Lab#2

☐ 實驗內容

☐ 實驗結果

☐ 實驗心得

☐ 附錄

Name: 張嵩禾
StudentID: B083040041
Subject:List SchedulingDate: 4/21

□ 實驗內容

目標：最小化工作排程所需的總時間，List scheduling是在保持最佳資源利用的同時，以最短的週期來完成電路的目標。

Step 1:

前置參數設計

設計3個class，分別為stater、thisline、op，用以物件化的紀錄排程過程所需參數。

class stater:目的用於記錄在排程過程中，目前 state **所剩餘的ALU數量**以及**目前位於哪一個 state**。

```
class stater:  #Store the current available number of ALUs and current stage
    adder : int
    mul   : int
    statenow : int
```

class thisline:目的用於記錄input檔案中每一列的 source 以及 Result，**其中** done **用於記錄這列是否已經被輸出完成**，若以輸出過則可跳過這列。

```
class thisline: # Store operators and operation numbers
    op  : int
    op1 : int
    op2 : int
    res : int
    done : int
    def __init__(self,op ,op1,op2,res,done):
        self.op = op
        self.op1 = op1
        self.op2 = op2
        self.res = res
        self.done = done
```

class op :目的用於記錄排程過程中不同運算單位之狀態， rdy **用於記錄這個運算子**是否已準備完成(No Dependency)，counter 則用來**倒數目的運算子完成所需 state 數目**， adderr 和 mull 則用來倒數**ALU完成所需state數目**。

```
class op : # Record different operation numbers' state
    rdy : int
    counter : int # Used to count down whether the operation is completed
    adderr  : int # Used to count down whether the adder is avaliable now
    mull    : int # Used to count down whether the multiplexor is avaliable now
    def __init__(self,rdy,counter,adderr,mull):
        self.rdy = rdy
        self.counter = counter
        self.adderr = adderr
        self.mull = mull
```

Step 2:

計算 Critical path

使用參數: classer : 為一個存放 thisline 的 list，用於紀錄 txt 檔案中
每一列的參數(operator 、 source & Result) 。

path[] : 為一個存放 critical path 長度的 list。

```
# ---------------↓ computing the critical path length ↓-----------------------------------------
path.clear()
for i in range (200):
    path.append(0) # initialize the path number
counter = len(classer)-1
while(1):
    for i in range(counter + 1 , len(classer)):
        if(classer[counter].res == classer[i].op1 or classer[counter].res == classer[i].op2):
            if(classer[counter].op == 1):
                if(path[classer[i].res] + AdderCycle > path[classer[counter].res]):
                    path[classer[counter].res] = path[classer[i].res] + AdderCycle
        if(classer[counter].res == classer[i].op1 or classer[counter].res == classer[i].op2):
            if(classer[counter].op == 2):
                if(path[classer[i].res] + MultiplexorCycle > path[classer[counter].res]):
                    path[classer[counter].res] = path[classer[i].res] + MultiplexorCycle
    counter = counter - 1
    if(counter == -1):
        break
# ---------------↑ computing the critical path length ↑-----------------------------------------
```

```
counter = len(classer)-1
while(1):
    for i in range(counter + 1 , len(classer)):
```

先將 counter 設為 len (classer) -1 (由**檔案結尾往回看**)，由每一列開始**往後檢查 (counter+1)** 來計算 critical path。

```
if(classer[counter].res == classer[i].op1 or classer[counter].res == classer[i].op2):
    if(classer[counter].op == 1):
        if(path[classer[i].res] + AdderCycle > path[classer[counter].res]):
            path[classer[counter].res] = path[classer[i].res] + AdderCycle
```

若目前算式之目的位置，和之後的**來源位置相等且來源位置之critical path + 計算所需cycle 數目 > 目前算式之目的位置之critical path** ，將目的位置算子之**critical path 設為來源位置之critical path + 計算所需cycle 數目**，以達到計算最常路徑之目的。

Step 3:

