CAD for VLSI Design
Programming Assignment 2 Report

Student:黃柏燁
Student ID:109521018

I. Introduction

現今的 VLSI Design，電路中 transistor 的數量非常龐大，傳統手繪 layout 已無法應付現今龐大的電路設計。

EDA(Electronic Design Automation)，藉由電腦輔助介入，自動合成出相對應的 layout。Scheduling 是 EDA 領域中重要的一環，藉由 Scheduling，可以將電路合成出更小的 latency 或使用更少的 resource 讓 layout area cost 更小。

本次作業主要是要求使用 C/C++實作出 resource-constrained scheduler，scheduler 輸入一個欲 schedule 的電路 data flow graph(如下圖)

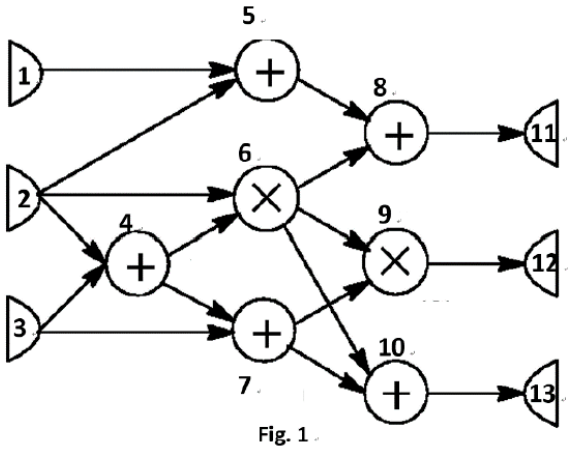


Fig. 1

並且轉出 schedule 後的 timing chart 以及 total latency(如下圖)

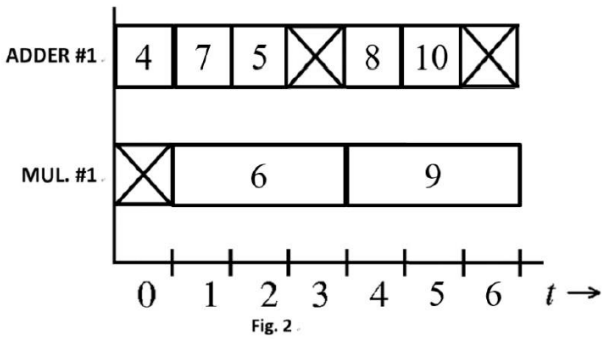


Fig. 2

II. Data Structure

因為 data flow graph 中每個點代表的意義相當多(number、type …)，所以我用一個 structure 建立了一個 double linked-list 來存取 parser 進來的 input 資訊，下圖為我建立的 struct node。

```
typedef struct node
{

int number;
int latency;
int priority;
int state;
int ongoing;
string type;
vector<node*> predecessor;
vector<node*> successor;
node *nextptr;
node *previousptr;

}node;
```

number: node 的數字
latency: node operation 所需 run time
priority: node 在 schedule 時的優先度
state: node 在每個 clock period 的狀態
(1: ready、0: unready、3: scheduled、2: ongoing)
ongoing: node operation 結束時的 clock period
type: node 的 operation type
predecessor: 每個 node 的祖先在 linked-list 中的位置
successor: 每個 node 的子孫在 linked-list 中的位置
nextptr: 指到下一個 node 位址
previousptr: 指到上一個 node 位址

