作業資訊： EE3450 計算機結構 Final Project
學生資訊： 郭柏辰 107012045
作業內容：

一、Use Euclid's Algorithm to solve GCD via recursive method

1. Approach

先描述本小題會使用到基於 Euclid's Algorithm 的 2 個事實如下，題目假定 a、b 為兩正整數。

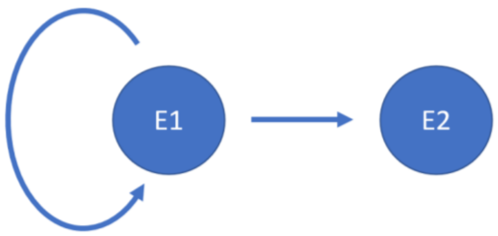
$$(E1) If\ a \neq b, say, a > b, then\ gcd(a,b) = gcd(a-b,b).$$
$$(E2)\ If\ a = b, then\ gcd(a,b) = a.$$

根據上述兩條件，我們若用 recursive 方法完成的話，我們先判斷是否 E2 條件成立，因為 E2 即為此演算法的中止條件，若成立則直接 return 數值 a。若 E2 不成立，則必為 E1 條件，比較 a、b 大小，若 a＞b 則 return $gcd(a-b,b)$的數值，即遞迴呼叫另一個 function。

另外，使用 assembly code 完成的部分，其邏輯與 c code 相同。先比較傳入的 argument a, b 的值是否相同，相同則回傳 a，不同則進行比較 a, b 的值，使用'slt' 指令來完成，而由於要遞迴呼叫 function，因此要先將 return address 存入 stack 中再進行呼叫。

2. Results and Discussion

若我們先分析此演算法會進行的步驟，以 State diagram 表示，如下圖一。



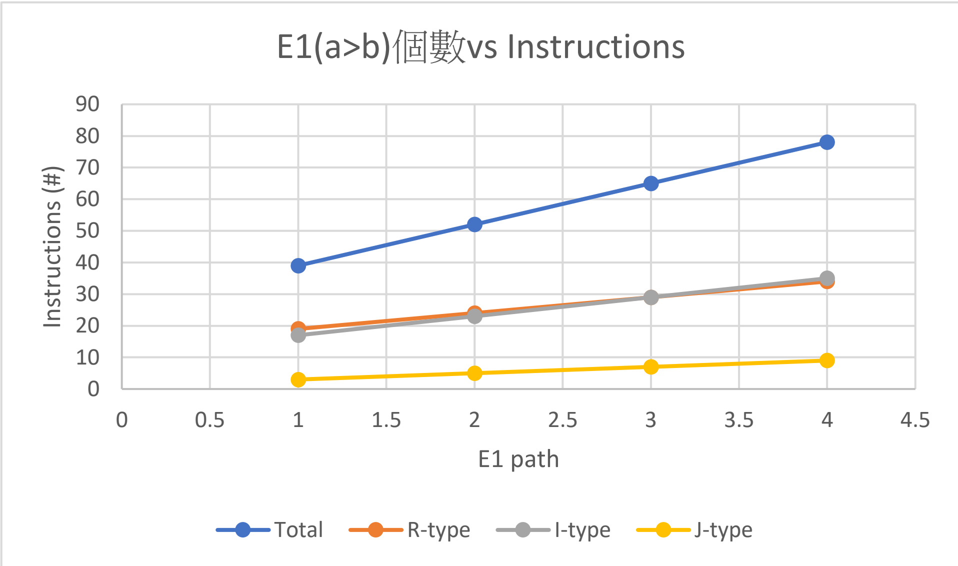
圖一、algorithm state diagram。

因此我們在此分析不同 path 的結果我們分成 3 種基本的情況來討論。分別為只有經過一次 E2 path 的情況、需要經過一次 E1(a>b)的情況以及需要經過一次 E1(a<b)的情況。我們分別以(a, b)輸入為(1, 1)、(2, 1)、(1, 2)為例子丟入分析。所得到 Instruction 分布如下表一。

