EE6094 CAD for VLSI Design

Programming Assignment 2: Scheduling

Student ID: 108501023
Student Name: 李品賢

Compile, Execute and Verification

1. Pull the source code, i.e., *108501023_PA2.cpp*, *Makefile*, *testcase1*, *testcase2*, *testcase3* and *checker* into the workstation folder.

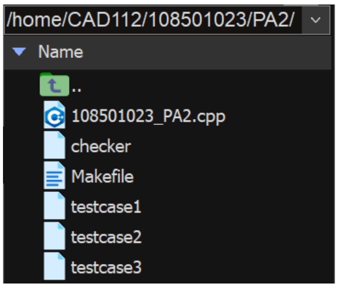


Fig. 1

2. Use *Makefile* as a trigger point to run the *108501023_PA2.cpp* program, and then the output *testcase1.out* / *testcase2.out* / *testcase3.out* are generated.

```
- make all
- make run Testcase=testcase1
- make run Testcase=testcase2
- make run Testcase=testcase3
- make clean
```

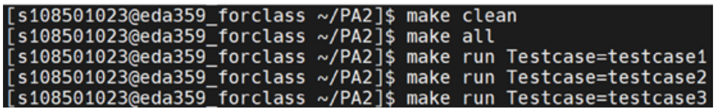


Fig. 2

3. Use checker to check whether output files fits the standard output format. The screenshots of the command are included in completion part.

```
- ./checker testcase1 testcase1.out
- ./checker testcase2 testcase2.out
- ./checker testcase3 testcase3.out
```

Completion

All three testcases are successfully passed the checker, the screen shows three cute "Nyan Cat". The following three figure (Fig. 1, Fig. 2, Fig. 3) are the results.

