรม Simulator

ส่วนของโปรแกรม Simulator จะสามารถแบ่งหน้าที่การทำงานของแต่ละคลาสได้ดังนี้

- ***Simulator.java*** จะทำงานส่วนภายนอกของกรอบ Simulate ทั้งหมด ซึ่งประกอบไปด้วย การสั่งสร้าง Stage *(ConstructStage)* การอ่านขอมูลรหัสคอมพิวเตอร์ *(ReadFile)* การบันทึกรหัสคอมพิวเตอร์ลงใน Stage *(CopyInstruction)* แล้วสั่งเรียก Stage ทำงานในที่สุด
- ***Stage.java*** จะมีหน้าที่ในการเรียกให้ Decoder ทำงานไปเรื่อย ๆ จนกว่าจะเกิดสถานะ halt ขึ้นมา และรายงานสถานะ *(PrintState)* ออกมาหลังการทำงานในแต่ละรอบ
- ***Decoder.java*** มีหน้าที่ในการดึงคำสั่งปัจจุบันออกจากหน่วยความจำของ Stage *(Fetch)* นำคำสั่งไปถอดรหัส (Decode) แล้วเรียกใช้คลาส type-instructions ในการดำเนินการคำสั่งในที่สุด
- ***Type Instructions*** ซึ่งประกอบไปด้วย R I J และ O จะมีหน้าที่ในการดำเนินการคำสั่งที่ถูกถอดรหัสแล้ว

การทดสอบโปรแกรม Assembler

การทดสอบความสามารถในการทำงานของโปรแกรม Assembler ขณะที่โปรเจคตัวนี้ยังอยู่ในขั้นตอนการพัฒนามีวัตถุประสงค์หลักคือการตรวจสอบว่าโปรแกรมสามารถค้นพบเงื่อนไขที่ไม่เหมาะสมหรือไม่ การตรวจสอบความถูกต้องของการแปลงภาษาแอสเซมบลีเป็นรหัสเครื่องนั้นจะถูกผนวกอยู่ในกระบวนการทดสอบอื่น ๆ

ผลการทดสอบ

ข้อมูลการทดสอบ	ผลของการทดสอบ	ความคาดหมาย
โค้ดไม่มีข้อผิดพลาดใด ๆ	0: ไม่เกิดปัญหาใด	เป็นไปตามที่คาดหมาย
มีจำนวน arguments ในการเรียกฟังก์ชันใช้ไม่พอ	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
มีจำนวน arguments ในการเรียกฟังก์ชันมากเกินไป	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
ไฟล์อินพุตไม่มีอยู่จริง	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
ไฟล์เอาต์พุตไม่มีอยู่จริง	0: ไม่เกิดปัญหาใด	เป็นไปตามที่คาดหมาย
ทั้งไฟล์อินพุตและไฟล์เอาต์พุตไม่มีอยู่จริง	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีการใช้ label ที่ไม่ได้มีการประกาศไว้	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีการประกาศ label ซ้ำ 1 รอบ	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีการประกาศ label ซ้ำ 2 รอบ	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีการประกาศ label แต่ไม่ได้ใช้	0: ไม่เกิดปัญหาใด	เป็นไปตามที่คาดหมาย
ชื่อ label ไม่ได้ขึ้นต้นด้วยตัวหนังสือ	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
ชื่อ label มีความยาวเกิน 6	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีบรรทัดที่ไม่เขียนอะไรเลย	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีบรรทัดที่ขาด operator	0: ไม่เกิดปัญหาใด	เป็นไปตามที่คาดหมาย
โค้ดมีคำสั่ง R-type ที่มี operand ไม่เพียงพอ	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีคำสั่ง I-type ที่มี operand ไม่เพียงพอ	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีคำสั่ง J-type ที่มี operand ไม่เพียงพอ	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีคำสั่ง J-type ที่มี operand มากเกินไป	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีคำสั่ง O-type ที่มี operand มากเกินไป	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
เลขหลังคำสั่ง .fill ไม่ได้เป็นจำนวนเต็ม 32 บิต	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย
โค้ดมีคำสั่ง .fill ที่มี operand น้อยเกินไป	1: แปลงโค้ดไม่ผ่าน	เป็นไปตามที่คาดหมาย

ตารางผลทดสอบโปรแกรม Assembler

สรุปผลการทดสอบ

จากการทดสอบโปรแกรม Assembler ไม่ได้เกิดปัญหาไม่พบเจอเงื่อนไขที่ไม่ต้องการใด ๆ

Multiplication Algorithm

Data: *mcand*, where *mcand* is a 32-bit integer.
and *mplier*, where *mplier* is a 32-bit integer.
Result: $p = mcand \cdot mplier$

1 $p \leftarrow 0$
2 $i \leftarrow 0$
3 **while** $i \neq 2^{16}$ **do**
4 | **if** $\sim (i \ \& \ mcand) \neq -1$ **then**
5 | | $p \leftarrow p + mplier$
6 | **end**
7 | $i \leftarrow i + i$
8 | $mcand \leftarrow mcand + mcand$
9 **end**

อัลกอริทึมการคูณเลข

โค้ดในภาพเป็นอัลกอริทึมการคูณเลข 15 บิต สองตัวด้วยกันที่นำมาใช้ทดสอบด้วยโค้ดภาษาแอสเซมบลี ซึ่งโค้ดที่ว่าจะถูกในไปแปลงเป็นรหัสคอมพิวเตอร์ผ่านโปรแกรม Assembly แล้วนำไปจำลองด้วยโปรแกรม Simulator

ผลการทดสอบ

mcand	mplier	p	ความถูกต้อง	ความคาดหมาย	จำนวน step ที่ใช้	หมายเหตุ
32766	10383	340209378	ถูกต้อง	ถูกต้อง	125	
32766	-10383	-340209378	ถูกต้อง	ถูกต้อง	125	mplier ติดลบแต่ยังให้ผลลัพธ์ที่ถูกต้องอยู่
-7777	32766	1892531394	ผิด	ถูกต้อง	121	mcand ติดลบ
0	0	0	ถูกต้อง	ถูกต้อง	111	
57195	291	16643745	ถูกต้อง	ถูกต้อง	123	57195 เป็นเลขขนาด 15 บิต
5718	1	5718	ถูกต้อง	ถูกต้อง	118	
1	7290	7290	ถูกต้อง	ถูกต้อง	112	
90	90	8100	ถูกต้อง	ถูกต้อง	115	

ตารางผลการทดสอบโปรแกรมคูณ

สรุปผลการทดสอบ

จากการทดสอบอัลกอริทึมการคูณเลข จะพบได้ว่าทุก ๆ กรณีที่มีเงื่อนไขที่ถูกต้องจะสามารถทำงานได้ตามต้องการอยู่ ส่วนในกรณีที่ข้อมูลมีเงื่อนไขผิดก็อาจยังทำงานได้ผลลัพธ์ที่ถูกต้องบ้าง อย่างเช่น ในกรณีที่ mplier เป็นเลขติดลบ หรือ mcand เป็นเลขขนาดมากกว่า 16 นอกเหนือจากนี้โค้ดภาษาแอสเซมบลีที่ใช้ในการคำนวณอัลกอริทึมนี้ ทำงานอยู่ในช่วงเวลา 111 คำสั่ง ไปจนถึง 125 คำสั่ง

Recursive Combination Algorithm

```
Data: n, where n is a 32-bit integer.
   and r, where r is a 32-bit integer.
Result: c = (n r)
Function Cnr(n, r):
    if r = 0 ∨ n = r then
        c ← 1
    else
        c ← Cnr(n − 1, r) + Cnr(n − 1, r − 1)
    end
return
```

อัลกอริทึมหาจำนวนวิธีการจัดหมู่

โค้ดในภาพเป็นอัลกอริทึมการหาจำนวนวิธีจัดหมู่ โดยใช้โค้ดภาษาแอสเซมบลี ซึ่งโค้ดที่ว่า จะถูกในไปแปลงเป็นรหัสคอมพิวเตอร์ผ่านโปรแกรม Assembly แล้วนำไปจำลองด้วยโปรแกรม Simulator

ผลการทดสอบ

n	r	c	ความถูกต้อง	ความคาดหมาย	จำนวน step ที่ใช้	หมายเหตุ
7	3	35	ถูกต้อง	ถูกต้อง	909	
3	7	-	ผิด	ถูกต้อง	-	r ต่ำกว่า n การทำงานไม่สิ้นสุด
7	-3	-	ผิด	ถูกต้อง	-	r ติดลบ การทำงานไม่สิ้นสุด
0	0	1	ถูกต้อง	ถูกต้อง	10	
120	0	1	ถูกต้อง	ถูกต้อง	10	
1	1	1	ถูกต้อง	ถูกต้อง	11	
20	14	38760	ถูกต้อง	ถูกต้อง	1034876	
-2	-2	1	ผิด	ถูกต้อง	11	ทั้ง n และ r ติดลบ