表一、Problem 1 模擬結果與 type 分布。

situation	Instruction (#)	R-type (#)	I-type (#)	J-type (#)	R-type (%)	I-type (%)	J-type (%)
Only one E2	26	14	11	1	0.538	0.423	0.038
One E2, one E1(a>b)	39	19	17	3	0.487	0.436	0.077
One E2, one E1(a<b)	37	17	17	3	0.459	0.459	0.081

另外若是我們分析多個不同的 E1 個數，可以發現每經過多個 E1 path 的 instruction 數量是固定的，因此如下圖二顯示各種類分布的 instruction 是直線上升。



圖二、E1(a>b)個數對 instructions 比較。

因此經過分析我們知道每經過一次 E1(a>b) path，instruction 數量多 13 個，其中 R-type 多 5 個，I-type 多 6 個，J-type 多 2 個。另外同理，每經過一次 E1(a<b) path，instruction 數量多 11 個，其中 R-type 多 3 個，I-type 多 6 個，J-type 多 2 個。

在這裡我們需要討論說為何 R-type 在這兩個類似的路徑中會相差 2 個，由於為了節省 code size，我們在 E1(a>b)路徑中預加了 b 的值讓後面扣回來，來節省後面需要用的 jump instruction 數量和 code size。因此我們犧牲了此路徑的速度來換取 code size 的優化。

```
    # prepare needed arguments
    slt $t0, $a0, $a1           # else(a < b) 1:0
    bne $t0, $0, lableAlB       # 'else (a < b)', branch while a less than b
    sub $a0, $a0, $a1           # else if (a > b) a = a - b
    add $a1, $a1, $a0           # let b = (b + a) - a  below

3 references
lableAlB:
    sub $a1, $a1, $a0           # b = b - a
    jal     GCDloop             # recursive call
    lw      $ra, 0($sp)         # pop return address from the stack.
    addi    $sp, $sp, 4         # restore the stack
    j   ret                     # exit procedure
```

圖三、解釋上述 trade off 的部份。

再來我們分析三種基本的 instruction 所使用到的各種類分布如下表二。

表二、Problem 1 各種類分布。

situation	ALU (#)	Jump (#)	Branch (#)	Memory (#)	Other (#)
Only one E2	10	2	1	0	13
One E2, one E1(a>b)	15	5	3	2	14
One E2, one E1(a<b)	13	5	3	2	14

另外，若我們以同樣方法分析 E1 path 的數量，我們可以計算出每增加一級 E1 path 所需要增加的各種類個數，如下表三。

表三、Problem 1 增加級數的各種類分布。

situation	ALU (#)	Jump (#)	Branch (#)	Memory (#)	Other (#)
+ E1(a > b)	+5	+3	+2	+2	+1
+ E1(a < b)	+3	+3	+2	+2	+1

在此就更清楚表明了我在 code 中做的取捨，也就是剛好會多 2 次的 ALU。

最後我們檢視我們的 code size，compile 之後總共使用了 33 個 word 的位置來儲存 instructions。

3. Additional Discussion

證明此演算法的正確性以及其有限次數性。

定理：$a = bq + r \rightarrow \gcd(a, b) = \gcd(b, r)$

證明：假定兩數 g、h 分別代表$\gcd(a, b)$、$\gcd(b, r)$的值，即

$$g = \text{gcd}\ (a, b)$$
$$h = \text{gcd}\ (b, r)$$

(1) $g\big|a$且$g\big|b$，$\because r = a - bq \rightarrow q|r \therefore g \leq h$

(2) $h\big|b$且 $h\big|r$，$\because a = bq + r \rightarrow h|m \therefore g \geq h$

根據上(1)、(2)關係式，可知 g = h，即$\text{gcd}(a, b) = \text{gcd}\ (n, r)$

那在此我們 implement 演算法的方法為令 q 為±1，根據 a、b 誰大來改變 q 值。那根據我們每一次的運算都會使得下一級的 a、b 小於上一級，也就是 if a > b, $gcd_n(a_n, b_n) = gcd_{n+1}(a_{n+1} = a_n - b_n, b_{n+1} = b_n)$，其中因為$a_n,\ b_n \in$ $positive\ interger \therefore a_{n+1} \in positive\ interger$