	testcase1	testcase2	testcase3
Completion	O	O	O
Hardware	4	19	1393

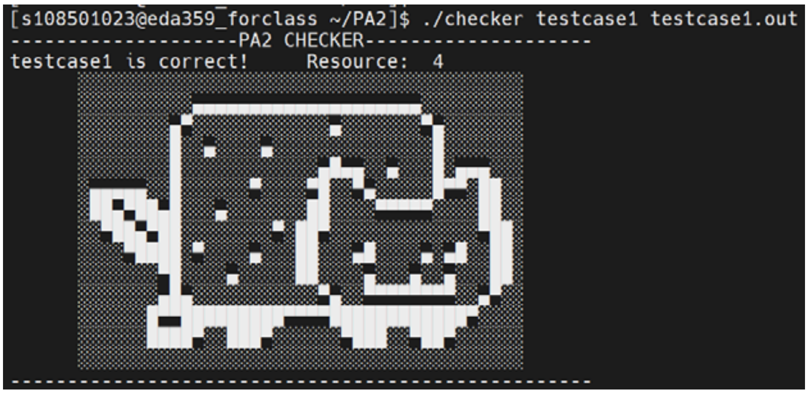


Fig. 3

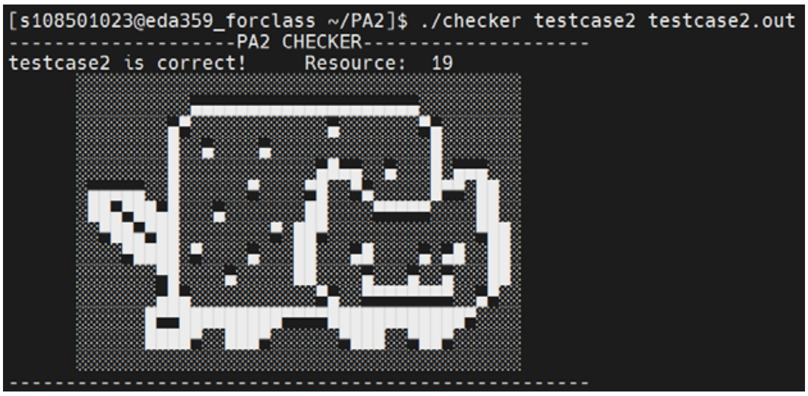


Fig. 4

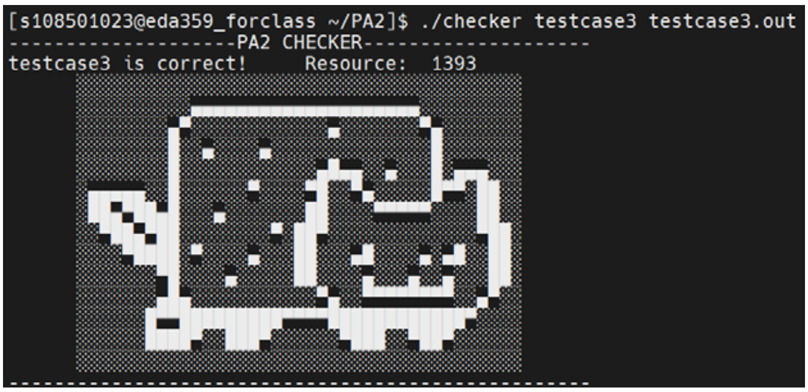


Fig. 5

Data Structure and Algorithm

Data structure:

I use 2-D linked-list array (left-side of Fig. 4) to represent a graph (right-side of Fig. 4). The grandparent relationship between each node can be included in this representation. The greatest benefit of this structure is that it takes fewer time to implement topology sort either from the top or the bottom, both of which are the algorithms included in this program.

For the starter, this structure actually represents a matrix. (Fig. 5) Each row of the matrix represents the out-degree of the vertex, e.g., matrix[0][1] is equal to 1, which means that vertex 0 points to vertex 1. In contrast, each column represents the in-degree of the vertex.

Headnodes and nodes are used. The x-axis headnodes record the successor relationship while the y-axis headnodes record the predecessor relationship. Nodes record edge information and point to another node by the according predecessor or successor relationship.

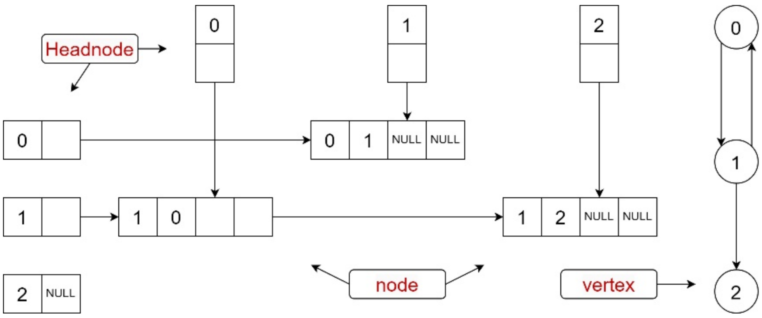


Fig. 6

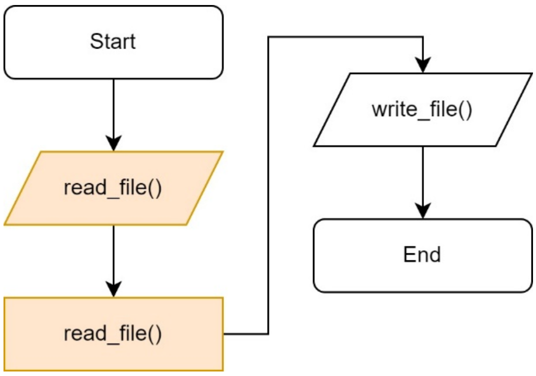
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