ตารางผลการทดสอบหาจำนวนวิธีการจัดหมู่

สรุปผลการทดสอบ

จากการทดสอบหาจำนวนวิธีการจัดหมู่ ในกรณีนีปกติส่วนมากจะสามารถทำงานได้ตามที่ต้องการ แต่ในบางกรณีเราจะไม่สามารถระบุได้ว่าการทำงานจะเป็นไปถูกต้องหรือไม่ อย่างเช่นในกรณีที่ผลลัพธ์มีค่าเกิน 32 บิต หรือชุดข้อมูลที่ใช้เนื้อที่มากเกินไปการคำนวณ นอกจากนี้ในหลายกรณีเช่น r เป็นค่าติดลบ หรือ n มีค่าต่ำกว่า r จะส่งผลให้ตัวโปรแกรมทำงานไม่หยุด ซึ่งมีความอันตรายมาก แต่อย่างไรก็ตาม ขอบเขตของโปรเจคนี้ไม่ได้ต้องการให้คำนึงถึงความปลอดภัยของโปรแกรมภาษาแอสเซมบลี จึงถือว่าโปรแกรมที่เขียนมานี้ยังพอใช้ได้อยู่

Exponentiation Algorithm

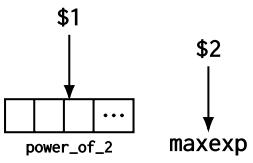
```
   Data: m, where m ∈ ℤ and −2^31 ≤ m ≤ 2^31 − 1;
      and e, where e ∈ ℤ and 0 ≤ e ≤ 32.
   Result: p = m^e
 1 if e = 0 then
 2 |  p ← 1
 3 |  return
 4 else if e < 0 then
 5 |  p ← 0
 6 |  return
 7 else if e > 32 then
 8 |  p ← 2^32 − 1
 9 |  return
10 end
11 maxexp ← 4
12 Let power_of_2[0 ... maxexp] be a new array
13 power_of_2[0] ← 1
14 i ← 1
15 while i ≠ maxexp do
16 |  x ← power_of_2[i]
17 |  power_of_2[i + 1] ← x + x
18 |  i ← i + 1
19 end
20 Let productTree[0 ... maxexp] be a new array
21 j ← maxexp
22 s ← 0
23 while e ≠ 0 do
24 |  t ← power_of_2[j]
25 |  if e ≤ t then
26 |  |  productTree[s] ← j)
27 |  |  e ← e − t
28 |  |  s ← s + 1
29 |  else
30 |  |  j ← j − 1
31 |  end
32 end
33 maxfactor ← productTree[0]
34 Let power_of_2^k[0 ... maxfactor] be a new array
35 power_of_2^n[0] ← m
36 k ← 0
37 while k ≠ maxfactor do
38 |  x ← power_of_2^k[i]
39 |  power_of_2^k[i + 1] ← x · x
40 end
41 p ← power_of_2^k[parseTree[0]]
42 h ← 1
43 while h ≠ s do
44 |  p ← p · power_of_2^k[parseTree[h]]
45 |  h ← h + 1
46 end
```

อัลกอริทึมยกกำลัง

โปรแกรมนี้จะทำงานด้วยการแบ่งเลขยกกำลังให้อยู่ในรูปของผลคูณ $m^e = \prod_i m^{2^{k_i}}$ โดยที่ $k_i \in \mathbb{N}$ ยกตัวอย่างเช่น $5^{13} = 5^8 \cdot 5^4 \cdot 5^1$ ในขั้นตอนแรกโปรแกรมจะแบ่งตัวชี้กำลัง ***e*** ให้อยู่ในรูปของ $\sum_i 2^{k_i}$ มาก่อน ซึ่งสามารถนำกลวิธี Greedy Strategy ในการคำนวณในส่วนนี้ได้ ขั้นตอนถัดไปจะใช้ Dynamic Programming ในการหารูปผลคูณ $m^{2^{k_i}}$ ทั้งหมดออกมา ทั้งนี้เนื่องจากทราบกันอยู่แล้วว่า $m^{2^{n+1}} = m^{2^n} \cdot m^{2^n}$ แสดงว่าในขั้นตอนที่กำลังจะคำนวณ $m^{2^{n+1}}$ อยู่นั้น โปรแกรมจะสามารถหาเลขนี้ได้จากการนำ m^{2^n} มาคูณกับตัวมันเอง แทนที่จะต้องคูณ m ทั้งหมด 2^{n+1} ครั้ง ในท้ายที่สุดโปรแกรมเรียกใช้ตัวเลขที่ได้จากขั้นตอนที่แล้วมาคูณด้วยกัน เพื่อให้ได้ผลลัพธ์เป็น $\prod_i m^{2^{k_i}}$ ในที่สุด

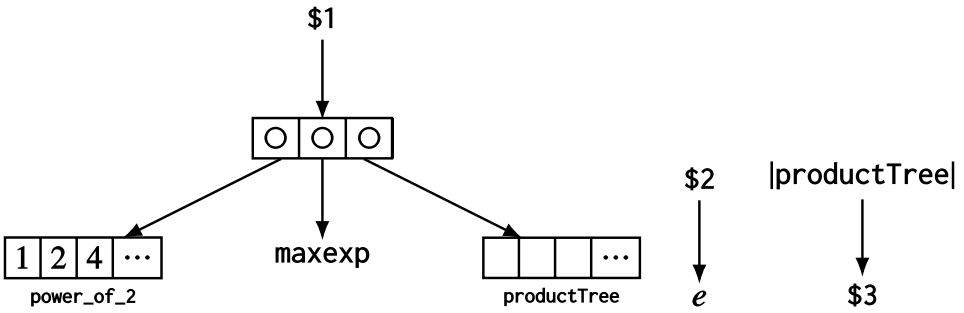
ในโค้ดภาษาแอสเซมบลีนั้น เราจะสามารภแบ่งแยกส่วนฟังก์ชันในออกมาเป็น 5 ส่วนใหญ่ดังนี้

- ***power*** มีหน้าที่ในการจัดสรรอาเรย์และตัวแปรทั้งหมด เตรียมจัดการส่งอินพุตสำหรับฟังก์ชัน แล้วเรียกใช้ฟังก์ชันตามอัลกอริทึม
- ***pow2f*** **(บรรทัด 12-19)** เขียนข้อมูลตัวเลขอาเรย์ ***power_of_2*** ให้อยู่ในรูป $2^i \mapsto \{1,2,4,8,16\}$ ซึ่งฟังก์ชั่นนี้จะรับ address ของ ***power_of_2*** บนเรจิสเตอร์ 1 ส่วนเรจิสเตอร์ที่ 2 จะเก็บค่าเลขชี้กำลังของ 2 ที่มากที่สุด (***maxexp***) ซึ่งในโปรแกรมนี้ตั้งค่าดังกล่าวให้เป็น 4 ไว้ เนื่องจากว่าถ้าหากพบค่า $e \geq 32$ ตัวโปรแกรมจะไม่สามารถคำนวณได้ถูกต้องแน่นอนจากการเกิดบิตที่เกินขอบเขต ดังนั้นไม่จำเป็นต้องพิจารณากรณีดังกล่าว นอกจากนี้ฟังก์ชัน ***pow2f*** จะไม่มีการรีเทิร์นค่าบนเรจิสเตอร์ 3 เนื่องจากได้ข้อมูลที่ต้องการทั้งหมดอยู่บนอาเรย์อยู่แล้ว



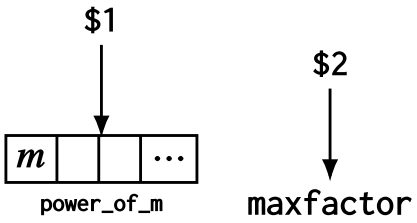
การรับอินพุตของฟังก์ชัน pow2f

- ***facf*** **(บรรทัด 23-32)** นำกลวิธี Greedy Strategy ในการแปลงค่า e ให้อยู่ในรูปของอนุกรม $\sum_i 2^{k_i}$ บนอาเรย์ ***productTree*** ยกตัวอย่างเช่น $13 \mapsto \{8,4,1\}$ ซึ่งเรจิสเตอร์ 1 จะส่งค่า address ของอาเรย์ขนาด 3 อาเรย์หนึ่ง โดยที่ตัวที่ 0 ของอาเรย์จะชี้ไปยังอาเรย์ ***power_of_2*** ตัวที่ 1 จะเก็บค่า ***maxexp*** ส่วนตัวที่ 2 จะชี้ไปยัง ***productTree*** สำหรับการบันทึกข้อมูล ส่วนเรจิสเตอร์ 2 จะเก็บค่า e ไว้ นอกจากนี้ ฟังก์ชัน ***facf*** จะรีเทิร์นขนาดของ ***productTree*** ไว้บนเรจิสเตอร์ 3 ไว้ในการคำนวณในขั้นตอนสุดท้าย



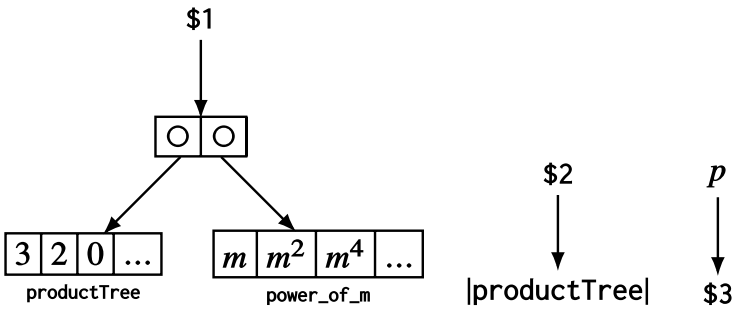
การรับอินพุตและการรีเทิร์นเอาต์พุตของฟังก์ชัน facf