且$0 < a_{n+1} < a_n$。而已知a_n是有限正整數，因此可以推論有限步驟後，a_{n+1}可以得到 gcd 的值，其最小值為 1。

二、Use Euclid's Algorithm to solve GCD via iterative method

1. Approach

先描述本小題會使用到基於 Euclid's Algorithm 的 2 個事實如下，題目假定 a、b 為兩正整數。

$$(E1) If\ a \neq b, say, a > b, then\ gcd(a, b) = gcd(a - b, b).$$
$$(E2)\ If\ a = b, then\ gcd(a, b) = a.$$

根據上述兩條件，我們若用 iterative 方法完成的話，我們同樣先在迴圈判斷是否 E2 條件成立，因為 E2 即為此演算法的中止條件，若成立則跳出迴圈，直接將數值 a 顯示出來。若 E2 不成立，則必為 E1 條件，比較 a、b 大小，若 a > b 則以下一級 $a_{n+1} = a - b, a_{n+1} = b$的數值繼續進行迴圈，以上皆在同一個 function 中完成。

另外，使用 assembly code 完成的部分，其邏輯與 c code 相同。先比較在 register 的 a, b 的值是否相同，相同則回傳 a，不同則進行比較 a, b 的值，我們以'slt'來完成，並直接進行邏輯減法運算即可。

2. Results and Discussion

同樣的我們先分析此演算法會進行的步驟，其 State diagram 與第一題相同，因此我們在此分析不同 path 的結果我們分成 3 種基本的情況來討論。分別為只有經過一次 E2 path 的情況、需要經過一次 E1(a>b)的情況以及需要經過一次 E1(a<b)的情況。我們分別以(a, b)輸入為(1, 1)、(2, 1)、(1, 2)為例子丟入分析。所得到 Instruction 分布如下表四。

表四、Problem 2 模擬結果與 type 分布。

situation	Instruction (#)	R-type (#)	I-type (#)	J-type (#)	R-type (%)	I-type (%)	J-type (%)
Only one E2	20	9	11	0	0.450	0.550	0.000
One E2, one E1(a>b)	26	13	13	0	0.500	0.500	0.000
One E2, one E1(a<b)	24	11	13	0	0.458	0.542	0.000

另外若是我們分析多個不同的 E1 個數，可以發現每經過多個 E1 path 的 instruction 數量是固定的，因此我們可以計算出每增加一級 E1 path 所需要增加的各種類個數，如下表五。

表五、Problem 2 增加級數的各種 type 增加量。

situation	Instruction (#)	R-type (#)	I-type (#)	J-type (#)
+ E1(a > b)	+7	+4	+3	+0
+ E1(a < b)	+5	+2	+3	+0

在這裡我們需要討論說為何 R-type 在這兩個類似的路徑中會相差 2 個，由於為了節省 code size，我們在 E1(a>b)路徑中預加了 b 的值讓後面扣回，可以讓我們不需要使用 j-type instruction。

再來我們分析三種基本的 instruction 所使用到的各種類分布如下表六。

表六、Problem 2 各種類分布。

situation	ALU (#)	Jump (#)	Branch (#)	Memory (#)	Other (#)
Only one E2	10	0	1	0	9
One E2, one E1(a>b)	13	0	3	3	10
One E2, one E1(a<b)	11	0	3	0	10

另外，若我們以同樣方法分析 E1 path 的數量，我們可以計算出每增加一級 E1 path 所需要增加的各種類個數，如下表七。

表七、Problem 2 增加級數的各種類分布表

situation	ALU (#)	Jump (#)	Branch (#)	Memory (#)	Other (#)
+ E1(a > b)	+3	+0	+3	+0	+1
+ E1(a < b)	+1	+0	+3	+0	+1

在此就更清楚表明了我在 code 中做的取捨，也就是剛好會多 2 次的 ALU。

最後我們檢視我們的 code size，compile 之後總共使用了 22 個 word 的位置來儲存 instructions。

三、Use Binary GCD Algorithm to solve GCD via recursive method

1. Approach

先描述本小題會使用到基於 Euclid's Algorithm 的 2 個事實如下，題目假定 a、b 為兩正整數。

$$(E1) If\ a \neq b, say, a > b, then\ gcd(a,b) = gcd(a-b,b).$$
$$(E2)\ If\ a = b, then\ gcd(a,b) = a.$$