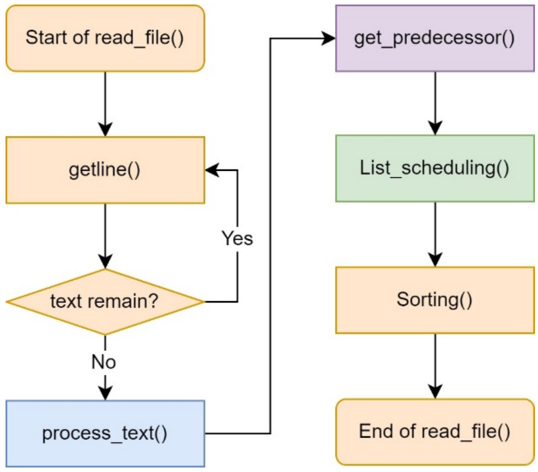
Fig. 7

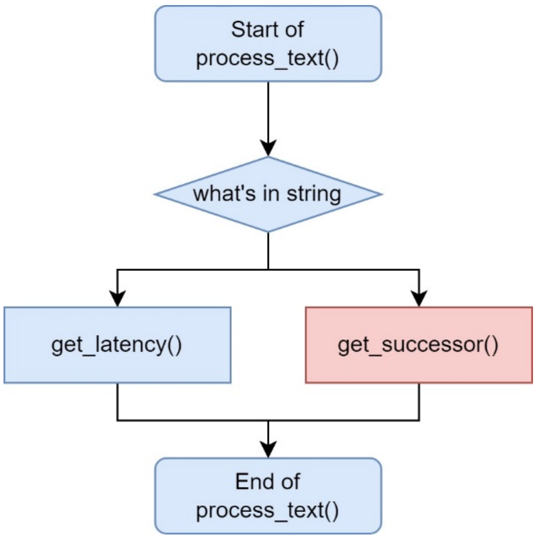
Algorithm:

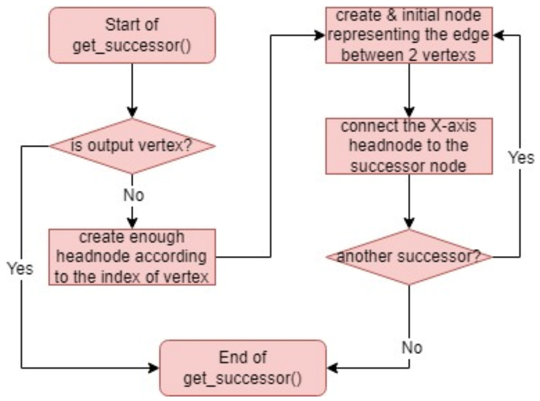
I use list scheduling algorithm to complete this assignment. The other algorithm inside the list scheduling algorithm is ALAP algorithm which use the concept of topology sort. After ALAP algorithm executed, the algorithm of scheduling part also uses the concept of topology sort. The difference between two scenarios is mentioned above, which is that ALAP algorithm apply topology order staring from bottom, while the other use that starting from the top.