- ***powmf*** **(บรรทัด 37-40)** เขียนข้อมูลตัวเลขอาเรย์ ***power_of_m*** ให้อยู่ในรูป m^{2^n} ยกตัวอย่างเช่น $5^{13} \mapsto 5^8 \mapsto \{5,25,625,390625\}$ โดยที่ค่า 2^n สุดท้ายของอาเรย์ดังกล่าว จะขึ้นอยู่กับค่าที่สูงสุดจาก ***productTree*** เพื่อลดการเรียกใช้ฟังก์ชันการคูณเกินความจำเป็น โดยที่จะเรียกตัวเลขนี้ว่า ***maxfactor*** ซึ่งฟังก์ชั่นนี้จะรับ address ของ ***power_of_m*** บนเรจิสเตอร์ 1 และที่ดัชรีแรกสุดของอาเรย์จะบันทึกค่า m มาให้ ส่วนเรจิสเตอร์ที่ 2 จะเก็บค่า ***maxfactor*** โดยที่ฟังก์ชันนี้จะไม่มีการรีเทิร์นค่าที่รีจิสเตอร์ 3



การรับอินพุตของฟังก์ชัน powmf

- ***prod*** **(บรรทัด 43-46)** จะหาผลโดยที่แต่ละค่าใน ***productTree*** จะถูกใช้เป็นค่าดัชนีบน ***power_of_m*** แล้วจะหาผลคูณของทุกค่าที่ได้มา โดยที่เรจิสเตอร์ 1 จะส่งค่า address ของอาเรย์ขนาด 2 อาเรย์หนึ่ง ตัวที่ 0 ของอาเรย์จะชี้ไปยังอาเรย์ ***productTree*** ส่วนตัวที่ 1 จะชี้ไปยัง ***power_of_m*** ส่วนเรจิสเตอร์ 2 ขนาดของ ***productTree*** ไว้ นอกจากนี้ ฟังก์ชัน ***prod*** จะรีเทิร์นค่าของผลคูณสุดท้าย ซึ่งจะเป็นผลลัพธ์การยกกำลังเช่นกัน



การรับอินพุตและการรีเทิร์นเอาต์พุตของฟังก์ชัน prod

ผลการทดสอบ

m	e	p	ความถูกต้อง	ความคาดหมาย	จำนวน step ที่ใช้	หมายเหตุ
5	13	1220703125	ถูกต้อง	ถูกต้อง	1623	
13	0	1	ถูกต้อง	ถูกต้อง	10	เลขชี้กำลังเป็น 0
13	57	2147483647	ผิด	ถูกต้อง	51	e มีค่ามากกว่า 32
3	32	-501334399	ผิด	ถูกต้อง	1421	คำตอบใช้จำนวนบิตเกินขอบเขต
3	-7	0	ผิด	ถูกต้อง	15	e มีค่าติดลบ
17	5	1419857	ถูกต้อง	ถูกต้อง	1110	
0	31	0	ถูกต้อง	ถูกต้อง	2345	
2	32	0	ผิด	ถูกต้อง	1402	32 บิตแรกของ 2^{32} คือ 0
91	1	91	ถูกต้อง	ถูกต้อง	389	

ตารางผลการทดสอบโปรแกรมยกกำลัง

สรุปผลการทดสอบ

จากการทดสอบอัลกอริทึมการยกกำลัง เงื่อนไขที่ไม่สมบูรณ์จะให้ค่าตามต้องการทั้งหมด ซึ่งได้แก่ $e = 0 \Rightarrow p = 1$ $e < 0 \Rightarrow p = 0$ และ $e > 32 \Rightarrow p = 2^{32} - 1$ ส่วนในกรณีที่เงื่อนไขถูกต้อง ก็สามารถคำนวณได้ตามที่ต้องการ โดยใช้ระยะเวลาในการประมวลผลในช่วง 380 ถึง 2400 คำสั่ง นอกจากนี้อัลกอริทึมที่ใช้ในการคำนวณดังกล่าวประมวลผลได้เร็วกว่าอัลกอริทึมยกกำลังแบบ brute-force (คูณ m ทั้งหมด e ครั้ง) ประมาณช่วง $e > 5$ จากที่ $e = 5$ จะใช้ระยะการทำงานประมาณ 1100 คำสั่ง และฟังก์ชันการคูณที่ใช้ในโปรแกรมนี้มีระยะการทำงานราว ๆ 200 คำสั่ง

สัดส่วนการทำงานและตารางเวลาทำงาน

สัดส่วนการทำงานจะแบ่งเป็น 4 ช่วงการทำงานดังนี้

ช่วงการทำงานที่ 1 คือการเขียนโปรแกรม Assembly และ โปรแกรม Simulator โดยใช้ภาษา Java ซึ่งมีการแบ่งหน้าที่ได้ดังนี้

1. **Simulator** (ณัฐชนน นันทศรี)
2. **Assembler** (ธนภัทร สมสิทธิ์)
3. **Stage** (เจตพล กอบกำ)
4. **Decoder** (ศรัณย์ กิมานุวัฒน์)
5. **RInstructions, IInstructions, JInstructions, OInstructions** (กษิดิ์ยศ หาญไพโรจน์)

Assignment Date: 17/9/2023

Deadline: 27/9/2023

Finish Date: 27/9/2023

GitHub: GitHub - Quercussi/Assembly_Interpreter

ช่วงการทำงานที่ 2 คือการทำ Multiplication Algorithm กับ Recursive Combination Algorithm ด้วยภาษาแอสเซมบลี ซึ่งมีการแบ่งหน้าที่ได้ดังนี้

1. **Multiplication Algorithm** (ธนภัทร สมสิทธิ์)
2. **Recursive Combination Algorithm** (กษิดิ์ยศ หาญไพโรจน์)

Assignment Date: 17/9/2023

Deadline: 1/10/2023

Finish Date: 27/9/2023

GitHub: GitHub - Quercussi/Assembly_Interpreter/sample_codes

ช่วงการทำงานที่ 3 คือการทำ Optimized Exponential Algorithm ด้วยภาษาแอสเซมบลี ซึ่งมีการแบ่งหน้าที่ได้ดังนี้

1. **main, power, mul, slt** (ธนภัทร สมสิทธิ์)
2. **pow2f** (เจตพล กอบกำ)
3. **factf** (กษิดิ์ยศ หาญไพโรจน์)
4. **powmf** (ณัฐชนน นันทศรี)
5. **prod** (ศรัณย์ กิมานุวัฒน์)

Assignment Date: 3/10/2023

Deadline: 8/10/2023

Finish Date: 5/10/2023

GitHub: GitHub - Quercussi/Assembly_Interpreter/sample_codes

ช่วงการทำงานที่ 4 คือการจัดทำรายงาน ซึ่งมีการแบ่งหน้าที่ได้ดังนี้

1. ผู้จัดทำรายงาน (เจตพล กอบกำ)
   (ณัฐชนน นันทศรี)
2. ผู้ตรวจสอบรายงาน (ธนภัทร สมสิทธิ์)
3. ผู้จัดเตรียมรูปภาพ (ธนภัทร สมสิทธิ์)

Assignment Date: 3/10/2023

Deadline: 15/10/2023

Finish Date: 9/10/2023

ช่วงการทำงานที่ 1 ช่วงการทำงานที่ 2 ช่วงการทำงานที่ 3 ช่วงการทำงานที่ 4

September

S	M	T	W	T	F	S
					01	02
03	04	05	06	07	08	09
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

October

S	M	T	W	T	F	S
01	02	03	04	05	06	07
08	09	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

ตารางเวลาทำงานตามที่วางแผน

ช่วงการทำงานที่ 1 ช่วงการทำงานที่ 2 ช่วงการทำงานที่ 3 ช่วงการทำงานที่ 4

September

S	M	T	W	T	F	S
					01	02
03	04	05	06	07	08	09
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

October

S	M	T	W	T	F	S
01	02	03	04	05	06	07
08	09	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

ตารางเวลาทำงานจริง

ภาคผนวก

Simulator.java

```
01    import java.io.*;
02
03    public class Simulator {
04
05        public static void main(String[] args) {
06            // Check whether there is an invalid number of arguments
07            if (args.length != 1) {
08                System.out.println("Usage: java Simulator <file_name>");
09                System.exit(1);
10            }
11
12            String fileName = args[0];
13            File inputFile = new File(fileName);
14
15            // Check whether the file exists.
16            boolean isExists = inputFile.exists();
17            if (!isExists) {
18                System.out.println("File '" + fileName + "' does not exist.");
19                System.exit(1);
20            }
21
22            // Construct a stage.
23            Stage stage = new Stage();
24            int[] memory = stage.getMemory();
25            int pc = 0;
26
27            // Extract machine code from the input file,
28            // then store it into the stage memory.
29            try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {
30                String line;
31                while ((line = br.readLine()) != null) {
32                    try {
33                        memory[pc] = Integer.parseInt(line.trim());
34                    } catch (NumberFormatException e) {
35                        System.out.println("Error: Cannot parse as an integer. Exception: " +
e.getMessage());
36                        System.exit(1);
37                    }
38                    pc++;
39                }
40                stage.setInstructionCount(pc);
41            } catch (IOException e) {
42                e.printStackTrace();
43            }
44
45            // Simulate the stage.
46            stage.simulate();
47            System.exit(0);
48        }
49    }
```