另外還有

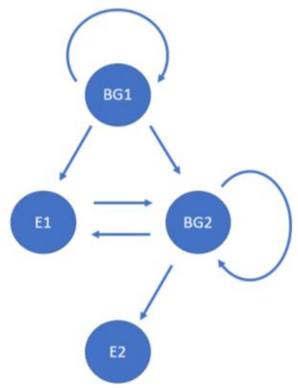
$$(BG1) If\ a \neq b, both\ a\ and\ b\ is\ even, then\ gcd(a,b) = 2 * gcd(\frac{a}{2},\frac{b}{2}).$$

$$(BG2) If\ a \neq b, one\ of\ a\ and\ b\ is\ odd, say\ b\ is\ odd, then\ gcd(a,b) = gcd(\frac{a}{2},b).$$

根據上述四個條件，我們若用 recursive 方法完成的話，我們先判斷是否 E2 條件成立，因為 E2 即為此演算法的中止條件，若成立則直接 return 數值 a。若 E2 不成立，則我們先判斷 a, b 的奇偶性，若其中一個為偶數的話，則進行 BG2 條件運算，遞迴呼叫$gcd(\frac{a}{2},b)$，這邊除 2 的方法，我們使用’srl’來完成。若兩個都是偶數則進行 BG1，遞迴呼叫 $gcd(\frac{a}{2},\frac{b}{2})$，在做 BG1 時要記得把 return 回來的值乘 2，這邊使用’sll’來完成。若兩者都是奇數，則必為 E1 條件，比較 a、b 大小，若 a > b 則 return $gcd(a-b,b)$的數值，即遞迴呼叫另一個 function。另外，由於要遞迴呼叫 function，因此要先將 return address 存入 stack 中再進行呼叫。

2. Results and Discussion

若我們先分析此演算法會進行的步驟，以 State diagram 表示，如下圖四。



圖四、algorithm state diagram。

因此我們在此分析不同 path 的結果我們分成 7 種基本的情況來討論。分別為只有經過一次 E2 path 的情況、需要經過一次 BG2(a > b)的情況、需要經過一次 BG2 (a < b)的情況、需要經過一次 BG2 和 E1(a > b)的情況、需要經過一次 BG2 和 E1(a < b)的情況、需要經過一次 BG2 和 BG1(a > b)的情況以及需要經過一次 BG2 和 BG1(a < b)的情況。我們分別以(a, b)輸入為(1, 1)、(2, 1)、(1, 2)、(3, 1)、(1, 3)、(4, 2)、(2, 4)為例子丟入分析。所得到 Instruction 分布如下表八。

表八、Problem 3 模擬結果與 type 分布。

situation	Instruction (#)	R-type (#)	I-type (#)	J-type (#)	R-type (%)	I-type (%)	J-type (%)
Only one E2 (1, 1)	26	14	11	1	0.538	0.423	0.038
One E2, one BG2(a > b) (2, 1)	40	16	20	4	0.400	0.500	0.100
One E2, one BG2(a < b) (1, 2)	40	16	20	4	0.400	0.500	0.100
One E2, one BG2 and E1(a > b) (3, 1)	58	21	30	7	0.362	0.517	0.121
One E2, one BG2 and E1(a < b) (1, 3)	56	19	30	7	0.339	0.536	0.125
One E2, one BG2 and BG1(a > b) (4, 2)	55	20	29	6	0.364	0.527	0.109

One E2, one BG2 and BG1(a < b) (2, 4)	55	20	29	6	0.364	0.527	0.109

由於經過前兩題的分析，我們知道 loop 走不同 path 的 instruction 數量具有加成性，因此在此題中我們也可以分析不同路徑所需要增加的 instruction 數量，如下表九。

表九、Problem 3 增加級數的各種 type 增加量。

situation	Instruction (#)	R-type (#)	I-type (#)	J-type (#)
+ BG2	+14	+2	+9	+3
+ E1(a > b)	+18	+5	+10	+3
+ E1(a < b)	+16	+3	+10	+3
+BG1	+15	+4	+9	+2

