作業資訊： EE3450計算機結構 Final Project

學生資訊： 郭柏辰 107012045

作業內容：

1. Use Euclid's Algorithm to solve GCD via recursive method
2. Approach

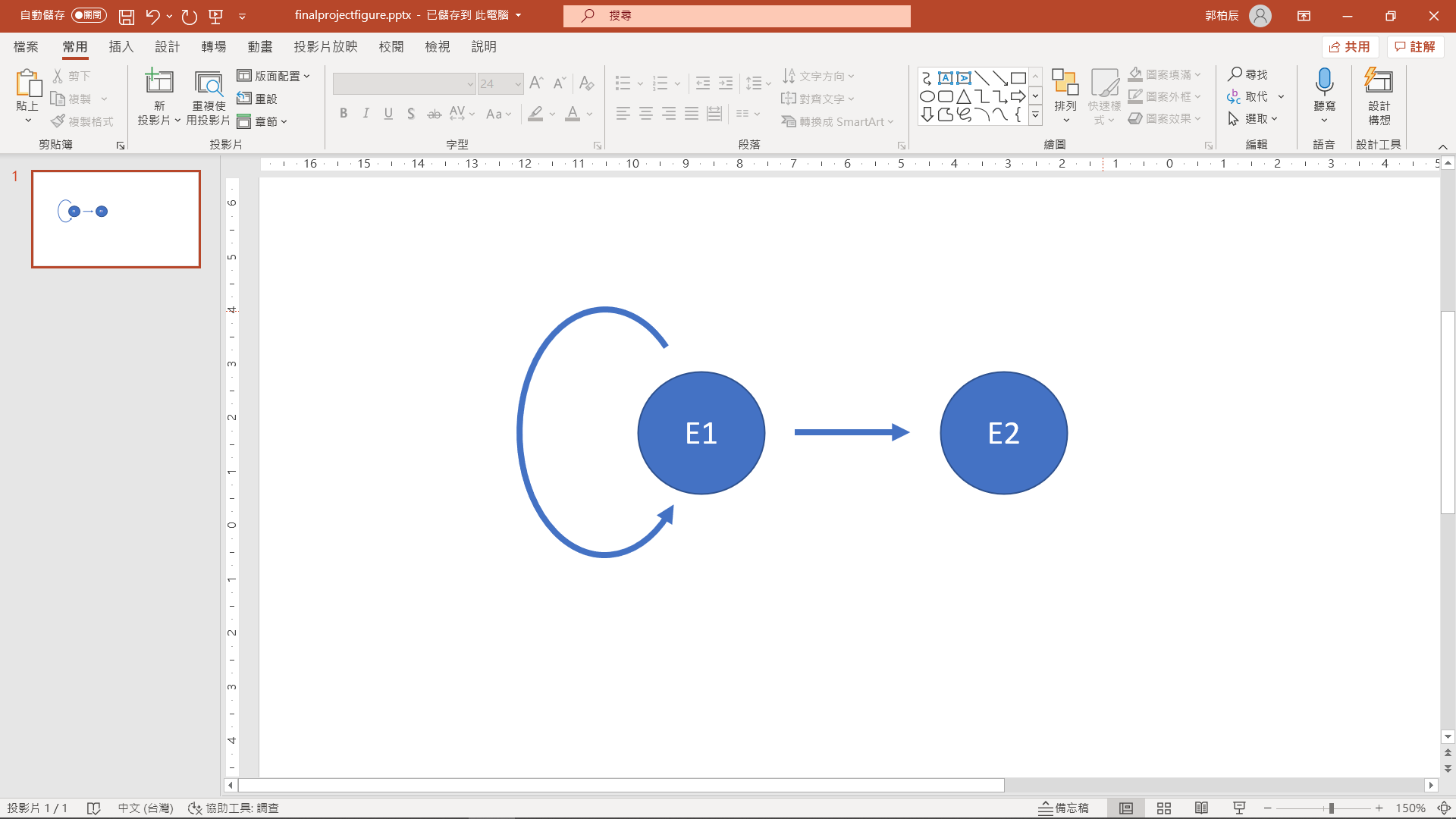
先描述本小題會使用到基於Euclid's Algorithm 的2個事實如下，題目假定a、b為兩正整數。

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 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。