判斷相依姓 & 加入 ready list

使用參數: rdyline : ready list 檢查相依性後，放入目前**可執行**之運算。

flagg1 & flagg2 : 用以記錄具相依性之算子，若不具相依性設為0。

```python
# ------------------------- ↓ checking the dependencies and add to the ready list ↓ ------------------------------
for zz in range (2000): # Each iteration represents a stage
    rdyline.clear()

    for i in range (len(classer)):

        if(classer[i].done != 1):

            if(classer[i].op == 1): # Adder
                if(stater.adder!=0):

                    for j in range (0 , i): # Check for dependencies

                        if((classer[i].op1==classer[j].res) ):
                            flagg1 = classer[j].res

                        if((classer[i].op2==classer[j].res) ):
                            flagg2  = classer[j].res
                    if(flagg1==0 and flagg2==0): # No dependencies
                        rdyline.append(i) # add to the ready list
else:

    if(((oper[flagg1].rdy==1) and (oper[flagg2].rdy==1)) or ( (flagg1 == 0)
    and (oper[flagg2].rdy==1) ) or ((oper[flagg1].rdy==1) and (flagg2 == 0))): # The dependent data is ready

        rdyline.append(i) # add to the ready list


    flagg1 = 0
    flagg2 = 0
```

```python
if(stater.adder!=0):

    for j in range (0 , i): # Check for dependencies

        if((classer[i].op1==classer[j].res) ):
            flagg1 = classer[j].res

        if((classer[i].op2==classer[j].res) ):
            flagg2  = classer[j].res
    if(flagg1==0 and flagg2==0): # No dependencies
        rdyline.append(i) # add to the ready list
```

先確認是否還有剩餘 ALU 可以使用，遍歷目前運算式之前的所有式子**確認是否存在相依性**，若存在，將相依之**運算子存入flagg1 or flagg 2**。反之若**不存在相依性(flagg1 and flagg 2 == 0)**，則**將此運算式加入 ready list**。

Step 3:

判斷相依姓 & 加入ready list

使用參數: rdyline : ready list 檢查相依性後，放入目前**可執行**之運算。

flagg1 & flagg2 : 用以記錄具相依性之算子，若不具相依性設為0。

```python
# ------------------------- ↓ checking the dependencies and add to the ready list ↓ ------------------------------
for zz in range (2000): # Each iteration represents a stage
    rdyline.clear()

    for i in range (len(classer)):

        if(classer[i].done != 1):

            if(classer[i].op == 1): # Adder

                if(stater.adder!=0):

                    for j in range (0 , i): # Check for dependencies

                        if((classer[i].op1==classer[j].res) ):
                            flagg1 = classer[j].res

                        if((classer[i].op2==classer[j].res) ):
                            flagg2  = classer[j].res
                    if(flagg1==0 and flagg2==0): # No dependencies
                        rdyline.append(i) # add to the ready list
else:
    if(((oper[flagg1].rdy==1) and (oper[flagg2].rdy==1)) or ( (flagg1 == 0)
    and (oper[flagg2].rdy==1) ) or ((oper[flagg1].rdy==1) and (flagg2 == 0))): # The dependent data is ready

        rdyline.append(i) # add to the ready list

    flagg1 = 0
    flagg2 = 0
```

```python
if(((oper[flagg1].rdy==1) and (oper[flagg2].rdy==1)) or ( (flagg1 == 0)
and (oper[flagg2].rdy==1) ) or ((oper[flagg1].rdy==1) and (flagg2 == 0))):

    rdyline.append(i) # add to the ready list
```

若目前之運算式來源存在相依性，**oper(class op list) 中的 rdy 參數來確認目前來源是否以運算完成**。若已運算完成將其加入ready list 。以上演示為 adder 之情況，Multiplexor 亦如此。

Step 4:

輸出

定義函式：output ( )：找出目前 ready list 中 **critical path 最長之運算式**，並將其輸出。

```
def output():
    ff = open("Scheduling_outcome.txt","a")
    max = 0
    for i in range(len(rdyline)): # find the longest critical path
        if(path[classer[rdyline[i]].res] >= max):
            max = path[classer[rdyline[i]].res]
            falger = rdyline[i]

    if(classer[falger].op == 1): # Adder
        if(stater.adder!=0):
            stater.adder=stater.adder-1 # Adder using
            stringout = "state : " + str(nowstate.statenow) +" v" + str(classer[falger].res) + " v" + str(classer[falger].op1)
            print(stringout)
            ff.write(stringout + "\n")
            oper[classer[falger].res].counter = AdderCycle # Check whether the operation number has finished computing
            oper[classer[falger].res].rdy = 0
            oper[classer[falger].res].adderr = AdderCycle # Check whether the adder has finished computing
            haveedit.append(falger) # already output
```

```
for i in range(len(rdyline)): # find the longest critical path
    if(path[classer[rdyline[i]].res] >= max):
        max = path[classer[rdyline[i]].res]
        falger = rdyline[i]
```