在這裡我們需要討論說為何 R-type 在這兩個類似的路徑中會相差 2 個，由於為了節省 code size，我們在 E1(a>b)路徑中預加了 b 的值讓後面扣回來，來節省後面需要用的 jump instruction 數量和 code size。因此我們犧牲了此路徑的速度來換取 code size 的優化。
再來我們分析七種基本的 instruction 所使用到的各種類分布如下表十。

表十、Problem 2 各種類分布。

situation	ALU (#)	Jump (#)	Branch (#)	Memory (#)	Other (#)
Only one E2 (1, 1)	10	2	1	0	13
One E2, one BG2(a > b) (2, 1)	15	6	4	2	13
One E2, one BG2(a < b) (1, 2)	10	6	4	2	13
One E2, one BG2 and E1(a > b) (3, 1)	22	10	8	4	14
One E2, one BG2 and E1(a < b) (1, 3)	20	10	8	4	14
One E2, one BG2 and BG1(a > b) (4, 2)	22	9	7	4	13
One E2, one BG2 and BG1(a < b) (2, 4)	22	9	7	4	13

另外，若我們以同樣方法分析 E1 path 的數量，我們可以計算出每增加一級 E1 path 所需要增加的各種類個數，如下表十一。

表十一、Problem 3 增加級數的各種類分布表。

situation	ALU (#)	Jump (#)	Branch (#)	Memory (#)	Other (#)
+BG2	+5	+4	+3	+2	+0
+ E1(a > b)	+7	+4	+4	+2	+1
+ E1(a < b)	+5	+4	+4	+2	+1
+BG1	+7	+3	+3	+2	+0

在此就更清楚表明了我在 code 中做的取捨，也就是剛好會多 2 次的 ALU。

最後我們檢視我們的 code size，compile 之後總共使用了 49 個 word 的位置來儲存 instructions。

3. Additional Discussion

   現有 code 的速度優化方法，由於為了減省 code size，在此沒有將此方法實作出來，但在此作一些優化的討論。由於我們可以從 state diagram 中觀察到 BG1 只會做有限次數之後若是跳到其他 state 就永遠不會在做了，因此一個可能的優化方法就是，我們在判斷完 E2 條件，先一直判斷是否是 BG1 條件並且做運算，若不是的話之後我們就多了一個假定條件，就是 a 與 b 不會同時是偶數，因此就可以省略在去走判斷 BG1 條件的情況了，可以在每一次的 loop 中減少判斷一次 a、b 的奇偶性。

四、Comparison

1. Overview

   (1) Instruction

   首先我們先比較 3 種不同方式的各種 path 的 instruction，如下表十二。

表十二、三種不同方法的各 state path 的 instruction 數量(A 為黃色，B 為綠色，C 為橘色)。

situation	Instruction (#)	R-type (#)	I-type (#)	J-type (#)
Only one E2	26	14	11	1
+ E1(a > b)	+13	+5	+6	+2
+ E1(a < b)	+11	+3	+6	+2
Only one E2	20	9	11	0
+ E1(a > b)	+7	+4	+3	+0
+ E1(a < b)	+5	+2	+3	+0
Only one E2	26	14	11	1
+ BG2	+14	+2	+9	+3
+ E1(a > b)	+18	+5	+10	+3
+ E1(a < b)	+16	+3	+10	+3
+BG1	+15	+4	+9	+2

根據上表十二觀察，首先先看基本量(E2)，由於方法 A 與方法 C 皆是採用 recursive method，因此我們可以看到相較於方法 B，他們最基本都還需要進行 call function 以及 return 的動作，因此基本 instructions 比 B 還多。而同時，由於 B 不需要 call function，因此在此可以以省下不需要使用的 J-type instruction，因此 B 方法沒有 J-type。

再來我們看到每增加 1 級的 instruction 增加數量。我們可以看到明顯的比較，C>A>B。用 iterative 所增加的 instruction 量自然最少，而使用 recursive 的會較多，由於方法 C 的判斷條件多，因此 C 的 instruction 增加量最多。