1. 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。

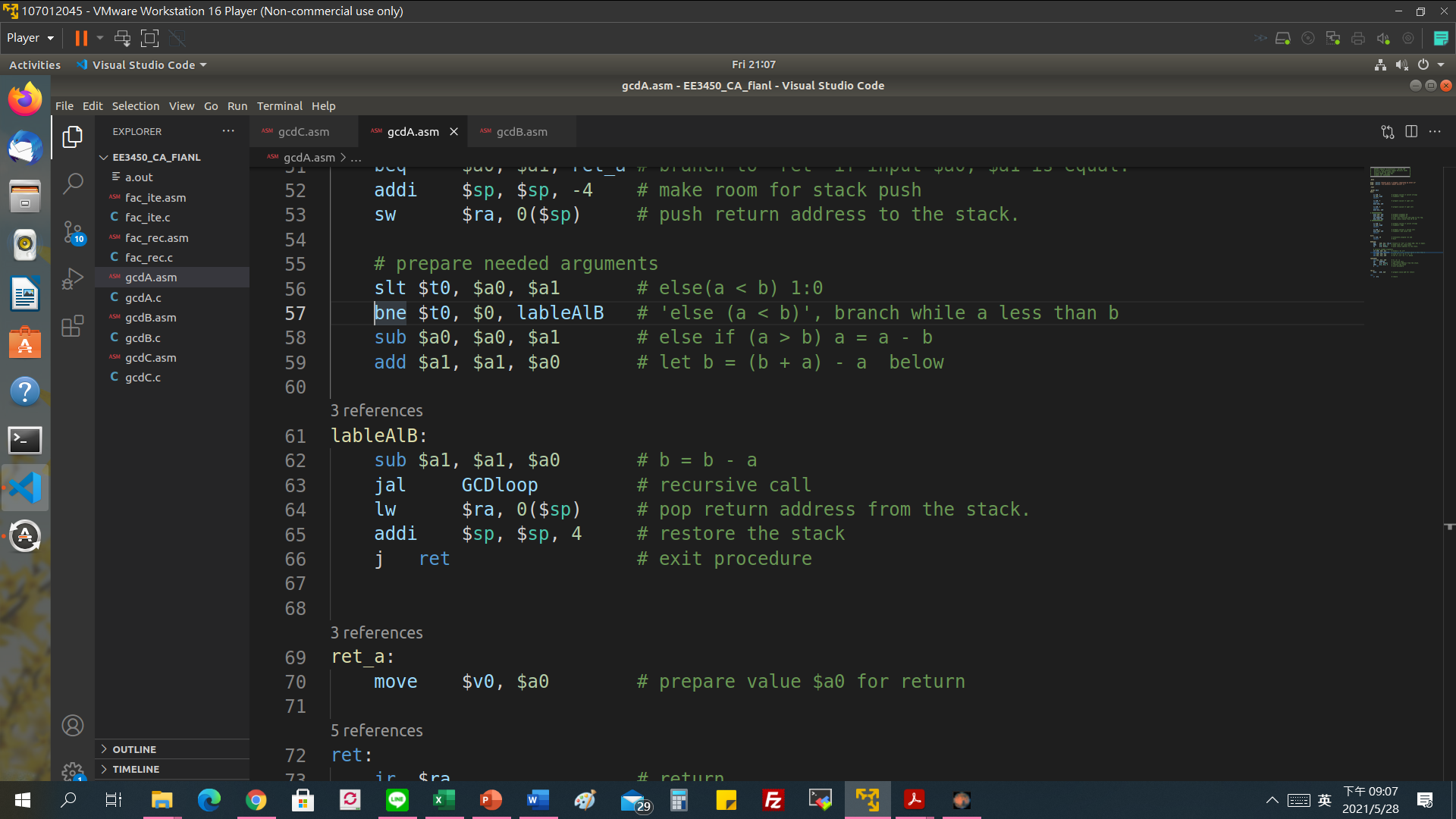
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| situation | Instruction  (#) | R-type  (#) | I-type  (#) | J-type  (#) | R-type  (%) | I-type  (%) | J-type  (%) |
| Only one E2 | 26 | 14 | 11 | 1 | 0.538 | 0.423 | 0.038 |
| One E2, one E1(a>b) | 39 | 19 | 17 | 3 | 0.487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



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。

1. Additional Discussion

證明此演算法的正確性以及其有限次數性。

定理：

證明：假定兩數g、h分別代表、的值，即

根據上(1)、(2)關係式，可知g = h，即

那在此我們implement演算法的方法為令q為，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, ，其中因為