遍歷目的算子 (classer [rdyline[i]].res) 之 critical path (path [classer [rdyline [i]]. res])，**找出最大Critical path 之運算式並記錄**。

```
if(stater.adder!=0):
    stater.adder=stater.adder-1 # Adder using
    stringout = "state : " + str(nowstate.statenow) +" v" + str(classer[falger].res) + " v" + str(classer[falger].op1)
    print(stringout)
    ff.write(stringout + "\n")
    oper[classer[falger].res].counter = AdderCycle # Check whether the operation number has finished computing
    oper[classer[falger].res].rdy = 0
    oper[classer[falger].res].adderr = AdderCycle # Check whether the adder has finished computing
    haveedit.append(falger) # already output
```

將此運算式輸出，並同時減少一個使用到的ALU (stater.adder = stater.adder-1)。另外將此運算式之目的運算子之 counter 以及 adder (or mull) 參數設為 ALU 所需 cycle 數目，藉此來**倒數此運算所需 states 數目**，用以**判斷 ALU 以及目的算子是否已運算結束**。

Step 5:

更新 state

使用參數: op.counter : 記錄此算子所運用到的ALU運算需state數。
op.rdy : 記錄此算子是否已運算完畢(0 : 未完成 、 1 : 完成)。
op.adder : 記錄 Adder 所需 state 數。
op.mull : 記錄 Multiplexor 所需 state 數。
stater.adderr : 記錄 Adder 剩餘數量。
stater.mull : 記錄 Multiplexor 剩餘數量。
stater.state : 記錄目前state。

```
# -------------------------↓ Update the state ↓-------------------------------------------

else: # there are nothing in the ready list update!!
    stater.statenow = stater.statenow + 1

    for x in range(200): # Update the ALU and operation numbers' states
        if(oper[x].counter!=0):
            oper[x].counter = oper[x].counter - 1
            if(oper[x].counter == 0):
                oper[x].rdy = 1

        if(oper[x].adderr!=0):
            oper[x].adderr = oper[x].adderr - 1
            if(oper[x].adderr == 0):
                stater.adder = stater.adder + 1
        if(oper[x].mull!=0):
            oper[x].mull = oper[x].mull - 1
            if(oper[x].mull == 0):
                stater.mul = stater.mul + 1

# -------------------------↑ Update the state ↑-------------------------------------------
```

```
for x in range(200): # Update the ALU and operation numbers' states
    if(oper[x].counter!=0):
        oper[x].counter = oper[x].counter - 1
        if(oper[x].counter == 0):
            oper[x].rdy = 1

    if(oper[x].adderr!=0):
        oper[x].adderr = oper[x].adderr - 1
        if(oper[x].adderr == 0):
            stater.adder = stater.adder + 1
```

每一輪更新 state 時，將 op.counter 和 op.adder (or op.mull) - 1 ，表示**更新一輪後剩餘 clock cycle 數目 – 1**。此外，若**倒數 ALU 之參數倒數為 0** ，則將可用 ALU 數目 + 1，**表運算完成，可供其他運算式使用**。

☐ 實驗結果

透過帶入**不同的 cycle 數目**以及**不同 Limitation of ALUs**，來驗證排程結果是否正確，以下為 **MultiplexorCycle 設為 2 AdderCycle 設為 2**，且**ALUs數目個限制為 2** 之結果，可以看到與投影片上所做出的排程有一樣的state數目。