(2) Code size

Code size 以方法 B 為最小(22 words)，方法 A 為其次(33 words)，以方法 C 為最大(49 words)，在撰寫程式時，有稍微針對重複性的指令進行合併，在一些速度上與 size 做取捨。

(3) Complexity

Code 複雜度以方法 B 最為簡單，方法 A 其次，方法 C 最為複雜。而以數字收斂程度來說方法 C 是收斂最快的，而以方法 B 和方法 A 是差不多的收斂速度，奇收斂速度定義是我們所需要運算的級數。

2. Compare Part. A and Part. B

我們先根據上表十二不同方法的各 state path 的 instruction 數量來做 A 和 B 的比較差異，A 與 B 唯一不同的地方在於是呼叫 function 來做運算或是直接相減。我們可以

看到方法 A 在每一級需要多做的事情為'j'、'lw'、'sw'、'addi'、'addu'(move)。以上所需因此如下表，我們也可以看到 ALU、jump 數量以及 branch 都會多，而 memory 剛好一個 load、一個 save，會多 2 個 memory，如下表十三。

表十三、比較方法 A、方法 B 的各種分布。

situation	ALU (#)	Jump (#)	Branch (#)	Memory (#)	Other (#)
+ E1(a > b)	+5	+3	+2	+2	+1
+ E1(a < b)	+3	+3	+2	+2	+1
+ E1(a > b)	+3	+0	+3	+0	+1
+ E1(a < b)	+1	+0	+3	+0	+1

因此這邊可以推論方法 A 比方法 B 還沒有效率，無論是在時間上或是速度上都沒優勢。

3. Compare Part. A and Part. C

我們比較同樣為 recursive method 的方法 A 和方法 C，可以看見儘管方法 C 在每一級的運算消耗都比方法 A 還要大，如上表三種不同方法的各 state path 的 instruction 數量。由由於方法 C 演算法的關係，方法 C 在每一次進行 BG1 或者 BG2 所使數字收斂的速度遠大於方法 A，我們以一極端案例為例若我們要尋找$\gcd(2^n,1)$。

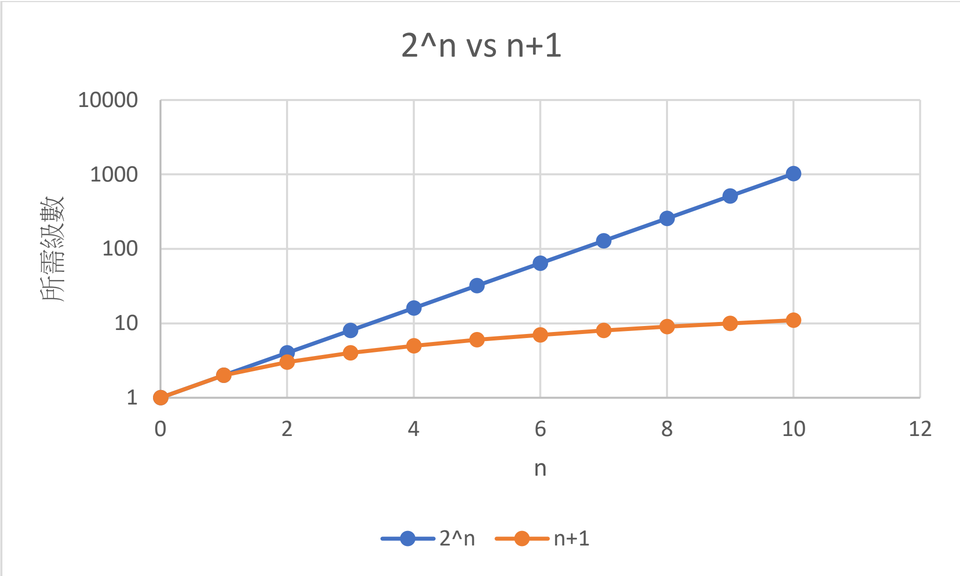
$$method\ A\colon \gcd(2^n,1) = \gcd(2^n-1,1) = \gcd(2^n-2,1) = \cdots = \gcd(1,1) = 1$$

若以方法 A 需要收斂到答案需要做2^n次才會得到答案。

$$method\ C\colon \gcd(2^n,1) = \gcd(2^{n-1},1) = \gcd(2^{n-2},1) = \cdots = \gcd(1,1) = 1$$