Stage.java

```
001    public class Stage {
002        private final int[] memory;       // Memory array
003        private final int[] register;     // Register array
004        private int pc;             // Program counter
005        private int nextPc;         // Next program counter (automatically set to Pc+1)
006        private boolean isHalt;     // Halt flag
007        private int stepCount;
008        private final Decoder decoder;
009        private int instructionCount;
010        private final StringBuilder sb = new StringBuilder();
011
012        /**
013         * Constructor for the stage class.
014         */
015        public Stage() {
016            memory = new int[65536];
017            register = new int[8];
018            pc = 0;
019            nextPc = 0;
020            isHalt = false;
021            stepCount = 0;
022            decoder = Decoder.getInstance();
023        }
024
025        /**
026         * Call the decoder to decode the stage,
027         * and setting the program counter to the next one.
028         */
029        public void iterate() {
030                nextPc = (pc + 1) % memory.length;
031                decoder.execute(this);
032                register[0] = 0;
033                pc = nextPc;
034                stepCount++;
035        }
036
037        /**
038         * Execute the stage until it halts.
039         */
040        public void simulate() {
041            while (!isHalt) {
042                printState();
043                iterate();
044            }
045
046            // print the last state.
047            System.out.println("machine halted\ntotal of " + stepCount + " instructions exe-
cuted");
048            System.out.println("final state of machine:\n");
049            printState();
050        }
051
052        /**
053         * Get the current instruction code
054         * @return the current instruction code.
055         */
056        public int getInstruction() {
057            // Ensure the program counter (pc) is within the memory bounds
058            if (pc >= 0 && pc < memory.length) {
059                return memory[pc];
060            } else {
061                // Handle an out-of-bounds access
062                return 0x1C00000; // Force halt
063            }
064        }
065
```

Stage.java (continued)

```
066         /**
067          * Get the memory array.
068          * @return the memory.
069          */
070         public int[] getMemory() {
071             return memory;
072         }
073 
074         /**
075          * Get the register array.
076          * @return the registers.
077          */
078         public int[] getRegister() {
079             return register;
080         }
081 
082         /**
083          * Get the current program counter.
084          * @return the current program counter
085          */
086         public int getPc(){
087             return pc ;
088         }
089 
090         /**
091          * Set the next program counter.
092          * @param newPc is the next program counter to be set.
093          */
094         public void setNextPc(int newPc) {
095             nextPc = newPc;
096         }
097 
098         /**
099          * Call the stage to halt.
100          */
101         public void setHalt() {
102             isHalt = true;
103         }
104 
105         /**
106          * Set the number of memory addresses that is to be printed.
107          * @param newInstructionCount is the updating instruction count.
108          */
109         public void setInstructionCount(int newInstructionCount) {
110             instructionCount = newInstructionCount;
111         }
112 
113         /**
114          * Print the details of the current state of the stage.
115          */
116         public void printState(){
117             sb.setLength(0); // clears the string builder
118 
119             sb.append("@@@\nstate:\n").append("\tpc ").append(pc).append('\n');
120 
121             sb.append("\tmemory:\n");
122             for(int i = 0; i < instructionCount; i++)
123                 sb.append("\t\tmem[").append(i).append("] ").append(memory[i]).append('\n');
124 
125             sb.append("\tregisters:\n");
126             for(int i = 0; i < register.length; i++)
127                 sb.append("\t\treg[").append(i).append("] ").append(register[i]).append('\n');
128 
129             sb.append("end state\n\n");
130             System.out.println(sb);
131         }
132     }
```

Decoder.java

```
01    public class Decoder {
02        private static Decoder instance;
03        private Decoder() {}
04
05        /**
06         * Return the singleton instance of the class
07         * @return is the instance of the class.
08         */
09        public static Decoder getInstance() {
10            if(instance == null)
11                instance = new Decoder();
12            return instance;
13        }
14
15        /**
16         * Retrieve the instruction from the stage,
17         * dissect it to extract a collection of fields,
18         * and then assign an instruction class to execute.
19         * @param stage is the Stage object that is to be computed.
20         */
21        public void execute(Stage stage) {
22            int instruction = stage.getInstruction();
23            int opcode = (instruction >> 22) & 0b111;
24
25            switch (opcode) {
26                // R-Type
27                case 0b000, 0b001 -> {
28                    int rs, rt, rd;
29                    rs = (instruction >> 19) & 0b111;
30                    rt = (instruction >> 16) & 0b111;
31                    rd = instruction & 0b111;
32                    RInstruction.getInstance().executeR(stage, opcode, rs, rt, rd);
33                }
34                // I-type
35                case 0b010, 0b011, 0b100 -> {
36                    int rs, rt, offset;
37                    rs = (instruction >> 19) & 0b111;
38                    rt = (instruction >> 16) & 0b111;
39                    offset = instruction & 0xFFFF;
40                    IInstruction.getInstance().executeI(stage, opcode, rs, rt, offset);
41                }
42                // J-Type
43                case 0b101 -> {
44                    int rs,rd;
45                    rs = (instruction >> 19) & 0b111;
46                    rd = (instruction >> 16) & 0b111;
47                    JInstruction.getInstance().executeJ(stage, opcode, rs, rd);
48                }
49                // O-type
50                case 0b110, 0b111 -> OInstruction.getInstance().executeO(stage, opcode);
51            }
52        }
53    }
```

RInstruction.java

```
01    public class RInstruction {
02
03        private static RInstruction instance ;
04
05        private RInstruction(){}
06
07        /**
08         * Return the singleton instance of the class
09         * @return is the instance of the class.
10         */
11        public static RInstruction getInstance(){
12            if(instance == null){
13                instance = new RInstruction();
14            }
15
16            return instance ;
17        }
18
19        /**
20         * Execute an R-type instruction for the input state.
21         * @param stage is the Stage object that is to be computed.
22         * @param opcode is operation code of the instruction.
23         * @param rs is the index of the primary source register.
24         * @param rt is the index of the secondary source register.
25         * @param rd is the index of the destination register.
26         */
27        public void executeR(Stage stage,int opcode,int rs,int rt,int rd){
28
29            int []reg = stage.getRegister() ;
30
31            if(rd == 0){
32                return; // does nothing.
33            }
34
35            if(opcode == 0){         // add instruction
36                reg[rd] = reg[rs] + reg[rt] ;
37            }else if(opcode == 1){  // nand instruction
38                reg[rd] = ~(reg[rs] & reg[rt]) ;
39            }
40        }
41    }
```

IInstruction.java

```
01    public class IInstruction {
02
03        private static IInstruction instance ;
04
05        private IInstruction(){}
06
07        /**
08         * Return the singleton instance of the class
09         * @return is the instance of the class.
10         */
11        public static IInstruction getInstance(){
12            if(instance == null){
13                instance = new IInstruction();
14            }
15
16            return instance ;
17        }
18
19        /**
20         * Execute an I-type instruction for the input state.
21         * @param stage is the Stage object that is to be computed.
22         * @param opcode is operation code of the instruction.
23         * @param rs is the index of the primary source register.
24         * @param rt is the index of the secondary source register.
25         * @param offset is the offset of the instruction.
26         */
27        public void executeI(Stage stage,int opcode,int rs,int rt,int offset){
28
29            int []reg = stage.getRegister() ;
30            int []mem = stage.getMemory() ;
31            int offset_field = sign_extend(offset) ;
32            int mem_address = offset_field + reg[rs] ;
33
34            if(opcode == 2){          // lw instruction
35                if(rt == 0){
36                    return ;
37                }
38                reg[rt] = mem[mem_address] ;
39
40            }else if(opcode == 3){    // sw instruction
41                mem[mem_address] = reg[rt] ;
42
43            }else if(opcode == 4){    // beq instruction
44                if(reg[rs] == reg[rt]){
45                    int newPc = stage.getPc() + 1 + offset_field ;
46                    stage.setNextPc(newPc) ;
47                }
48            }
49        }
50
51        /**
52         * Convert a 16-bit integer into a 32-bit integer.
53         * @param num is the converting integer.
54         * @return the converted integer.
55         */
56        private int sign_extend(int num){
57            if ((num & (1<<15)) != 0) {
58                num -= (1<<16);
59            }
60
61            return num;
62        }
63    }
```

JInstruction.java

```
01    public class JInstruction {
02
03        private static JInstruction instance ;
04
05        private JInstruction(){}
06
07        /**
08         * Return the singleton instance of the class
09         * @return is the instance of the class.
10         */
11        public static JInstruction getInstance(){
12            if(instance == null){
13                instance = new JInstruction();
14            }
15
16            return instance ;
17        }
18
19        /**
20         * Execute a J-type instruction for the input state.
21         * @param stage is the Stage object that is to be computed.
22         * @param opcode is operation code of the instruction.
23         * @param rs is the index of the source register.
24         * @param rd is the index of the destination register.
25         */
26        public void executeJ(Stage stage,int opcode,int rs,int rd){
27
28            int []reg = stage.getRegister() ;
29
30            if(opcode == 5){    // jalr instruction
31                if(rd != 0){
32                    reg[rd] = stage.getPc() + 1 ;
33                }
34                stage.setNextPc(reg[rs]) ;
35            }
36        }
37    }
```

OInstruction.java

```
01    public class OInstruction {
02
03        private static OInstruction instance;
04
05        private OInstruction(){}
06
07        /**
08         * Return the singleton instance of the class
09         * @return is the instance of the class.
10         */
11        public static OInstruction getInstance(){
12            if(instance == null){
13                instance = new OInstruction();
14            }
15
16            return instance ;
17        }
18
19        /**
20         * Execute an O-type instruction for the input state.
21         * @param stage is the Stage object that is to be computed.
22         * @param opcode is operation code of the instruction.
23         */
24        public void executeO(Stage stage,int opcode){
25            if(opcode == 6){        // halt instruction
26                stage.setHalt();
27            }else if(opcode == 7){ // noop instruction
28                // does nothing.
29            }
30        }
31    }
```