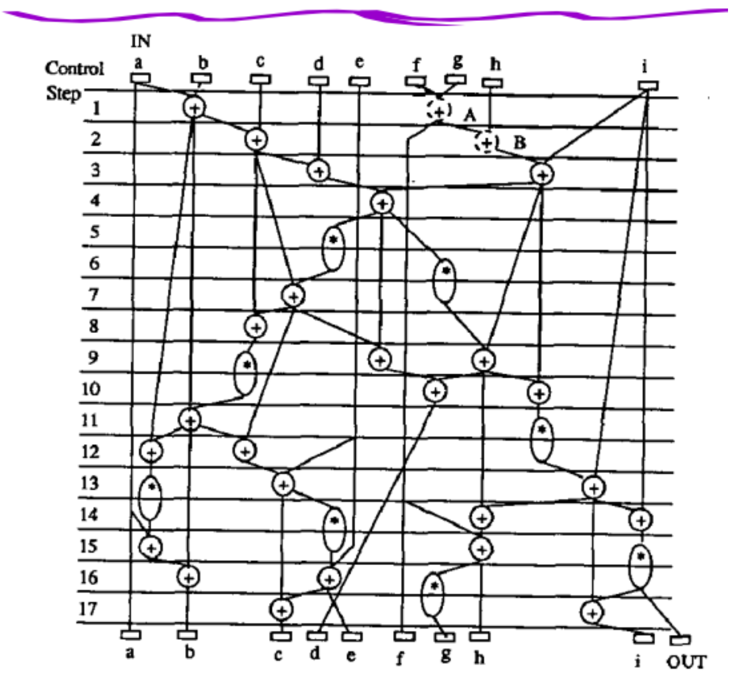
```
MultiplexorCycle = 2
AdderCycle = 1
```

```
intput a file nameDFG1.txt

 limitation of multiplexor...2

 limitation of Adder...2
state : 1 v11 v6 v7
state : 1 v10 v1 v2
state : 2 v13 v11 v8
state : 2 v12 v10 v3
state : 8 v25 v23 v15
state : 9 v21 v20 v20
state : 9 v27 v25 v25
state : 9 v22 v19 v16
state : 10 v24 v22 v23
state : 11 v26 v10 v21
state : 11 v32 v27 v9
state : 12 v29 v26 v19
state : 12 v28 v10 v26
state : 13 v31 v29 v5
state : 13 v35 v32 v9
state : 13 v30 v28 v28
state : 14 v33 v31 v31
state : 14 v34 v23 v32
state : 15 v38 v35 v35
state : 15 v37 v11 v34
state : 15 v36 v1 v30
state : 16 v40 v33 v5
state : 16 v41 v37 v37
state : 16 v39 v36 v26
state : 17 v43 v32 v38
state : 17 v42 v31 v40
```

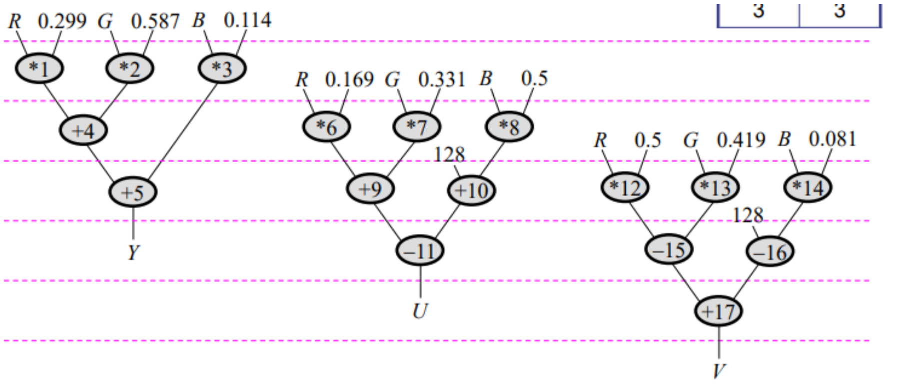
DFG1



同時也將 RGBtoYUV 編輯成可輸入排程之 txt 檔案（RGBtoYUV.txt），其餘之排程結果展示於附錄。

RGBtoYUV.txt

```
17
2 169 169 1
2 169 169 2
2 169 169 3
1 1 2 4
2 169 169 6
2 169 169 7
2 169 169 8
1 3 4 5
1 6 7 9
1 169 8 10
2 169 169 12
2 169 169 13
2 169 169 14
1 9 10 11
1 12 13 15
1 169 14 16
1 15 16 17
```