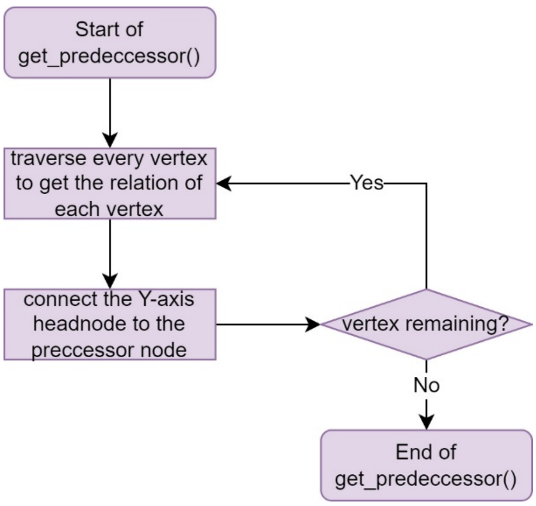
Flow Chart

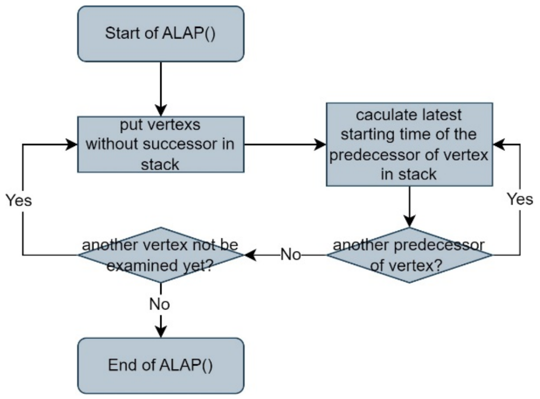

Start
read_file()
read_file()
write_file()
End

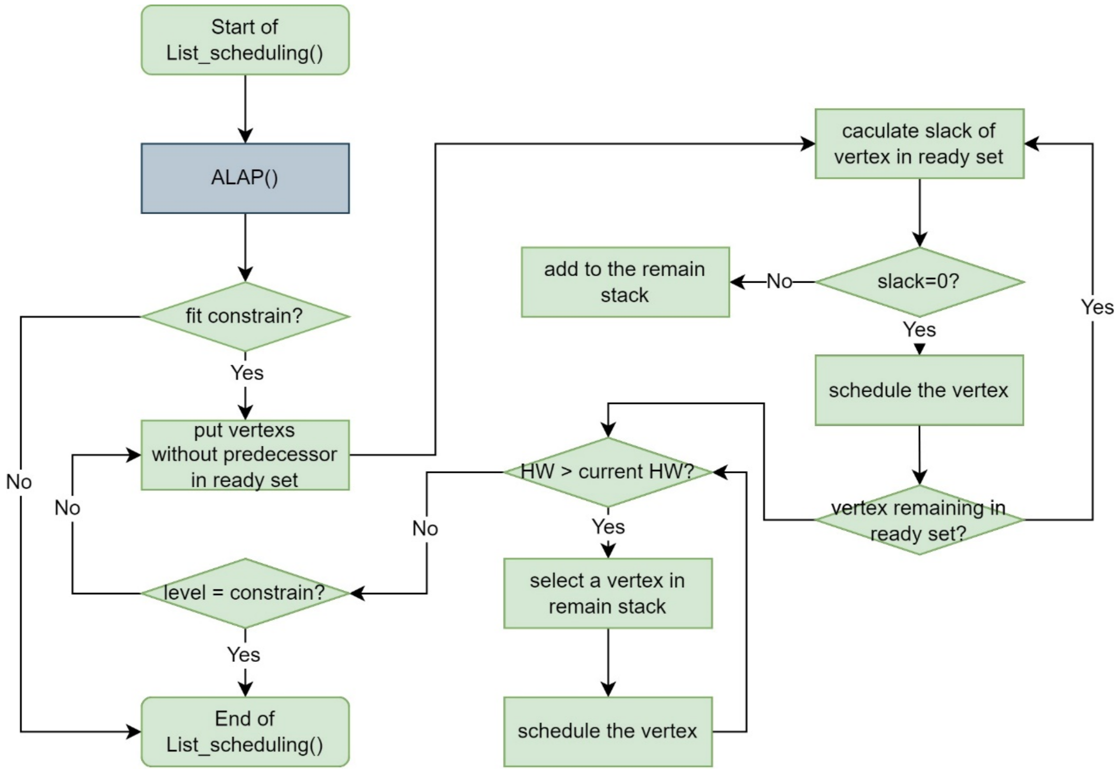

Start of read_file()
getline()
text remain?
Yes
No
process_text()
get_predecessor()
List_scheduling()
Sorting()
End of read_file()


Start of process_text()
what's in string
get_latency()
get_successor()
End of process_text()


Start of get_successor()
is output vertex?
No
Yes
create enough headnode according to the index of vertex
create & initial node representing the edge between 2 vertexs
connect the X-axis headnode to the successor node
another successor?
Yes
No
End of get_successor()


Start of get_predeccessor()
traverse every vertex to get the relation of each vertex
connect the Y-axis headnode to the preccessor node
vertex remaining?
Yes
No
End of get_predeccessor()


Start of ALAP()
put vertexs without successor in stack
caculate latest starting time of the predecessor of vertex in stack
another predecessor of vertex?
Yes
No
another vertex not be examined yet?
Yes
No
End of ALAP()



Source Code Explanation

Data structure:

1. struct node

   The *struct node* structure (Fig. 6) stores the edge information (*int suc, pre*). The variable *int pre* represents the vertex closer to the top, while the variable *int suc* represents the vertex closer to the bottom. The variable *int cycle* stores execution time of the vertex *"pre"*. Also, because it is the node in 2-D linked-list array, potiners to the successor and predecessor are included (*struct node *sucPtr, *prePtr*).

```
struct node {
    int suc, pre, cycle;
    struct node *sucPtr, *prePtr;
};
typedef struct node Node;
typedef Node *NodePtr;
```

Fig. 8

2. struct vertexInfo

   The *struct vertexInfo* structure (Fig. 7) stores the vertex information including latest starting time (*int latest*) for ALAP algorithm, successor counts (*int sucCount*), predecessor counts (*int preCount*), operation types (*char oper*) and valid bits (*bool valid*) for recognizing whether this vertex is included in input file and for representing the scheduled vertex.

```
struct vertexInfo {
    int latest, sucCount, preCount;
    char oper;
    bool valid;
};
typedef struct vertexInfo VertexInfo;
```