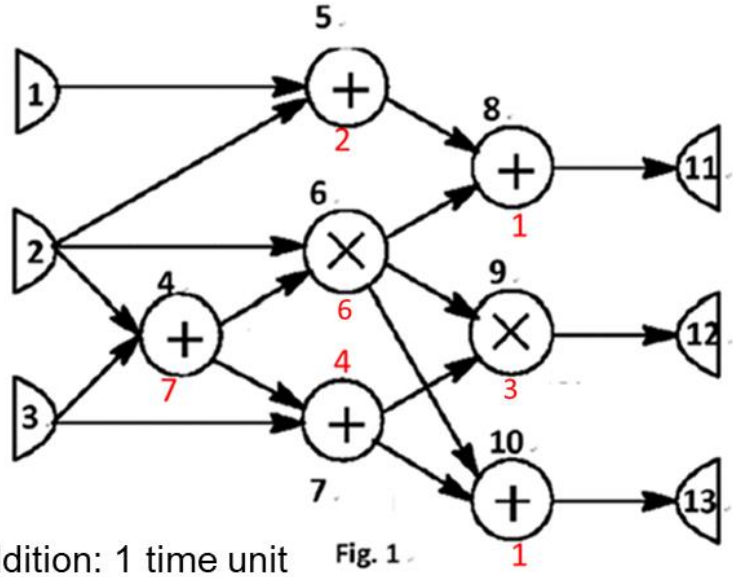
當我的 file parser 進來時，考慮到 input file 可能不一定會把 node type 按順序排好，存入 vector 之後，我將 i type 和 o type 分別排在 node type 最前面和最後面，以便後面的運算。

III. Algorithms

由於題目規定 scheduler 需為 resource-constrained，所以我採用 List

Scheduling Algorithm (ML-RCS)。

List Scheduling，是先將每個 node 的 priority 先決定，以 Fig .1 為例，從 output 往 input 計算得到每個點的 priority，如下圖。



Addition: 1 time unit Fig. 1

Multiplication: 3 time units

決定好 node 的 priority 之後，將所有 node 按照 priority 大小排成一個 list，再根據排好的 list 和 resource constrain 去 schedule 每個 clock period 的 operation，如下圖。

Node(ADD)	4	7	5	8	10
Priority	7	4	2	1	1

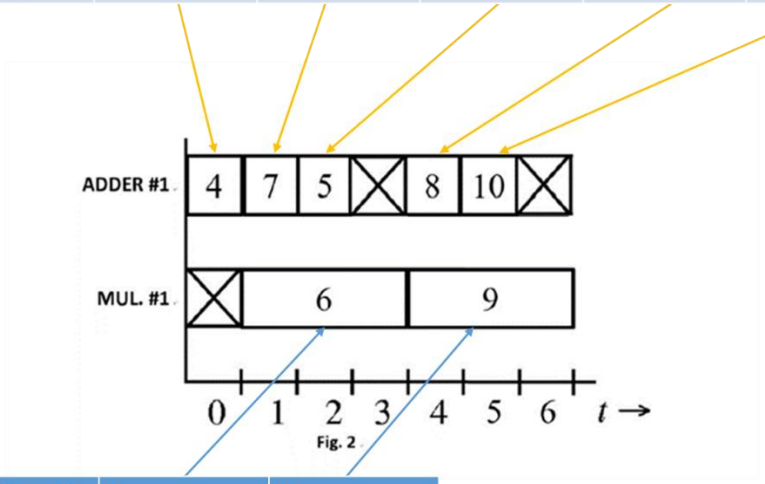


Fig. 2

Node(MUL)	6	9
Priority	6	3

在 schedule 的過程中，必須去檢查每個 node 的 predecessor，欲 schedule node 不能和他的 predecessor 在同一個 period，否則會產生 function failure。

IV. Implementation

A. Priority

Function prototype: void priority_list (node *lptr)

將指到 linked-list 最後一個位置的 pointer 放入，由於 priority 是由 output 端往 input 端計算，故在程式中我利用 struct node 中的*previousptr 當作下一個 node 索引的目標。

第一部分:

```
int n_p;

node *current = lptr;

while(current != NULL)
{

    if(current->type == "o")
    {
        for(int i = 0; i < current->predecessor.size(); i++)
        {
            lptr = current->predecessor[i];
            lptr->priority = lptr->latency;
        }
    }

    else
      break;

    current = current->previousptr;
    lptr = current;
}
```

此部分是先將 linked-list 中 node type 為 output 的 node 取出，將 output node's predecessor 計算出來，因 output port 不列入考慮，所以 predecessor node's priority 只等於本身的 run time。