□ 實驗心得

此次實驗，在最開始撰寫程式時，是先以 **FIFO 作為排程考量**，以檔案最優先讀取之運算式做最早的運算，用此方式做排程時，所**顯示之結果和最後使用 critical path 作為優先權考量之 list scheduling 其實並沒有相差太多**，然而，若使用 FIFO 的方式做排程，Input 檔案需做良好的設計，否則會造成排程效果不彰，但 **critical path 作為優先權卻沒有這樣的問題，同時也能達到更好的Scheduling效果**。

```
if(stater.adder!=0):

    for j in range (0 , i): # Check for dependencies

        if((classer[i].op1==classer[j].res) ):
            flagg1 = classer[j].res

        if((classer[i].op2==classer[j].res) ):
            flagg2  = classer[j].res
    if(flagg1==0 and flagg2==0): # No dependencies

        stater.adder=stater.adder-1
        stringout = "state : " + str(nowstate.statenow) +" " +str(classer[i].res) + " " + str(classer[i].op1) + " " + str(classer[i].op2)
        print(stringout)

        oper[classer[i].res].counter = AdderCycle # Check whether the operation number has finished computing
        oper[classer[i].res].rdy = 0
        oper[classer[i].res].adderr = AdderCycle # Check whether the adder has finished computing

        haveedit.append(i)
```

```
if(flagg1==0 and flagg2==0): # No dependencies

    stater.adder=stater.adder-1
    stringout = "state : " + str(nowstate.statenow) +" " +str(classer[i].res) + " " + str(classer[i].op1) + " " + str(classer[i].op2)
    print(stringout)

    oper[classer[i].res].counter = AdderCycle # Check whether the operation number has finished computing
    oper[classer[i].res].rdy = 0
    oper[classer[i].res].adderr = AdderCycle # Check whether the adder has finished computing
```

由以上程式碼可以看到，原本的撰寫方式為，若不存在相依性直接輸出，這樣不需要判斷 critical path 之長度即可作**先讀入先排程**。

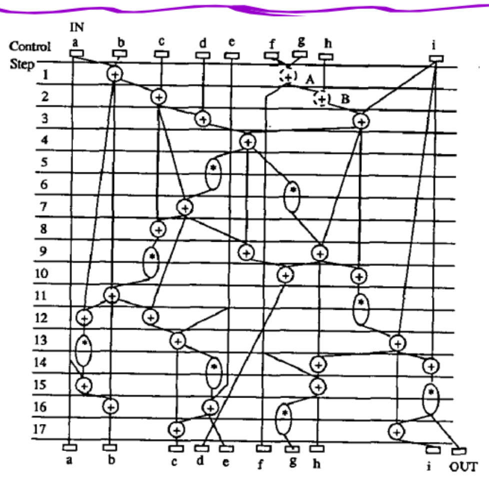
然而經由幾次比對測試之後，發現兩種排程在設計好 Input file 的前提下，並沒有顯著的 state 差異 ，進步效果有限，或許是樣本不夠多，或是測試不夠完善，但我認為這樣的結果或許表示著較簡單的排程演算法也能達到不錯的效果，因此未來在撰寫時可以多方考量。

□ 附錄

Resource Constraint
Mult Constraint: 2 Add Constraint: 2

```
state : 1 v11 v6 v7
state : 1 v10 v1 v2
state : 2 v13 v11 v8
state : 2 v12 v10 v3
state : 3 v15 v13 v9
state : 3 v14 v12 v4
state : 4 v16 v14 v15
state : 5 v17 v16 v16
state : 5 v18 v16 v16
state : 7 v19 v17 v12
state : 7 v23 v18 v15
state : 8 v20 v12 v19
state : 8 v25 v23 v15
state : 9 v21 v20 v20
state : 9 v27 v25 v25
state : 9 v22 v19 v16
state : 10 v24 v22 v23
state : 11 v26 v10 v21
state : 11 v32 v27 v9
state : 12 v29 v26 v19
state : 12 v28 v10 v26
state : 13 v31 v29 v5
state : 13 v35 v32 v9
state : 13 v30 v28 v28
state : 14 v33 v31 v31
state : 14 v34 v23 v32
state : 15 v38 v35 v35
state : 15 v37 v11 v34
state : 15 v36 v1 v30
state : 16 v40 v33 v5
state : 16 v41 v37 v37
state : 16 v39 v36 v26
state : 17 v43 v32 v38
state : 17 v42 v31 v40
****************************
*    Resource Constraint    *
*                           *
*     Mult Constraint:2     *
*     Add  Constraint:2     *
*                           *
****************************
```

DFG1



Resource constraints:

*	+
1	1
1	2
2	1
2	2

Resource Constraint
Mult Constraint: 1 Add Constraint: 1

```
state : 1 v11 v6 v7
state : 2 v10 v1 v2
state : 3 v13 v11 v8
state : 4 v12 v10 v3
state : 5 v15 v13 v9
state : 6 v14 v12 v4
state : 7 v16 v14 v15
state : 8 v17 v16 v16
state : 10 v19 v17 v12
state : 10 v18 v16 v16
state : 11 v20 v12 v19
state : 12 v23 v18 v15
state : 12 v21 v20 v20
state : 13 v25 v23 v15
state : 14 v27 v25 v25
state : 14 v26 v10 v21
state : 15 v29 v26 v19
state : 16 v32 v27 v9
state : 17 v31 v29 v5
state : 18 v28 v10 v26
state : 18 v33 v31 v31
state : 19 v35 v32 v9
state : 20 v30 v28 v28
state : 20 v34 v23 v32
state : 21 v40 v33 v5
state : 22 v38 v35 v35
state : 22 v37 v11 v34
state : 23 v36 v1 v30
state : 24 v22 v19 v16
state : 24 v41 v37 v37
state : 25 v43 v32 v38
state : 26 v42 v31 v40
state : 27 v39 v36 v26
state : 28 v24 v22 v23
****************************
*    Resource Constraint    *
*                           *
*     Mult Constraint:1     *
*     Add  Constraint:1     *
*                           *
****************************
```

Resource Constraint
Mult Constraint: 1 Add Constraint: 2

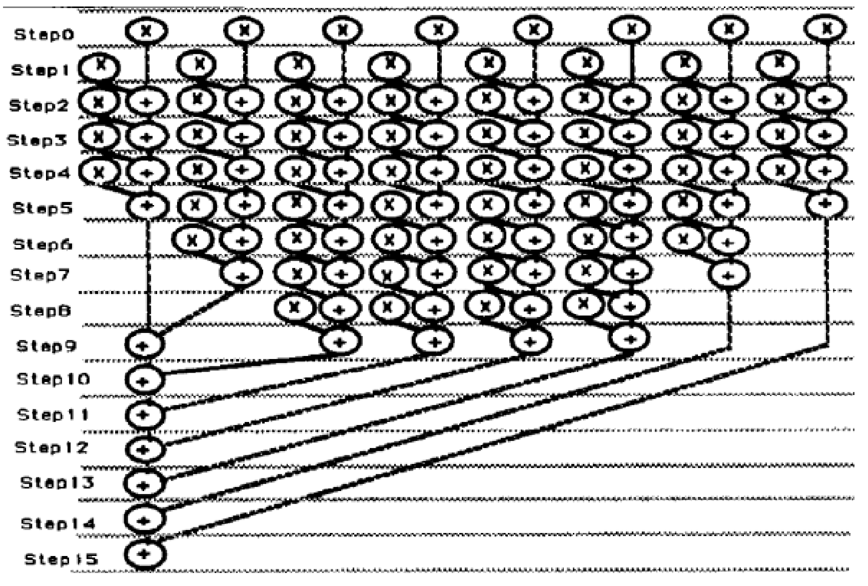
```
state : 1 v11 v6 v7
state : 1 v10 v1 v2
state : 2 v13 v11 v8
state : 2 v12 v10 v3
state : 3 v15 v13 v9
state : 3 v14 v12 v4
state : 4 v16 v14 v15
state : 5 v17 v16 v16
state : 7 v19 v17 v12
state : 7 v18 v16 v16
state : 8 v20 v12 v19
state : 8 v22 v19 v16
state : 9 v23 v18 v15
state : 9 v21 v20 v20
state : 10 v25 v23 v15
state : 10 v24 v22 v23
state : 11 v27 v25 v25
state : 11 v26 v10 v21
state : 12 v29 v26 v19
state : 12 v28 v10 v26
state : 13 v32 v27 v9
state : 13 v31 v29 v5
state : 13 v30 v28 v28
state : 14 v35 v32 v9
state : 14 v34 v23 v32
state : 15 v33 v31 v31
state : 15 v37 v11 v34
state : 15 v36 v1 v30
state : 16 v39 v36 v26
state : 17 v38 v35 v35
state : 17 v40 v33 v5
state : 18 v42 v31 v40
state : 19 v43 v32 v38
state : 19 v41 v37 v37
****************************
*    Resource Constraint    *
*                           *
*     Mult Constraint:1     *
*     Add  Constraint:2     *
*                           *
****************************
```