Fig. 9

Variable:

1. int latency

   store the latency constrain
2. int ADDCount, MULCount, vertexNum

   store the total adder count, multiplier count and vertex count.
3. string line

   catch the text of input file line by line
4. vector<NodePtr> sucHead, preHead

   headnodes for the 2-D linked-list array. *sucHead* is a x-axis header for row, while y-axis *preHead* is for column.
5. vector<VectorInfo> vertex

   use a vector as a container to store the vertex information
6. vector< vector<int>> output

   use a 2-D vector to store the scheduled vertex number in each level
7. vector< vector<int>> MULstate

   record the index of specific vertexes with type '*' and executing cycle of it.

Function:

1. void read_file()

   read input file.
2. void write_file()

   write output file.
3. void process_text()

   process text including latency and vertex information from input line.
4. int str_to_int()

   turn the string to the integer.
5. void get_successor()

   construct the 2-D structure with horizontal direction according to the incoming vertex information.
6. void get_ predecessor ()

   construct the 2-D structure with vertical direction according to information of current 1-D table.
7. int get_latency()

receive the latency constrain.

8. void extend_vertex_count()

   create enough headnode according to the incoming vertex number.

9. void create_node(int, int)

   create a node in 2-D structure. Arguments of function are both end of edge.

10. bool ALAP()

    implement ALAP algorithm

11. void List_Scheduling()

    implement list scheduling algorithm.

12. void schedule_vertex(int, int, char, vector<int> &)

    If the condition is meet, then schedule the specific vertex. Arguments of function are index of vertex, level of time step, operation type and vector MULstate.

13. decrement_preCount(int)

    called at the end of execution cycle of specific vertex, i.e., first cycle of ‘i’ vertex, first cycle of ‘+’ vertex and the third cycle of ‘*’ vertex.

14. void sorting()

    sort the vertex number in ascending order in each time step.

```
class Scheduling {
public:
    Scheduling();
    void read_file(fstream &);
    void write_file(fstream &);

private:
    int latency, ADDCount, MULCount, vertexNum;
    string line;
    vector<NodePtr> sucHead, preHead;
    vector<VertexInfo> vertex;
    vector<vector<int>> output;
    vector<vector<int>> MULstate;

    void process_text();
    int str_to_int(string);
    void get_predecessor();
    int get_latency();
    void get_successor();
    void extend_vertex_count();
    void creat_node(int, int);
    bool ALAP();
    void List_Scheduling();
    void schedule_vertex(int, int, char, vector<int> &);
    void decrement_preCount(int);
    void sorting();
};
```

Fig. 10

Code:

In the code explanation, I will explain the most important part in my source code, such as main(), *process_text()*, *get_predecessor()*, *get_latency()*, *get_successor()*, *ALAP()*, *List_Scheduling()*, *schedule_vertex()* and *decrement_preCount()*.

1. int main()

   The function of main function is to trigger the entire program, so it only contains the function of read input file and write output file.

```
int main(int argc, char *argv[]) {

    fstream myFile;
    Scheduling S;
    string intputFile = argv[1];
    string outputFile = intputFile + ".out";

    myFile.open(intputFile, ios::in);
    if(!myFile) {
        cerr << "Error opening file" << endl;
        return 1;
    }
    else {
        S.read_file(myFile);
        myFile.close();
    }

    myFile.open(outputFile, ios::out);
    S.write_file(myFile);
    myFile.close();

    return 0;
}
```

Fig. 11

2. void read_file()

   There are four step in this function. For the starter, it catchs the line string of input file through *getline()*, and call *process_text()* to get either the latency constrain or the vertex information so as to construct the 1-D linked-list array.

   Then, it calls *get_predecessor()* to reconstruct 1-D linked-list array to one with 2-D.

   Furthermore, it implements the scheduling by calling *List_Scheduling()*.