第二部分:

```
node *temp;
int t = 0;

while(lptr != NULL)
{
    int counter = 0;

    for(int i = 0; i < lptr->successor.size(); i++)
    {
         temp = lptr->successor[i];
         n_p = lptr->latency + temp->priority;

         if(n_p > lptr->priority)
         {
            lptr->priority = n_p;
         }
    }
```

```
if(lptr->previousptr->type == "i")
{
    lptr = current;
    temp = lptr;

    while(temp->type != "i")
    {
        if(temp->priority == 0)
        {
            counter++;
        }

        temp = temp->previousptr;
    }

    if(counter == 0)
    {
        if(t == 2)
            break;
        t++;
    }
}
else
{
    lptr = lptr->previousptr;
}
}
```

此部分是藉由不斷搜索所有的 node 直到所有 node priority 都被決定為止，中間的 while loop 是負責用來檢查所有 node 是否都已被賦值，若有 node's priority 尚未被決定，則 counter > 0，不跳出外面的迴圈繼續搜索；反之，則 counter == 0，達成跳出迴圈的條件。

由於搜索 node 決定 priority 的順序並非固定是由靠近 output node 往靠近 input node，有可能會導致以下問題:

以下圖為例(假設搜索的順序為 7 -> 4 -> 6)

7 -> 4

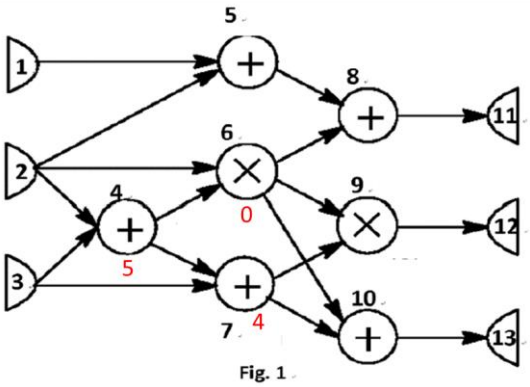


Fig. 1

4 -> 6

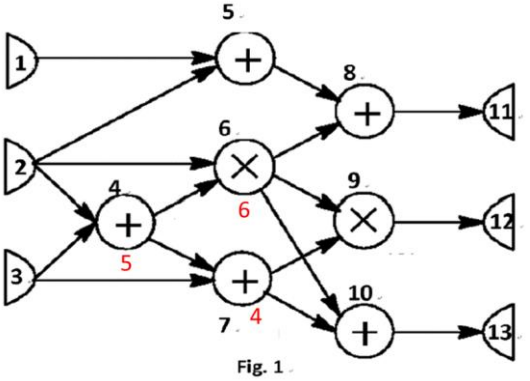


Fig. 1

因為上述的搜索順序，導致在決定 node 4's priority 時，node 6 尚未被決定，這時得到 node 4's priority 並非考慮十分周全。因此我在第一次所有 node's priority 被決定時(t == 1)，強制再做一次搜索(t++)，以避免上述情況生。

B. List Scheduling Algorithm (ML-RCS)

Function prototype: void list_l (node *ptr, vector<string> dfg, int num_node, char *argv)

第一部分:

```
while(ptr->type != "o")
{
    if((ptr->state == 2 && time != ptr->ongoing) || ptr->state == 3)
    {
        if(ptr->state == 2)
        {
            scheduled_o = scheduled_o + to_string(ptr->number) + " ";
            duplicate_count++;
        }
    }
    else if((ptr->state == 2 && time == ptr->ongoing) || ptr->type == "i")
    {
        ptr->state = 3;
    }
    else
    {
        for(int i = 0; i < ptr->predecessor.size(); i++)
        {
            pre = ptr->predecessor[i];

            if(pre->state == 3)
            {
                ptr->state = 1;

            }
            else
            {
                ptr->state = 0;
                break;
            }
        }

        if(ptr->state == 1)
        {
            ready.push_back(ptr);
        }
    }
```