Resource Constraint
Mult Constraint: 2 Add Constraint: 1

```
state : 1 v11 v6 v7
state : 2 v10 v1 v2
state : 3 v13 v11 v8
state : 4 v12 v10 v3
state : 5 v15 v13 v9
state : 6 v14 v12 v4
state : 7 v16 v14 v15
state : 8 v17 v16 v16
state : 8 v18 v16 v16
state : 10 v19 v17 v12
state : 11 v20 v12 v19
state : 12 v23 v18 v15
state : 12 v21 v20 v20
state : 13 v25 v23 v15
state : 14 v27 v25 v25
state : 14 v26 v10 v21
state : 15 v29 v26 v19
state : 16 v32 v27 v9
state : 17 v31 v29 v5
state : 18 v28 v10 v26
state : 18 v33 v31 v31
state : 19 v35 v32 v9
state : 19 v30 v28 v28
state : 20 v38 v35 v35
state : 20 v34 v23 v32
state : 21 v40 v33 v5
state : 22 v37 v11 v34
state : 23 v36 v1 v30
state : 23 v41 v37 v37
state : 24 v22 v19 v16
state : 25 v43 v32 v38
state : 26 v42 v31 v40
state : 27 v39 v36 v26
state : 28 v24 v22 v23
****************************
*    Resource Constraint    *
*                           *
*     Mult Constraint:2     *
*     Add  Constraint:1     *
*                           *
****************************
```

□ 附錄

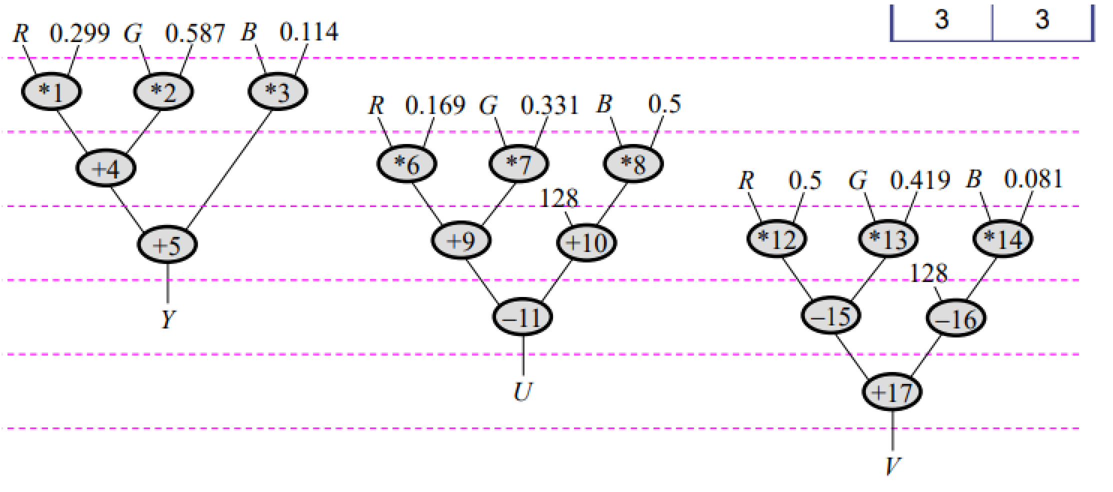
DFG2



Resource constraints:

*	+	
1	1	DFG2_outcome1_1.txt
1	2	DFG2_outcome1_2.txt
2	1	DFG2_outcome2_1.txt
2	2	DFG2_outcome2_2.txt
3	1	DFG2_outcome3_1.txt
3	2	DFG2_outcome3_2.txt
1	3	DFG2_outcome1_3.txt
2	3	DFG2_outcome2_3.txt
3	3	DFG2_outcome3_3.txt

☐ 附錄



*	+
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

RGB_outcome1_1.txt
RGB_outcome1_2.txt
RGB_outcome1_3.txt
RGB_outcome2_1.txt
RGB_outcome2_2.txt
RGB_outcome2_3.txt
RGB_outcome3_1.txt
RGB_outcome3_2.txt
RGB_outcome3_3.txt