电子科技大学信息与软件工程学院

**实 验 报 告**

学 号 2023091602012

姓 名 邓奕雷

（实验） 课程名称 EDA软件设计II

理论教师 孙静翎

实验教师 易黎

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：邓奕雷 学号：2023091602012 指导教师：易黎**

**实验地点：信软楼西303 实验时间：2025.04.24**

**一、实验名称：实验一 ASAP和ALAP调度算法实验**

**二、实验学时：4**

**三、实验目的：**

1. 深入理解 ASAP（As Soon As Possible）和 ALAP（As Late As Possible）调度算法的原理和实现方法。
2. 掌握使用编程手段对电路描述文件（如 BLIF 文件）进行解析和处理的能力。
3. 通过对不同 BLIF 文件的解析和调度，验证算法的正确性和通用性。
4. 提升对电路调度问题的分析和解决能力，加深对 EDA（电子设计自动化）软件设计的理解。

**四、实验原理：**

1. **ASAP 调度算法**

ASAP 算法是一种尽早调度算法，其核心思想是尽可能早地开始每个操作，以使得整个电路的执行时间最短。在算法执行过程中，每个节点的调度开始时间取决于其所有前驱节点的完成时间，只有当前驱节点都完成后，该节点才能开始调度。

1. **ALAP 调度算法**

ALAP 算法是一种尽可能晚调度算法，它从电路的输出节点开始，反向计算每个节点的最晚开始时间。在算法执行过程中，每个节点的调度开始时间取决于其后继节点的开始时间，只有当所有后继节点都调度完成后，该节点才能开始调度。

1. **BLIF 文件解析**

BLIF（Berkeley Logic Interchange Format）是一种用于描述数字电路的文本文件格式。在本实验中，通过解析 BLIF 文件中的.names、.inputs、.outputs等关键字，提取电路的节点信息和连接关系，构建图结构表示电路。

**五、实验内容：**

* + 1. 假设电路中每种运算需要消耗的周期数都为1，编程实现ASAP和ALAP调度算法。要求程序对三个不同的blif文件进行解析，都能得到正确的输出。
    2. （选做）假设有以下周期数消耗约定，编程实现ASAP和ALAP调度算法。要求程序对三个不同的blif文件进行解析，都能得到正确的输出。



**六、实验器材（设备、元器件）：**

1. 开发环境：Visual Studio 2022
2. 测试文件：sample.blif

**七、实验步骤：**

根据实验原理，使用 C++ 语言实现 ASAP 和 ALAP 调度算法。代码主要包括以下几个部分：

* 1. **节点和图的定义：**
     1. 定义Node类表示电路中的节点，Graph类表示电路的图结构。
     2. 前驱/后继节点集：考虑到ASAP算法需要判断前驱节点是否完成调度，ALAP算法需要判断后继节点是否完成调度，所以需要定两个vector分别维护一个节点的前驱节点集和后继节点集。
     3. 调度开始周期：维护每个节点的调度开始周期，默认为-1（未调度）
     4. 调度执行时间：维护不同类型节点的调度执行时间，默认为1
  2. **BLIF 文件解析：**
     1. 实现processInput函数，解析 BLIF 文件，构建图结
     2. 读取.inputs、.outputs、.names定义，构建节点及依赖关系。
     3. 根据真值表判断门类型（AND/OR/NOT）。
  3. **ASAP 算法实现：**
     1. 实现ASAP函数，进行尽早调度。
     2. 初始化输入节点为周期0。
     3. 逐周期选择所有前驱节点均已完成调度的节点，标记其开始周期。
  4. **ALAP 算法实现：**
     1. 实现ALAP函数，进行尽可能晚调度。
     2. 根据ASAP结果确定总周期数requiredCycle。
     3. 逆序调度，从后往前选择所有后继节点的最早开始调度时间，减去当前节点的调度执行时间，作为当前节点的开始周期。
  5. **结果输出：**
     1. 输出各节点的调度周期及分类（AND/OR/NOT）。
     2. 对比不同文件的总周期数，验证逻辑正确性。