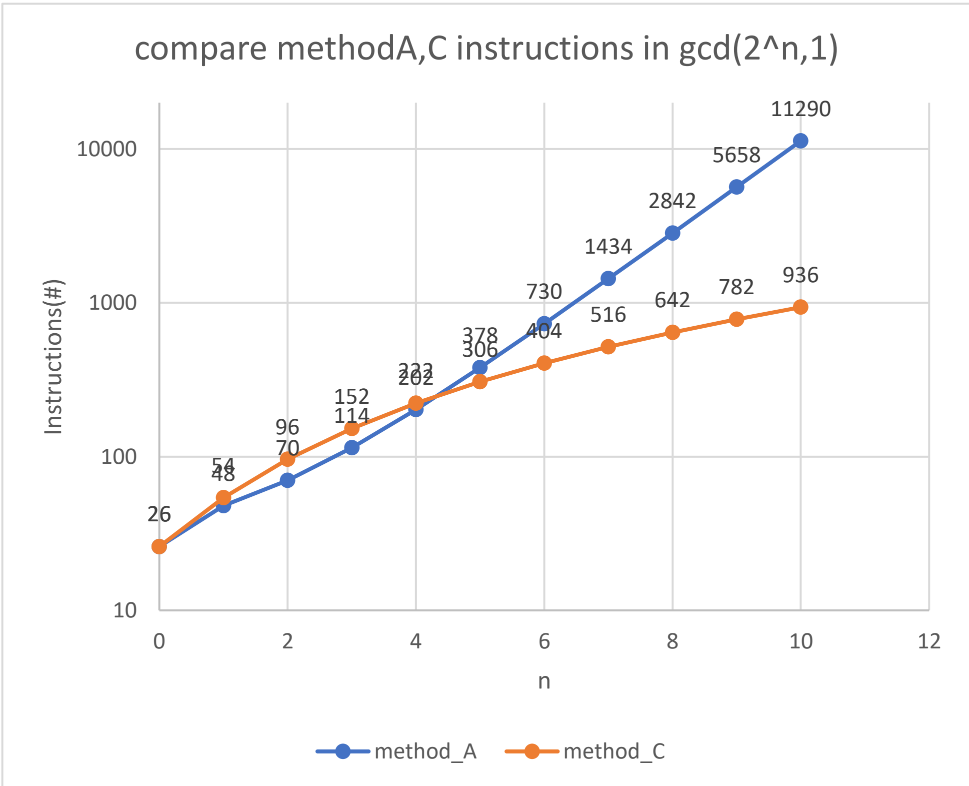
若以方法 C 需要收斂到答案需要做$n+1$次才會得到答案。

如此在運算級數上就有極大的差異，如下圖五。



圖五、比較 2^n 與 n+1 級數之差異，數值以對數軸表示。

因此若我們考量每一級的 instruction 增加量，比較方法 A 和方法 C，如下圖六。



圖六、計算 gcd(2^n,1)使用方法 A 與方法 C 的 Instructions 數量比較

根據上圖六比較我們就可以看見方法 C 在數字較大(約為 2^5 以上)之後就會展現優勢。

五、Conclusion

1. 在相同演算法下，方法 B 比方法 A 快。
2. 在不同演算法下，比較 GCD 數字差距小(即有差異的 2 個位元間隔數量)時通常方法 B 會比方法 C 還有方法 A 還快。
3. 在不同演算法下，比較 GCD 數字差距大時(即有差異的 2 個位元間隔數量)，方法 C 會比方法 A 和方法 B 還要快。
4. 在不同演算法下，比較 GCD 數字的大小，與各種方法的速度無絕對正相關，例如丟入(178956970,89478485)與丟入(2,1)對於演算速度上是一樣的。
5. 對於方法 A 與方法 C 來說，在級數少時 R-type 的分布較多，其次為 I-type，最少為 J-type。而在級數多時，I-type 的分布會較多，R-type 為其次，最少為 J-type。
6. 對於方法 B 來說，大致上的 I-type 分布會比 R-type 多，沒有 J-type。