且。而已知是有限正整數，因此可以推論有限步驟後，可以得到gcd的值，其最小值為1。

1. Use Euclid's Algorithm to solve GCD via iterative method
2. 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。

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。

1. 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。

1. Use Binary GCD Algorithm to solve GCD via recursive method
2. Approach

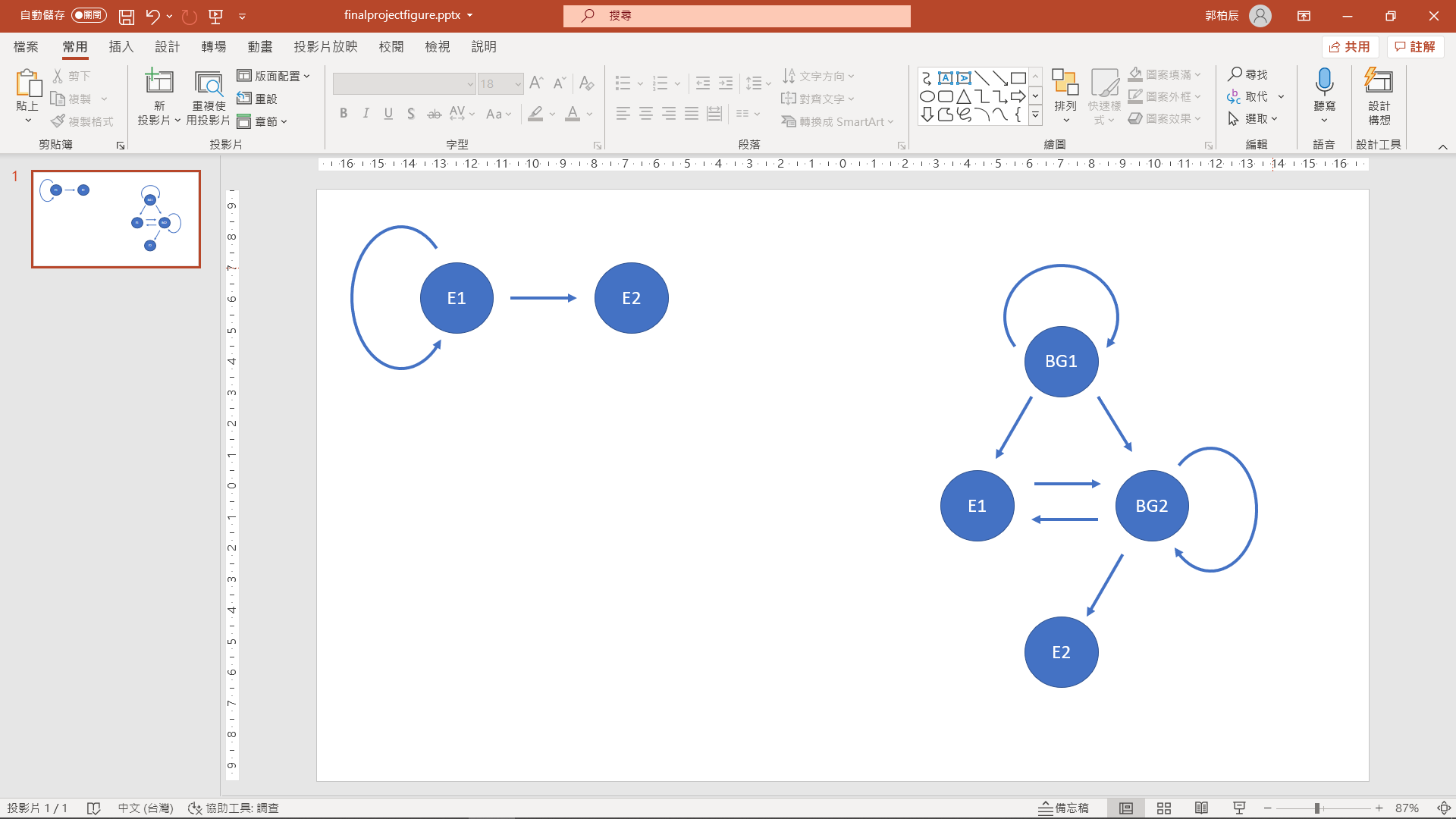
先描述本小題會使用到基於Euclid's Algorithm 的2個事實如下，題目假定a、b為兩正整數。

另外還有

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，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，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 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。

1. 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。

1. 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。

1. Comparison
2. Overview
3. 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。

1. 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。

1. 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。

1. 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。

1. 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。

若以方法A需要收斂到答案需要做次才會得到答案。

若以方法C需要收斂到答案需要做次才會得到答案。

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。

1. Conclusion
2. 在相同演算法下，方法B比方法A快。
3. 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。
4. 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。
5. 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。
6. 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。
7. 對於方法B來說，大致上的I-type分布會比R-type多，沒有J-type。