**八、实验结果与分析（含重要数据结果分析或核心代码流程分析）**

* 1. **节点和图的定义：**

|  |
| --- |
| class Node {  public:  // 后继节点  list<Node\*> next;  // 前驱节点  list<Node\*> pre;  // 节点类型  Type type;  // 节点名称  string name;  // 调度开始周期，-1表示未调度  int startCycle;  // 操作执行时间  int executionTime;  int getEndCycle() {  return startCycle + executionTime;  }  Node(string nodeName, int execTime = 1) : name(nodeName), type(DEFAULT), executionTime(execTime) {  startCycle = -1;  }  Node(int execTime = 1) : type(DEFAULT), executionTime(execTime) {  startCycle = -1;  }  };  class Graph {  public:  // 节点集  map<string, Node\*> nodes;  // 图的大小  int size;  // 已调度节点的数量  int processCount;  // 清空调度状态  void clear() {  for (auto& pair : nodes) {  pair.second->startCycle = -1;  }  processCount = 0;  }  // 初始化图  Graph() : size(0) {  size = 0;  processCount = 0;  }  }; |

* 1. **BLIF 文件解析：**

|  |
| --- |
| // 处理输入，初始化图  void processInput(Graph& g, vector<std::string> blifContent) {  int size = blifContent.size();  int index = 0;  while (index++ < size) {  string line = blifContent[index];  if (line.find(".names") != string::npos) {  // 提取节点名称  string names = line.substr(line.find(" ") + 1);  size\_t space;  // 输入节点  list<string> inputs;  while ((space = names.find(" ")) != string::npos) {  inputs.push\_back(names.substr(0, space));  names = names.substr(space + 1);  }  // 输出节点  string output = names;  // 创建输入节点  for (auto input : inputs) {  if (g.nodes.find(input) == g.nodes.end()) {  g.nodes[input] = new Node(input);  g.nodes[input]->name = input;  g.size++;  }  }  // 创建输出节点  if (g.nodes.find(output) == g.nodes.end()) {  g.nodes[output] = new Node(output);  g.nodes[output]->name = output;  g.size++;  }  // 添加前驱和后继关系  for (auto input : inputs) {  g.nodes[input]->next.push\_back(g.nodes[output]);  g.nodes[output]->pre.push\_back(g.nodes[input]);  }  // 判断节点类型  Type type;  // 偏移量  int offset = 1;  // 下一行  string nextLine = blifContent[index + offset];  while (index + offset < size) {  if (nextLine.find(".names") != string::npos || nextLine.find(".inputs") != string::npos ||  nextLine.find(".outputs") != string::npos || nextLine.find(".model") != string::npos ||  nextLine.find(".end") != string::npos) {  break;  }  // NOT字符串只有3列  if (nextLine.size() == 3) {  break;  }  offset++;  nextLine = blifContent[index + offset];  }  // NOT  if (offset == 1) {  type = NOT;  g.nodes[output]->executionTime = 1;  }  // AND  if (offset == 2) {  type = AND;  g.nodes[output]->executionTime = 2;  }  // OR  if (offset > 2) {  type = OR;  g.nodes[output]->executionTime = 3;  }  g.nodes[output]->type = type;  continue;  }  else if (line.find(".inputs") != string::npos) {  // 输入节点  string inputs = line.substr(line.find(" ") + 1);  size\_t space;  while ((space = inputs.find(" ")) != string::npos) {  string input = inputs.substr(0, space);  if (g.nodes.find(input) == g.nodes.end()) {  g.nodes[input] = new Node(input);  g.nodes[input]->type = INPUT;  g.nodes[input]->name = input;  g.size++;  }  inputs = inputs.substr(space + 1);  }  if (!inputs.empty()) {  if (g.nodes.find(inputs) == g.nodes.end()) {  g.nodes[inputs] = new Node(inputs);  g.nodes[inputs]->type = INPUT;  g.nodes[inputs]->name = inputs;  g.size++;  }  }  }  else if (line.find(".outputs") != string::npos) {  // 输出节点  string outputs = line.substr(line.find(" ") + 1);  size\_t space;  while ((space = outputs.find(" ")) != string::npos) {  string output = outputs.substr(0, space);  if (g.nodes.find(output) == g.nodes.end()) {  g.nodes[output] = new Node(output);  g.nodes[output]->type = OUTPUT;  g.nodes[output]->name = output;  g.size++;  }  outputs = outputs.substr(space + 1);  }  if (!outputs.empty()) {  if (g.nodes.find(outputs) == g.nodes.end()) {  g.nodes[outputs] = new Node(outputs);  g.nodes[outputs]->type = OUTPUT;  g.nodes[outputs]->name = outputs;  g.size++;  }  }  }  else if (line.find(".model") != string::npos) {  // 模型信息  std::cout << "模型名称：" << line.substr(line.find(" ") + 1) << endl;  }  else if (line.find(".end") != string::npos) {  break;  }  }  } |