Assembler.java

```
001    import exceptions.*;
002
003    import java.io.File;
004    import java.io.FileWriter;
005    import java.nio.charset.Charset;
006    import java.nio.charset.StandardCharsets;
007    import java.nio.file.Path;
008    import java.nio.file.Files;
009    import java.nio.file.Paths;
010
011    import java.io.IOException;
012    import java.io.FileNotFoundException;
013    import java.nio.file.InvalidPathException;
014
015    import java.util.HashMap;
016    import java.util.List;
017    import java.util.Map;
018
019    public class Assembler {
020        public enum Operator {
021            ADD, NAND, LW, SW, BEQ, JALR, HALT, NOOP, FILL
022        }
023
024        // A map storing each label and its correlated address
025        static Map<String, Integer> labels = new HashMap<>();
026        // A list storing each lines in the assembly code.
027        static List<String> lines;
028
029        public static void main(String[] args) {
030            // Check whether there is an invalid number of arguments.
031            if(args.length != 2) {
032                System.out.println("only 2 arguments are required. <assembly-code-file>
<machine-code-file>");
033                System.exit(1);
034            }
035
036
037            File inputFile = new File("src\\" + args[0]);
038            File outputFile = new File("src\\" + args[1]);
039            try {
040                assemble(inputFile,outputFile); // Compile the code
041
042            } catch (FileNotFoundException | DuplicatedLabel | UnknownInstruction |
UndefinedLabel | OverflowingField | IllegalLabel e) {
043                // Unable to open the file.
044                System.out.println("error: " + e.getMessage());
045                System.exit(1);
046            }
047
048            System.exit(0);
049        }
050
051        /**
052         * Compile the assembly code from input file and the return out the machine code in
outputFile
053         * @param inputFile is the file storing input assembly code.
054         * @param outputFile is the file to which the output machine code will be stored.
055         */
056        public static void assemble(File inputFile, File outputFile) throws DuplicatedLabel,
FileNotFoundException,
057                UnknownInstruction, UndefinedLabel, OverflowingField, IllegalLabel {
058
```

Assembler.java (continued)

```
059             // Attempt to open the input file.
060             Charset charset = StandardCharsets.UTF_8;
061             Path inputPath;
062             try {
063                 inputPath = Paths.get(inputFile.getAbsolutePath()).normalize();
064             } catch (InvalidPathException e) {
065                 throw new FileNotFoundException(e.getMessage() + " cannot be converted to a
path.");
066             } catch (SecurityException e) {
067                 throw new FileNotFoundException(e.getMessage() + " cannot be accessed.");
068             }
069
070             // Get the assembly code from the input file.
071             String code;
072             try { code = Files.readString(inputPath, charset); }
073             catch (IOException e) { throw new FileNotFoundException(e.getMessage() + " cannot
be found."); }
074
075             lines = code.lines().toList();
076
077             // Setting up labels
078             int lineCount = 0;
079             for(String line : lines) {
080                 lineCount++;
081                 String label = line.split(tab)[0].strip();
082                 if(label.isBlank()) // Ignore lines without the label assignment.
083                     continue;
084
085                 // Check whether the label is not too long
086                 if(label.length() > 6)
087                     throw new IllegalLabel("illegal label \"" + label + "\" at line " +
lineCount + ".\n" +
088                             "Labels must be 6 characters or less.");
089
090                 // Check whether the label starts with a letter.
091                 if(!Character.isLetter(label.charAt(0)))
092                     throw new IllegalLabel("illegal label \"" + label + "\" at line " +
lineCount + ".\n" +
093                             "The first character in a label must be a letter.");
094
095                 // Check whether there are duplicated label assignments.
096                 if(labels.put(label, lineCount-1) != null)
097                     throw new DuplicatedLabel("duplicated label \"" + label + "\" at line " +
lineCount + ".");
098             }
099
100             // Creating machine-code
101             StringBuilder sb = new StringBuilder();
102             lineCount = 0;
103             for(String line : lines) {
104                 String[] tokens = line.split(tab);
105                 int tokensLength = Math.min(tokens.length, 5);
106                 lineCount++;
107
108                 // Check whether there is an operator at all.
109                 if(tokensLength <= 1)
110                     throw new UnknownInstruction("no operator found at line " + lineCount +
".");
111
```

Assembler.java (continued)

```
112                 int opcodeShift = 22;
113                 int field0Shift = 19;
114                 int field1Shift = 16;
115                 int field2Shift = 0;
116                 Operator operator = getOperator(tokens[1], lineCount);
117 
118                 // Store an O-type instruction.
119                 if(operator == Operator.HALT || operator == Operator.NOOP) {
120                     checkExcessTokens(tokens, 2, tokensLength, lineCount);
121                     sb.append(operator.ordinal() << opcodeShift).append('\n');
122                     continue;
123                 }
124 
125                 // Get the zeroth field.
126                 int i_field0 = variableInstance(tokens, 2, lineCount);
127 
128                 // Store a .fill assignment.
129                 if(operator == Operator.FILL) {
130                     checkExcessTokens(tokens, 3, tokensLength, lineCount);
131                     // Since .fill is for storing 32 bits, there is no need to check for
overflow.
132                     sb.append(i_field0).append('\n');
133                     continue;
134                 }
135 
136                 // Get the first field.
137                 int i_field1 = variableInstance(tokens, 3, lineCount);
138 
139                 // check for overflowing fields
140                 checkOverflow(i_field0,0,7,tokens[2],lineCount);
141                 checkOverflow(i_field1,0,7,tokens[3],lineCount);
142                 short field0 = (short) i_field0;
143                 short field1 = (short) i_field1;
144 
145                 // concatenate opcode, field0, and field1.
146                 int basicFields = (operator.ordinal()<<opcodeShift) | (field0<<field0Shift) |
(field1<<field1Shift);
147 
148                 // Store a J-type instruction.
149                 if(operator == Operator.JALR) {
150                     checkExcessTokens(tokens, 4, tokensLength, lineCount);
151                     sb.append(basicFields).append('\n');
152                     continue;
153                 }
154 
155                 // Store an I-type instruction.
156                 if(operator == Operator.LW || operator == Operator.SW || operator ==
Operator.BEQ) {
157                     int filter = 65535; // MAGIC!!!!
158 
159                     // Get the second field (16 bits).
160                     int field2 = variableInstance(tokens, 4, lineCount, (operator ==
Operator.BEQ));
161                     checkOverflow(field2,-32768,32767,tokens[4], lineCount);
162                     field2 &= filter;
163 
164                     sb.append(basicFields | (field2 << field2Shift)).append('\n');
165                     continue;
166                 }
167 
```

Assembler.java (continued)

```
168                 // R-type instructions
169                 if(operator == Operator.ADD || operator == Operator.NAND) {
170                     // Get the second field (3 bits).
171                     int i_field2 = variableInstance(tokens, 4, lineCount);
172                     checkOverflow(i_field2, 0, 7, tokens[4], lineCount);
173                     short field2 = (short) i_field2;
174 
175                     sb.append(basicFields | (field2 << field2Shift)).append('\n');
176                     continue;
177                 }
178 
179                 // Program should not reach here if opcode is decoded properly.
180                 throw new UnknownInstruction("unexpected error has occurred during
compilation.");
181             }
182 
183             // Remove the last \n
184             if(!sb.isEmpty() && sb.charAt(sb.length()-1) == '\n')
185                 sb.deleteCharAt(sb.length()-1);
186 
187             // Write machine code onto the output file.
188             try {
189                 FileWriter outputWriter = new FileWriter(outputFile, false);
190                 outputWriter.write(sb.toString());
191                 outputWriter.close();
192             } catch (IOException e) {
193                 throw new FileNotFoundException("unable to write " + e.getMessage() + ".");
194             }
195         }
196 
197         /**
198          * Get the value from a token and automatically check whether it is valid.
199          * @param tokens is the list of tokens within a line.
200          * @param index is the index of the token list that is to be evaluated.
201          * @param lineCount is the line number that the token list is extracted from.
202          * @param isLabelRelative isLabelRelative is used to specify whether the field
203          *                        should be differentiated with the line address.
204          *                        This argument is specifically used for the BEQ instruction.
205          * @return the variable at index if there are no exceptions.
206          * @throws UndefinedLabel if a label is called without any assignment.
207          * @throws UnknownInstruction if there are missing operands.
208          */
209         private static int variableInstance(String[] tokens, int index, int lineCount, boolean
isLabelRelative)
210                 throws UndefinedLabel, UnknownInstruction {
211             String str;
212 
213             // Missing operand (Array out of bound)
214             try {
215                 str = tokens[index].strip();
216             } catch (ArrayIndexOutOfBoundsException ignore) {
217                 throw new UnknownInstruction("missing operand at line " + lineCount + ".");
218             }
219 
220             // Missing operand (whitespace)
221             if(str.isBlank())
222                 throw new UnknownInstruction("missing operand at line " + lineCount + ".");
223 
224             // Parse integer
225             try {
226                 return Integer.parseInt(str);
227             } catch (NumberFormatException ignored) {}
228 
```