   Finally, sorting the output the vertex number in ascending order in each time step is the last step by calling *sorting()*.

```
void Scheduling::read_file(fstream& myFile) {

    // 1. read the file line by line, construct the 1-D data structure
    // 2. reconstruct it from 1-D to 2-D data structure
    // 3. do the list scheduling algorithm
    // 4. sort each output line in ascending order

    while (getline(myFile, line))
        process_text();

    get_predecessor();
    List_Scheduling();
    sorting();
}
```

Fig. 12

3. void process_text()

If the string “Latency constrain” is detected within input line string, it will receive the latency by calling *get_latency()*. I will not explain this function due to its straightforward implementation.

Also, if the starting character of input line string is a integer number, it calls *get_succerror()*, which construct the 1-D array by the vertex information from the input line string.

```
void Scheduling::process_text() {

    // get the latency constrain and each node infomation

    if(line.find("Latency constrain") != string::npos){
        vector<int> temp;

        latency = get_latency();

        for(int i = 0; i < latency+1; i++)
            output.push_back(temp);
    }
    else if(line[0] >= 48  && line[0] <= 57)
        get_successor();
}
```

Fig. 13

4. void get_successor()

If the incoming vertex is not output vertex and there is any successor according to the line string, then it enters the while loop (code 158).

The loop does two main things, one is creating enough headnodes according to the incoming vertex number (code 163-165), another one is creating the node in linked-list array, using x-axis headnode point to it and increasing the successor count of the incoming vertex by 1 (code 167-168).

After end of loop, it sets the valid bit of incoming vertex and stores the operation type of it (code 171-172).

```
146 void Scheduling::get_successor() {
147 
148     // process string to constructure 1-D data structure
149     // each sucHead[vertex_index] will point to its successors
150 
151     NodePtr temp = NULL;
152     NodePtr NOP = NULL;
153     string suc_str = " ";
154     int pre = str_to_int(line);
155 
156     suc_str = line.substr(line.find(" ")+3, line.length()-line.find(" ")-3) + "  ";
157 
158     while(suc_str[0] != ' ' && line[line.find(" ")+1] != 'o') {
159 
160         int suc = str_to_int(suc_str);
161         suc_str =  suc_str.substr(suc_str.find(" ")+1, suc_str.length() - suc_str.find(" ")-1);
162 
163         if(sucHead.size() < suc+1)
164             for(int i = sucHead.size(); i < suc+1; i++)
165                 extend_vertex_count();
166 
167         vertex[pre].sucCount++;
168         creat_node(pre, suc);
169     }
170 
171     vertex[pre].valid = true;
172     vertex[pre].oper = line[line.find(" ")+1];
173 }
```

Fig. 14

5. void get_predecessor()

The for loop (code 214) examines every x-axis headnode to receive the edge information.

Then, the while loop (code 218) does three things, one is increasing the predecessor count of the vertex closer to the bottom by 1. Another is assigning the delay time in each node (code 219-226). The delay time stored in each node represents the execution time of the vertex closer to the top ("*pre*") in edge relation. The other is connecting nodes in 1-D table to y-axis headnode (code 228-230)

```
204 void Scheduling::get_predecessor() {
205
206     // reconstructure it to 2-D data structure
207     // each preHead[vertex_index] will point to its predeccessors
208     // e.g. edge(0, 1) will be pointed by sucHead[0] and preHead[1]
209
210     NodePtr temp;
211
212     vertexNum = sucHead.size()-1;
213
214     for(int i = 1; i <= vertexNum; i++) {
215
216         temp = sucHead[i];
217
218         while(temp) {
219             vertex[temp->suc].preCount++;
220
221             if(vertex[i].oper == '+')
222                 temp->cycle = 1;
223             else if(vertex[i].oper == '*')
224                 temp->cycle = 3;
225             else if(vertex[i].oper == 'i')
226                 temp->cycle = 1;
227
228             temp->prePtr = preHead[temp->suc];
229             preHead[temp->suc] = temp;
230             temp = temp->sucPtr;
231         }
232     }
233 }
```

Fig. 15

6. bool ALAP()

The first for loop (code 242) searches all vertex to find the valid one with 0 successor count, and put them into the stack.

The second for loop (code 246) starts calculating latest starting time of each vertex with 0 successor count.