此部分為決定在此時的 period 中，每個 node 是否已經 ready to schedule，必須考慮到 node's predecessor 是否全部為 scheduled node，若有 node 尚未被 scheduled 或者還在 ongoing，則將 node state 保持為 unready(state = 0)；反之，則更改至 ready(state = 1)，最後再把此 period 中所有 ready node push 進去 vector<*node>ready 之中。

第二部分:

```
for(int i = 0; i < multiplier-duplicate_count; i++)
{
   for(int j = 0; j < ready.size(); j++)
   {

      ptr = ready[j];

      if(ptr->type == "*" && p_num < ptr->priority && ptr->state == 1)
      {
          p_num = ptr->priority;
          s_node = ptr;
      }

   }

   if(p_num == 0)
   {
       break;
   }
   else
   {
      scheduled_m = scheduled_m + to_string(s_node->number) + " ";
      s_node->state = 2;
      s_node->ongoing = time + s_node->latency;
      p_num = 0;
   }
}

for(int i = 0; i < adder; i++)
{
   for(int j = 0; j < ready.size(); j++)
   {
      ptr = ready[j];

      if(ptr->type == "+" && p_num < ptr->priority && ptr->state == 1)
      {
          p_num = ptr->priority;
          s_node = ptr;
      }

   }

   if(p_num == 0)
   {
       break;
   }
   else
   {
       scheduled_a = scheduled_a + to_string(s_node->number) + " ";
       s_node->state = 3;
       p_num = 0;
   }
```

此部分將 ready node schedule 至此 period，比較所有 ready node's priority 和根據 resource constrain，從所有 ready node 中由 priority 大到小選擇 node schedule 至此 period，並且將此 node's state 更改為 scheduled(state = 3)。

第三部分:

```
ptr = first;
int counter = 0;

while(!ptr->successor.empty())
{
    if(ptr->predecessor.empty())
    {
        ptr = ptr->nextptr;
        continue;
    }

    if(ptr->state != 3)
    {
        counter++;
    }

    ptr = ptr->nextptr;
}


if(counter == 0)
    break;
```

此部分在檢查是否所有的 node 都被 scheduled，若還有 node 尚未被 scheduled(counter > 0)，則繼續往下一個 period schedule；反之(counter == 1)，則跳出迴圈，結束 scheduling。

V. Makefile

```
# Compiler
CXX = g++

# Path
SRC_PATH = src
BUILD_PATH = build

# Executable
EXE = go

# Source
SOURCES = $(SRC_PATH)/main.cpp
OBJECTS = $(BUILD_PATH)/main.o

# Compiler flags
CXXFLAG = -O3 -Wall
INCLUDE = -I$(SRC_PATH)

# Make-command list
.PHONY: all run clean

# Target: Dependencies
#       Shell-command
all: $(BUILD_PATH) $(EXE)


run: $(EXE)
        ./go $(INPUT) $(OUTPUT)

clean:
        @echo "Removing objects"
        @rm -rf $(BUILD_PATH)
        @echo "Removing executable file"
        @rm -rf $(EXE)

$(EXE): $(OBJECTS)
        @echo "Generating executable file: $^ -> $@"
        @$(CXX) $^ -o $@

$(BUILD_PATH)/%.o: $(SRC_PATH)/%.cpp
        @echo "Compiling: $< -> $@"
        @$(CXX) -std=c++11 $(CXXFLAGS) $(INCLUDE) -c $< -o $@

$(BUILD_PATH):

        @echo "Creating object directory"
        @mkdir -p $@

~
```

VI. Hardest part of the assignment

在實作本次作業時，我在很多地方都遇到問題，尤其是一開始決定 data structure 為 linked-list 後，對於大學沒學過資料結構的我來說，確實在一開始有點苦手。我花了很多時間爬了很多論壇，查了很多資料才能勉勉強強寫出一個簡單的架構。

在實現 algorithm 時，雖然有演算法的 pseudo code，但是要寫成 code 反而寫不出來的困境。透過不斷地詢問助教以及用紙畫出 data flow，才慢慢地在程式中實現 algorithm。