Assembler.java (continued)

```
229             // Get the label address
230             Integer parsedLabel = labels.get(str);
231             if (parsedLabel != null)
232                 return parsedLabel - (isLabelRelative ? lineCount : 0);
233 
234             // Undefined label
235             throw new UndefinedLabel("undefined label \"" + str + "\" at line " + lineCount +
"."); 
236         }
237 
238         /**
239          * Get the value from a token and automatically check whether it is valid
240          * for non-BEQ instructions.
241          * @param tokens is the list of tokens within a line.
242          * @param index is the index of the token list that is to be evaluated.
243          * @param lineCount is the line number that the token list is extracted from.
244          * @return the variable at index if there are no exceptions.
245          * @throws UndefinedLabel if a label is called without any assignment.
246          * @throws UnknownInstruction if there are missing operands.
247          */
248         private static int variableInstance(String[] tokens, int index, int lineCount) throws
UndefinedLabel, UnknownInstruction {
249             return variableInstance(tokens,index,lineCount,false);
250         }
251 
252         /**
253          * Get the operator as an enumerated object
254          * @param str is the input operator as string
255          * @param lineCount is the line number that the token list is extracted from.
256          * @return the operator as an enumerated object.
257          * @throws UnknownInstruction if the input string is not a valid operator.
258          */
259         private static Operator getOperator(String str, int lineCount) throws
UnknownInstruction {
260             String strOperator = str.strip().toUpperCase();
261 
262             // Check whether there is an operator.
263             if(strOperator.isBlank())
264                 throw new UnknownInstruction("no operator found at line " + lineCount + ".");
265 
266             if(strOperator.equals(".FILL"))
267                 return Operator.FILL;
268 
269             try {
270                 return Operator.valueOf(strOperator);
271             } catch (IllegalArgumentException ignore) {
272                 throw new UnknownInstruction("unknown operator \"" + strOperator + "\" at line
" + lineCount + ".");
273             }
274         }
275 
```

Assembler.java (continued)

```
276         /**
277          * Check whether there are any tokens within the given range.
278          * @param tokens is the list of tokens within a line.
279          * @param beginIndex is the first index that will be checked.
280          * @param endIndex is the last index that will be checked.
281          * @param lineCount is the line number that the token list is extracted from.
282          * @throws UnknownInstruction if there is an excess field.
283          */
284         private static void checkExcessTokens(String[] tokens, int beginIndex, int endIndex,
int lineCount) throws UnknownInstruction {
285             for(int i = beginIndex; i < endIndex; i++)
286                 try {
287                     if (!tokens[i].isBlank())
288                         throw new UnknownInstruction("excess operand " + tokens[i] + " at line
" + lineCount + ".");
289                 } catch (ArrayIndexOutOfBoundsException ignore) { return; }
290         }
291 
292         /**
293          * Check whether the input number is within the input range.
294          * @param number is the input number that will be checked.
295          * @param min is the lower bound of the input range.
296          * @param max is the upper bound of the input range.
297          * @param token is the additional string for exception message.
298          * @param lineCount is the line number that the token is extracted from.
299          * @throws OverflowingField if the input number is out of range.
300          */
301         private static void checkOverflow(int number, int min, int max, String token, int
lineCount) throws OverflowingField {
302             if(number < min || number > max)
303                 throw new OverflowingField("overflowing field0 \"" + token + "\" at line " +
lineCount + "\n" +
304                         "The required field must be in between " + min + " and " + max + ".");
305         }
306 
307         private static final String tab = "\t";
308     }
```

Multiplier16pos.txt

```
x0      beq    0      0      main   ; register 0
a1      noop                        ; function argument 1
a2      noop                        ; function argument 2
a0      noop                        ; return value
t0      noop                        ; temp 0
sp      noop                        ; stack pointer
t1      noop                        ; temp 1
ra      noop                        ; return address
pos1    .fill  1
neg1    .fill  -1
c65536  .fill  65536
mulAdr  .fill  mul
mcand   .fill  32766
mplier  .fill  10383
main    lw     0      sp     neg1
        lw     0      a1     mcand  ; assign a1
        lw     0      a2     mplier ; assign a2
        lw     0      t0     mulAdr
        jalr   t0     ra             ; use mul function
        halt
mul     sw     0      sp     sp     ; (5)* = sp
        sw     0      ra     ra     ; (7)* = ra
        add    x0     x0     a0     ; sum = 0
        lw     0      t1     neg1   ; set t1 to -1
        lw     0      sp     c65536 ; set sp to 2^16
        lw     0      t0     pos1   ; for t0=1,2,4,...,2^{15}
muladd  nand   a1     t0     ra
        beq    t1     ra     skAdd  ; if ra=-1, jump to skAdd+1
        add    a2     a0     a0     ; sum += a2
skAdd   add    t0     t0     t0     ; t0<<=1
        add    a2     a2     a2     ; a2<<=1
        beq    sp     t0     mulret ; if t0=2^16, jump to mulret+1
        beq    0      0      muladd ; else jump to muladd+1
mulret  lw     0      ra     ra     ; (7)* = ra
        lw     0      sp     sp     ; (5)* = sp
        jalr   ra     0              ; jump back to return address
```

Multiplier16pos.mc

```
16777229
29360128
29360128
29360128
29360128
29360128
29360128
29360128
1
-1
65536
20
32766
10383
8716297
8454156
8519693
8650763
23527424
25165824
12910597
13041671
3
8781833
8716298
8650760
4980743
20381697
1245187
2359300
1179650
19660801
16842745
8847367
8716293
24641536
```

Cnr.txt

```
x0      beq    0      0      main
a1      noop                        ; function argument 1
a2      noop                        ; function argument 2
a0      noop                        ; return value
t0      noop                        ; temp 0
sp      noop                        ; stack pointer
t1      noop                        ; temp 1
ra      noop                        ; return address
pos1    .fill  1
pos4    .fill  4                    ; test
neg4    .fill  -4
cnrAdr  .fill  cnr
n       .fill  7
r       .fill  3
main    lw     0      sp     spAddr ; sp = stack
        lw     0      a1     n      ; set a1
        lw     0      a2     r      ; set a2
        lw     0      t0     cnrAdr
        jalr   t0     ra             ; use Cnr function
        halt
cnr     beq    0      a2     basCnr
        beq    a1     a2     basCnr
        lw     0      t0     cnrAdr ; t0 ALWAYS stays cnrAdr
        lw     0      t1     pos4
        add    t1     sp     sp     ; sp += 4
        sw     sp     ra     -4     ; (sp-4)* = ra
        sw     sp     a2     -3     ; (sp-3)* = r
        sw     sp     a1     -2     ; (sp-2)* = n
        nand   0      0      t1
        add    t1     a1     a1     ; n -= 1
        jalr   t0     ra             ; use Cnr
        sw     sp     a0     -1     ; (sp-1)* = cnr(n-1,r)
        nand   0      0      t1
        add    t1     a2     a2     ; r -= 1
        jalr   t0     ra             ; recursively use Cnr
        lw     sp     a1     -1     ; cnr(n-1,r) = (sp-1)*
        add    a1     a0     a0     ; return cnr(n-1,r) + cnr(n-1,r-1)
        lw     sp     a1     -2     ; n = (sp+2)*
        lw     sp     a2     -3     ; r = (sp+3)*
        lw     sp     ra     -4     ; ra = (sp+4)*
        lw     0      t1     neg4
        add    t1     sp     sp     ; sp -= 4
        jalr   ra     0              ; return
basCnr  lw     0      a0     pos1   ; set output to 1
        jalr   ra     0              ; return
stack   noop
spAddr  .fill  stack
```

Cnr.mc

```
16777229
29360128
29360128
29360128
29360128
29360128
29360128
29360128
1
4
-4
20
7
3
8716334
8454156
8519693
8650763
23527424
25165824
16908310
17432597
8650763
8781833
3473413
15728636
15400957
15335422
4194310
3211265
23527424
15466495
4194310
3276802
23527424
11141119
720899
11141118
11206653
11534332
8781834
3473413
24641536
8585224
24641536
29360128
45
```

power.txt

```
x0     beq    0      0      main
a1     noop                        ; function argument 1
a2     noop                        ; function argument 2
a0     noop                        ; return value
t0     noop                        ; temp 0
sp     noop                        ; stack pointer
t1     noop                        ; temp 1
ra     noop                        ; return address
m      .fill  5                   ; input base
e      .fill  13                  ; input power
main   lw     0      sp     spAddr ; sp = stack
       lw     0      a1     m      ; set a1
       lw     0      a2     e      ; set a2
       lw     0      t0     powerA ; getting function address
       jalr   t0     ra             ; call the function
       halt
powerA .fill  power
power  beq    0      a2     1      ; if e == 0 -> return 1
       beq    0      0      2      ; jump to next if
       lw     0      a0     c1
       jalr   ra     0
       lw     0      t1     sign   ; set t1 to sign
       nand   a2     t1     a0     ; check last bit of e
       nand   a0     a0     a0
       beq    a0     0      2      ; if e < 0 -> return 0
       add    0      0      a0
       jalr   ra     0
       lw     0      t1     c2     ; set t1 to 2
       add    t1     sp     sp     ; increment stack by 2
       sw     sp     a1     -2     ; temporarily save a1 at stack
       sw     sp     ra     -1     ; temporarily save ra at stack
       lw     0      a1     c32    ; set a1 to 32
       lw     0      t1     sltA
       jalr   t1     ra
       lw     sp     ra     -1     ; load ra back
       lw     sp     a1     -2     ; load a1 back
       lw     0      t1     n2     ; set t1 to 2
       add    t1     sp     sp     ; decrement stack by 2
       beq    a0     0      2      ; if e > 32 -> return INT_MAX
       lw     0      a0     INTMAX
       jalr   ra     0
       noop                        ____UNDESIREABLES HANDLING____
       lw     0      t1     c4     ; store 4 at $6
       add    sp     t1     sp     ; increment stack by 4
       sw     sp     a1     -4     ; store $1 (input arg 1)
       sw     sp     a2     -3     ; store $2 (input arg 2)
       sw     sp     t0     -2     ; store $4 (local variable)
       sw     sp     ra     -1     ; store $7 (return address)
       noop                        ____START FUNCITON____
       add    0      sp     t0     ; set t0 to be varible address
       lw     0      t1     c4     ; set t1 to 4
       add    t1     sp     sp     ; allocate 4 address for variables
       sw     t0     t1     0      ; set maxexp to 4 at t0
       sw     t0     sp     2      ; allocate productTree in advance at t0 + 2
       lw     0      ra     c1     ; set ra to 1
       add    ra     t1     t1     ; set t1 = maxexp + 1
       add    t1     sp     sp     ; increment sp by maxexp + 1 for productTree
       noop                        ____END OF VARIABLE DECLARATION____
```