The for loop (code 250) inside second loop examines every predecessor of the vertex. First, it decrements the successor count of predecessor by 1 (code 252), and then put it into stack if its successor count is 0 (code 254-255). Because this is a ALAP algorithm, the predecessor chooses the minimum value of difference between it and its successor (code 257-259). By the way, the initial value of latest starting time of each node is latency+1. Notice that if there is a latest starting time of predecessor is less than 0, *ALAP()* return false, which means the scenario of this input file is impossible to solve in this latency constrain (code 261-262).

If it finally reach the end of *ALAP()*, it return true (code 266).

```
235 bool Scheduling::ALAP() {
236
237     // use topology sort to get latest starting time of each vertex
238
239     stack<int> s;
240     int index;
241
242     for(int i = 1; i <= vertexNum; i++)
243         if(!vertex[i].sucCount && vertex[i].valid)
244             s.push(i);
245
246     for(int i = 1; i <= vertexNum; i++) {
247         index = s.top();
248         s.pop();
249
250         for(NodePtr ptr = preHead[index]; ptr; ptr = ptr->prePtr) {
251
252             vertex[ptr->pre].sucCount--;
253
254             if(!vertex[ptr->pre].sucCount && vertex[i].valid)
255                 s.push(ptr->pre);
256
257             if(vertex[index].latest - ptr->cycle < vertex[ptr->pre].latest){
258
259                 vertex[ptr->pre].latest = vertex[index].latest - ptr->cycle;
260
261                 if(vertex[ptr->pre].latest < 0)
262                     return false;
263             }
264         }
265     }
266     return true;
267 }
```

Fig. 16

7. void List_Scheduling()

First, it calls *ALAP()*. If *ALAP()* returns false, then this function is unnecessary to execute, and prints the waring statement.

```
void Scheduling::List_Scheduling() {

    // if slack < 0 while doing ALAP, can't be scheduling
    // using list scheduling algorithm to schedule vertexs

    if(!ALAP()) {
        cout << "It's impossible to solve in this latency constrain" << endl;
        return;
    }

```

Fig. 17

If *ALAP()* return true, then it starts to schedule every vertex in each time step.

The first for loop (code 289) examines every time level until level is equal to latency. The level is used to calculate the slack later.

The for loop (code 291) inside first for loop searches all vertex to find the valid one with 0 predecessor count, and put them into the “ready queue”.

The for loop (code 297) examines every vertex in “ready queue” and calculates the slack which is the difference of latest starting time of vertex and level (code 302). Afterwards, if it's input vertex, there is nothing to do, it calls *decrement_preCount()* directly (to decrease all successor's *preCount*) (code 304-305). Continued, if it's slack is 0, it calls *schedule_vertex()* accordingly to do some operation, which will be explained in the text later (code 306-311). Otherwise, the vertex should be push into "remain stack" (code 312-313).

```
279     queue<int> ready;
280     stack<int> remain;
281     int index, slack;
282     int readyCount = 0, remainCount = 0;
283     vector<int> currentADDCount(latency+1, 0);
284     vector<int> currentMULCount(latency+1, 0);
285 
286     // vertex of preCount==0 and valid==true is chosen to ready queue
287     // notice that the executing '*' vertex is with false valid bit
288 
289     for(int level = 0; level <= latency; level++) {
290 
291         for(int i = 1; i <= vertexNum; i++)
292             if(!vertex[i].preCount && vertex[i].valid)
293                 ready.push(i);
294 
295         readyCount = ready.size();
296 
297         for(int i = 0; i < readyCount; i++) {
298 
299             index = ready.front();
300             ready.pop();
301 
302             slack = vertex[index].latest - level;
303 
304             if(vertex[index].oper == 'i')
305                 decrement_preCount(index);
306             else if(slack == 0) {
307                 if(vertex[index].oper == '+')
308                     schedule_vertex(index, level, '+', currentADDCount);
309                 else if(vertex[index].oper == '*')
310                     schedule_vertex(index, level, '*', currentMULCount);
311             }
312             else
313                 remain.push(index);
314         }
```

Fig. 18

The for loop (code 320) is to select those vertexes originally in "ready queue", but now is in "remain stack" because of its nonzero slack. Then, it does the same scheduling procedure as same as the above, only if there are spare hardware left unused (code 325-328).

```
        // vertex of preCount==0, valid==true and slack!=0 is chosen to remain stack

        remainCount = remain.size();

        for(int i = 0; i < remainCount; i++) {

            index = remain.top();
            remain.pop();

            if(vertex[index].oper == '+' && ADDCount > currentADDCount[level])
                schedule_vertex(index, level, '+', currentADDCount);
            else if(vertex[index].oper == '*' && MULCount > currentMULCount[level])
                schedule_vertex(index, level, '*', currentMULCount);
        }
```

Fig. 19

Then, the for loop (code 332) examines all executing '*' vertex. It first decreases cycles by 1, and then check if it is equal to 0. If it is 0, then it call *decrement_preCount()* (to decrease all successor's *preCount*) (code 333-337) so that this vertex will not include in new MULstate vector. If it is not 0, just add to new MULstate.

```
331        vector<vector<int>> temp_;
332        for(int i = 0; i < MULstate.size(); i++) {
333            if(--MULstate[i][1] == 0)
334                decrement_preCount(MULstate[i][0]);
335            else
336                temp_.push_back(MULstate[i]);
337        }
338        MULstate = temp_;
```

Fig. 20

Afterward, if the current used hardware is more than original one, update it.

```
        if(ADDCount < currentADDCount[level])
            ADDCount = currentADDCount[level];
        if(MULCount < currentMULCount[level])
            MULCount = currentMULCount[level];
```

Fig. 21

8. void schedule_vertex(int index, int level, char oper, vector<int> currentCount)

If it's '+' vertex, then it is added to output vector. Also, the current adder count will be increased by 1, and it call *decrement_preCount()* (to decrease all successor's *preCount*) (code 349-353).

If it's '*' operation, then it is added to output vector for 3 cycle (3 space of output vector). Also, the current adder count and the adder count within 3 cylcle will be increased by 1(code 355-359). Then, it adds the index of vertex and cycle number in to MULstate vector (code 360-363). Notice that the *decrement_preCount()* has not yet been called here, because it's not the end cycle of operation.

```cpp
void Scheduling::schedule_vertex(int index, int level, char oper, vector<int> &currentCount) {

    // schedule vertex according to its type
    // notice decrement_preCount() can't be called with '*' op instantly due to 3 cycle
    // but the valid unset first, thus it will not be selected to ready queue

    if(oper == '+') {
        output[level].push_back(index);
        currentCount[level]++;
        decrement_preCount(index);
    }
    else if(oper == '*') {
        for(int j = level; j < level+3; j++) {
            vertex[index].valid = false;
            output[j].push_back(index);
            currentCount[j]++;
        }
        vector<int> temp;
        MULstate.push_back(temp);
        MULstate.back().push_back(index);
        MULstate.back().push_back(3);
    }
}
```

Fig. 22

9. drecement_preCount(int index)

It is used to decrease all successor's *preCount* by traversing from headnode.

```cpp
void Scheduling::decrement_preCount(int index) {

    // called at the end cycle of operation

    vertex[index].valid = false;
    for(NodePtr ptr = sucHead[index]; ptr; ptr = ptr->sucPtr)
        vertex[ptr->suc].preCount--;
}
```

Fig. 23

Makefile

Because this project use single .cpp file, there is only one executable file created, i.e., *108501023_PA2.o*. The following is source code of Makefile.

```makefile
# PA2 cpp compiler

.PHONY: all run clean
all: 108501023_PA2.o
	@g++ -std=c++11 108501023_PA2.o -o exe
run:
	@./exe $(Testcase)
clean:
	@rm *.o
108501023_PA2.o: 108501023_PA2.cpp
	@g++ -std=c++11 -c 108501023_PA2.cpp
```

Fig. 24

Hardness

The hardest part is the lack of corresponding knowledge while receiving the PA2 document, which, however, is a great chance for me to learn myself. I do some research on google with the relate material, such as the reference from the PA2 document and the chapter 3 of class slide. In this project, I have to review the data structure course, since the data structure I used comes from the chapter 6 including the topology sort concepts and implementations. I am thankful that I have taken the data structure course helping me to come up with useful architecture in mind.

Moreover, I derive some rules with compiler on workstation. Last time, I found that *getline()* would take '/n' as a last input character with the .txt input file in PA1 while it would <u>not</u> doing so with the non-file-extension input file.

Suggestion

I am grateful for having this project, it helps me integrating data structure background into this project.