power.txt (continued)

```
        sw      t0      sp      3       ; allocate power_of_2 at t0 + 3
        lw      t0      a1      3       ; set arg1 to power_of_2
        lw      t0      a2      0       ; set arg2 to maxexp
        add     t1      sp      sp      ; increment sp by maxexp + 1 for power_of_2
        lw      0       t1      pow2fA  ; getting function address
        jalr    t1      ra              ; call the function
        noop                            ____END OF PROCEDURE 1____
        add     0       sp      ra      ; temporarily set ra to current sp
        lw      0       t1      c3      ; set t1 to 3
        add     t1      sp      sp      ; increment sp by 3 for input args
        sw      sp      a1      -3      ; set (sp-3)* to be power_of_2
        sw      sp      a2      -2      ; set (sp-2)* to be maxexp
        lw      0       t1      c2      ; set t1 to 2
        lw      t0      t1      2       ; set t1 to productTree
        sw      sp      t1      -1      ; set (sp-1)* to be productTree
        add     0       ra      a1      ; set a1 to be the address of input args
        lw      t0      a2      -3      ; set a2 to e
        lw      0       t1      facfA   ; calling function address
        jalr    t1      ra              ; call the function
        sw      t0      a0      1       ; set productTreeSize at t0 + 1
        lw      0       t1      n3      ; set t1 to -3
        add     t1      sp      sp      ; deallocate the input args
        noop                            ____END OF PROCEDURE 2_____
        lw      t0      a2      2       ; set a2 to productTree
        lw      a2      a2      0       ; set a2 to maxfactor
        lw      t0      a1      3       ; set a1 to be power_of_m (We'll reuse the space at
power_of_2)
        lw      t0      t1      -4      ; set t1 to m
        sw      a1      t1      0       ; store m at the first index of power_of_m
        lw      0       t1      powmfA  ; getting funcition address
        jalr    t1      ra              ; call the function
        noop                            ____END OF PROCEDURE 3_____
        add     0       sp      ra      ; temporarily set ra to current sp
        lw      0       t1      c2      ; set t1 to 2
        add     t1      sp      sp      ; increment sp by 2 for input args
        lw      t0      t1      2       ; set t1 to productTree
        sw      sp      t1      -2      ; set (sp-2)* to be productTree
        lw      t0      t1      3       ; set t1 to power_of_m
        sw      sp      t1      -1      ; set (sp-1)* to be power_of_m
        add     0       ra      a1      ; set a1 to be the address of input args
        lw      t0      a2      1       ; set a2 to be productTreeSize
        lw      0       t1      prodA   ; getting the fuction address
        jalr    t1      ra              ; call the function address
        lw      0       t1      n2      ; set t1 to -2
        add     t1      sp      sp      ; deallocate the input args
        noop                            ____END OF PROCEDURE 4____
        lw      t0      ra      0       ; set ra to be maxexp          [deallocate
power_of_m]
        nand    ra      ra      t1      ; set t1 to be -(maxexp + 1)
        add     t1      t1      t1      ; set t1 to be -2(maxexp + 1)  [deallocate
productTree]
        lw      0       ra      n4      ; set t1 to -4               [deallocate variables]
        add     ra      t1      t1      ; set t1 to be -2(maxexp + 1) - 4
        add     t1      sp      sp      ; deallocate stack
        noop                            ____END OF DEALLOCATION____
        lw      sp      ra      -1      ; store ra (return address)
        lw      sp      t0      -2      ; store t0 (local variable)
        lw      sp      a2      -3      ; store a2 (input arg 2)
```

power.txt (continued)

```
       lw     sp     a1     -4     ; store a1 (input arg 1)
       lw     0      t1     n4     ; store -4 at t1
       add    sp     t1     sp     ; decrement stack by 4
       jalr   ra     0             ; return
       noop                        ##############################
       noop                        ##############################
facfA  .fill  facf
facf   lw     0      t1     c5
       add    sp     t1     sp     increment stack by 5
       sw     sp     0      -5
       sw     sp     1      -4     store $1 (input arg 1)
       sw     sp     2      -3     store $2 (input arg 2)
       sw     sp     4      -2     store $4 (local variable)
       sw     sp     7      -1     store $7 (return address)
       lw     a1     t0     1      i = maxexp
faclp          lw     sp     a1     -4     ; set a1 to input array
       beq    a2     0      facext ; if e == 0, return
       lw     0      t1     n1     ; set t1 to -1
       lw     a1     a1     0      ; set a1 to power_of_2
       add    t0     a1     a1     ; set a1 to power_of_2 + i
       lw     a1     a1     0      ; a1 = power_of_2[i]
       beq    a1     a2     3      ; if factor == e, jump
       lw     0      t1     sltA
       jalr   t1     ra
       beq    0      a0     facdnc ; if factor >= e, continue
       lw     0      t1     c1     t1 = 1
       nand   a1     a1     a1             set a1 to complement
       add    t1     a1     a1             a1 = a1 + 1
       add    a1     a2     a2             e  = e - factor
       lw     sp     a1     -4
       lw     a1     a1     2      address productTree
       lw     sp     ra     -5     get j
       add    ra     a1     a1     a1=productTree+j
       sw     a1     t0     0              productTree[j] = t0 = i
       add    t1     ra     ra             j += 1
       sw     sp     ra     -5
       beq    0      0      faclp          go back to the beginning of loop
facdnc         lw     0      t1     n1     t1 = -1
       add    t1     t0     t0     i -= 1
       beq    0      0      faclp          go back to the beginning of loop
facext          lw     sp     a0     -5
                lw     sp     7      -1     store $7 (return address)
                lw     sp     4      -2     store $4 (local variable)
                lw     sp     2      -3     store $2 (input arg 2)
                lw     sp     1      -4     store $1 (input arg 1)
                lw     0      t1     n5     store -5 at $6
                add    t1     sp     sp     decrement stack by 5
                jalr   ra     0             return
       noop                        ##############################
       noop                        ##############################
prodA  .fill  prod
prod   lw     0      t1     c4     ; store 4 at $6
       add    sp     t1     sp     ; increment stack by 4
       sw     sp     a1     -4     ; store $1 (input arg 1)
       sw     sp     a2     -3     ; store $2 (input arg 2)
       sw     sp     t0     -2     ; store $4 (local variable)
```

power.txt (continued)

```
        sw      sp      ra      -1      ; store $7 (return address)
        lw      0       t0      c1      ; set t0 to 1 (i)
        lw      a1      t1      0       ; set t1 to power_of_m
        lw      t1      t1      0       ; set t1 to power_of_m[0]
        lw      a1      ra      1       ; set ra to productTree
        add     ra      t1      ra      ; set ra to productTree + power_of_m[0]
        lw      ra      a0      0       ; set a0 to productTree[power_of_m[0]]
prodlp          beq     t0      a2      prodxt ; Loop until i = productTreeSize
        lw      a1      t1      0       ; set t1 to power_of_m
        add     t1      t0      t1      ; set t1 to power_of_m + i
        lw      t1      t1      0       ; set t1 to power_of_m[i]
        lw      a1      ra      1       ; set ra to productTree
        add     ra      t1      ra      ; set ra to productTree + power_of_m[i]
        lw      ra      a2      0       ; set a2 to be peoductTree[power_of_m[i]]
        add     a0      0       a1      ; set a1 to current product
        lw      0       t1      mulA    ; getting multipler function address
        jalr    t1      ra              ; call the function
        lw      0       t1      c1      ; set t1 to 1
        add     t0      t1      t0      ; i = i + 1
        lw      sp      a1      -4      ; load a1 (input arg 1)
        lw      sp      a2      -3      ; load a2 (input arg 2)
        beq     0       0       prodlp ; loop
prodxt          lw      sp      ra      -1      ; store $7 (return address)
        lw      sp      t0      -2      ; store $4 (local variable)
        lw      sp      a2      -3      ; store $2 (input arg 2)
        lw      sp      a1      -4      ; store $1 (input arg 1)
        lw      0       t1      n4      ; store -4 at $6
        add     sp      t1      sp      ; decrement stack by 4
        jalr    ra      0               ; return
        noop                            ################################
        noop                            ################################
pow2fA .fill    pow2f
pow2f   lw      0       t1      c4      ; store 4 at $6
        add     sp      t1      sp      ; increment stack by 4
        sw      sp      a1      -4      ; store $1 (input arg 1)
        sw      sp      a2      -3      ; store $2 (input arg 2)
        sw      sp      t0      -2      ; store $4 (local variable)
        sw      sp      ra      -1      ; store $7 (return address)
        lw      0       t0      c1      ; set t0 = 1
        sw      a1      t0      0       ; power_of_2[0] = 1
        add     0       0       t0      ; i = 0
pow2lp beq      t0      a2      pow2xt ; if i == maxexp, exit the loop
        add     a1      t0      ra      ; power_of_2[i]
        lw      ra      t1      0       ; t1 = power_of_2[i]
        add     t1      t1      t1      ; curr + curr
        sw      ra      t1      1       ; power_of_2[i+1] = curr + curr
        lw      0       t1      c1      ; t1 = 1
        add     t1      t0      t0      ; i += 1
        beq     0       0       pow2lp ; go tp loop
pow2xt  lw      sp      ra      -1      ; store $7 (return address)
        lw      sp      t0      -2      ; store $4 (local variable)
        lw      sp      a2      -3      ; store $2 (input arg 2)
        lw      sp      a1      -4      ; store $1 (input arg 1)
        lw      0       t1      n4      ; store -4 at $6
        add     sp      t1      sp      ; decrement stack by 4
        jalr    ra      0               ; return
```

power.txt (continued)

```
       noop                        ##############################
       noop                        ##############################
powmfA .fill  powmf
powmf          lw     0      t1     c4     ; store 4 at $6
       add     sp     t1     sp     ; increment stack by 4
       sw      sp     a1     -4     ; store $1 (input arg 1)
       sw      sp     a2     -3     ; store $2 (input arg 2)
       sw      sp     t0     -2     ; store $4 (local variable)
       sw      sp     ra     -1     ; store $7 (return address)
       add     0      0      t0     ; i = 0
powmlp         beq    t0     a2     powmxt ; if i == maxfactor , exit
       add     a1     t0     ra     ; ra = power_of_m + i
       lw      ra     t1     0      ; t1 = power_of_m[i]
       add     t1     0      a1     ; set a1 to curr
       add     t1     0      a2     ; set a2 to curr
       lw      0      t1     mulA   ; getting multipler function address
       jalr    t1     ra            ; call the function
       lw      sp     a1     -4     ; load a1 (input arg 1)
       lw      sp     a2     -3     ; load a2 (input arg 2)
       add     a1     t0     ra     ; ra = power_of_m + i
       sw      ra     a0     1      ; power_of_m[i+1] = curr * curr
       lw      0      t1     c1     ; t1 = 1
       add     t0     t1     t0     ; t0 + 1
       beq     0      0      powmlp ; back to loop
powmxt         lw     sp     ra     -1     ; store $7 (return address)
       lw      sp     t0     -2     ; store $4 (local variable)
       lw      sp     a2     -3     ; store $2 (input arg 2)
       lw      sp     a1     -4     ; store $1 (input arg 1)
       lw      0      t1     n4     ; store -4 at $6
       add     sp     t1     sp     ; decrement stack by 4
       jalr    ra     0             ; return
       noop                        ##############################
       noop                        ##############################
mulA   .fill  mul
mul    lw      0      t1     c4     ; store 4 at $6
       add     sp     t1     sp     ; increment stack by 4
       sw      sp     a1     -4     ; store $1 (input arg 1)
       sw      sp     a2     -3     ; store $2 (input arg 2)
       sw      sp     t0     -2     ; store $4 (local variable)
       sw      sp     ra     -1     ; store $7 (return address)
       add     x0     x0     a0     ; sum = 0
       lw      0      t1     n1     ; set t1 to -1
       lw      0      t0     c1     ; for t0=1,2,4,...,2^{31}
muladd nand    a1     t0     ra     ; check if current bit is 1
       beq     t1     ra     1      ; if ra=-1, skip addition
       add     a2     a0     a0     ; sum += a2
       add     t0     t0     t0     ; t0<<=1
       add     a2     a2     a2     ; a2<<=1
       beq     0      t0     mulret ; if t0=0, jump to mulret
       beq     0      0      muladd ; else jump to muladd
mulret lw      sp     ra     -1     ; store $7 (return address)
       lw      sp     t0     -2     ; store $4 (local variable)
       lw      sp     a2     -3     ; store $2 (input arg 2)
       lw      sp     a1     -4     ; store $1 (input arg 1)
       lw      0      t1     n4     ; store -4 at $6
       add     sp     t1     sp     ; decrement stack by 4
```

power.txt (continued)

```
       jalr   ra     0            ; return
       noop                       ##############################
       noop                       ##############################
sltA   .fill  slt
slt    lw     0      t1     c4     ; store 4 at $6
       add    t1     sp     sp     ; increment stack by 4
       sw     sp     a1     -4     ; store $1 (input arg 1)
       sw     sp     a2     -3     ; store $2 (input arg 2)
       sw     sp     t0     -2     ; store $4 (local variable)
       sw     sp     ra     -1     ; store $7 (return address)
       noop                       ____START FUNCTION____
       lw     0      t1     c1     ; set t1 to 1
       nand   a2     a2     a2     ; set a2 to its complement
       add    t1     a2     a2     ; get -a2
       add    a1     a2     a0     ; substract the two input args (a0 = a1 - a2)
       lw     0      t1     sign          ; set t1 to be the filter
       nand   t1     a0     a0     ; find if the last bit of a0 is 0.
       nand   a0     a0     a0
       beq    0      a0     1      ; if a1 - a2 >= 0, return promtly
       lw     0      a0     c1     ; set a0 to 1
       noop                       ____END FUNCTION____
       lw     sp     ra     -1     ; store $7 (return address)
       lw     sp     t0     -2     ; store $4 (local variable)
       lw     sp     a2     -3     ; store $2 (input arg 2)
       lw     sp     a1     -4     ; store $1 (input arg 1)
       lw     0      t1     n4     ; store -4 at $6
       add    t1     sp     sp     ; decrement stack by 4
       jalr   ra     0            ; return
c1     .fill  1
c2     .fill  2
c3     .fill  3
c4     .fill  4
c5     .fill  5
c32    .fill  32
INTMAX .fill  2147483647
n1     .fill  -1
n2     .fill  -2
n3     .fill  -3
n4     .fill  -4
n5     .fill  -5
sign   .fill  -2147483648         ; 1<<31, the last bit
spAddr .fill  stack
stack  noop
```

power.mc

```
16777225    29360128    8782143     9306112     8782136     8782136
29360128    15007747    3014661     11927552    3014661     3473413
29360128    10551299    24641536    9371649     15335420    15335420
29360128    10616832    29360128    4063239     15400957    15400957
29360128    3473413     29360128    12255232    15532030    15532030
29360128    8782024     120         19005454    15728639    15728639
29360128    24576000    8782137     9306112     4           29360128
29360128    29360128    3014661     3407878     19005453    8782133
5           327687      15269883    11927552    786439      5373954
13          8782135     15335420    9371649     12451840    3276802
8716610     3473413     15400957    4063239     3145729     655363
8454152     15335421    15532030    12189696    3145730     8782145
8519689     15400958    15728639    1572865     8782082     7536643
8650768     8782134     9175041     8782082     24576000    5963779
23527424    10878978    11141116    24576000    11141116    16973825
25165824    15663103    17825815    8782133     11206653    8585525
17          458753      8782140     2490372     786439      29360128
16908289    10682365    8978432     11141116    16449537    11534335
16777218    8781943     2162689     11206653    8782133     11337726
8585525     24576000    8978432     16842737    2490372     11206653
24641536    14876673    17432579    11534335    16842738    11141116
8782145     8782142     8782108     11337726    11534335    8782143
5636099     3473413     24576000    11206653    11337726    3473413
5963779     29360128    16973836    11141116    11206653    24641536
18350082    10616834    8782133     8782143     11141116    1
3           9568256     4784129     3014661     8782143     2
24641536    10551299    3211265     24641536    3014661     3
8782134     10944508    655362      29360128    24641536    4
3473413     13500416    11141116    29360128    29360128    5
15335422    8782051     8978434     201         29360128    32
15728639    24576000    11534331    8782136     259         2147483647
8454458     29360128    3735553     3014661     8782136     -1
8782108     327687      13369344    15335420    3014661     -2
24576000    8782134     3604487     15400957    15335420    -3
11534335    3473413     15728635    15532030    15400957    -4
11141118    10878978    16842730    15728639    15532030    -5
8782141     15663102    8782140     8651061     15728639    -2147483648
3473413     10878979    3407876     13369344    3           323
18350082    15663103    16842727    4           8782140     29360128
8585531     458753      11272187    19005447    8651061
24641536    10616833    11534335    786439      4980743
29360128    8781987     11337726    12451840    20381697
8782136     24576000    11206653    3538950     1245187
3014661     8782141     11141116    16646145    2359300
15335420    3473413     8782144     8782133     1179650
15400957    29360128    3473413     3407876     17039361
15532030    10944512    24641536    16842744    16842745
15728639    8323078     29360128    11534335    11534335
29360128    3538950     29360128    11337726    11337726
327684      8847679     164         11206653    11206653
8782136     4063238     8782136     11141116    11141116
3473413     3473413     3014661     8782143     8782143
15073280    29360128    15335420    3014661     3014661
15007746    11534335    15400957    24641536    24641536
8847669     11337726    15532030    29360128    29360128
4063238     11206653    15728639    29360128    29360128
3473413     11141116    8651061     228         285
```

(หน้าปกหลัง)