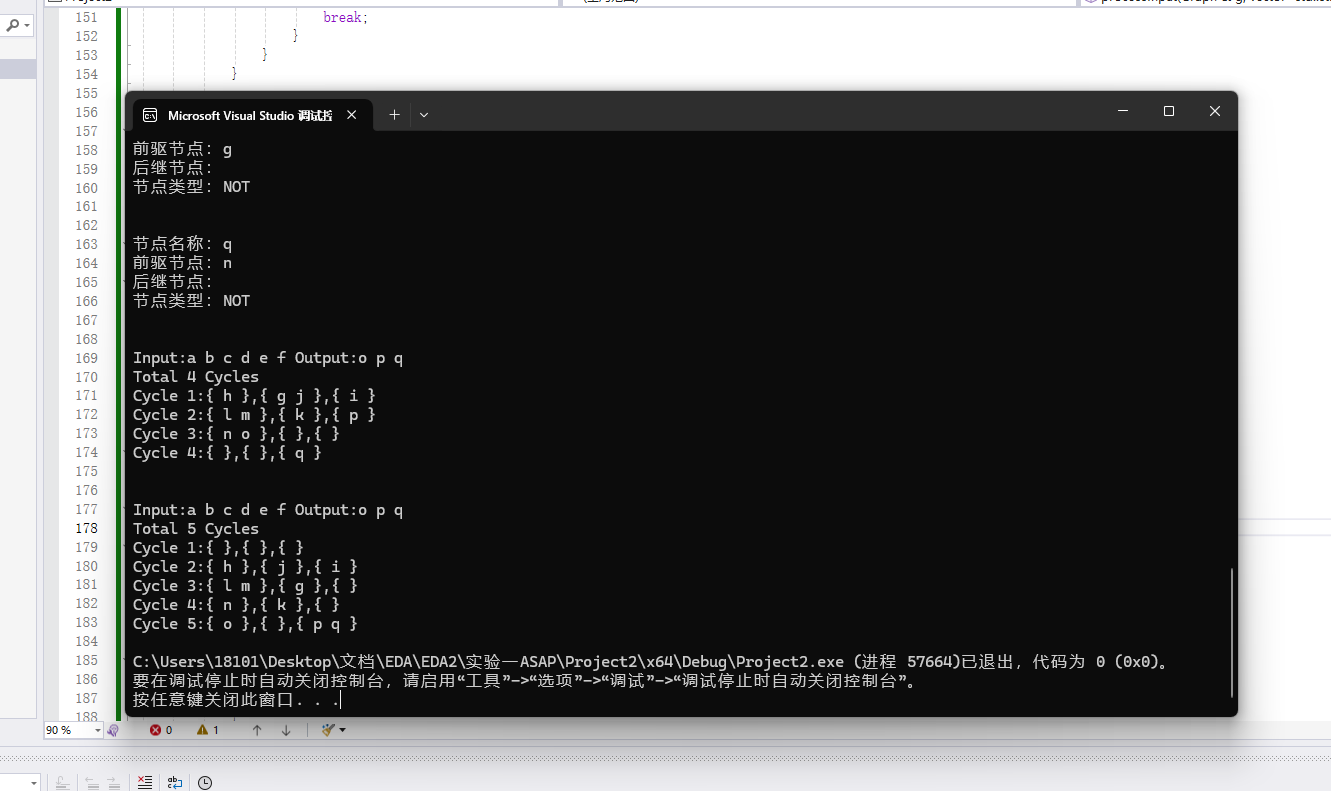
* 1. **ASAP 算法实现：**

|  |
| --- |
| // ASAP算法，返回周期数  int ASAP(Graph& g) {  // 每次循环是一个周期  int currentCycle = 0;  while (g.processCount < g.size) {  // 当前周期的调度节点集  vector<Node\*> processingNodes;  // 查找当前周期可以调度的节点  for (auto& pair : g.nodes) {  Node\* currentNode = pair.second;  // 跳过已调度节点  if (currentNode->startCycle != -1) {  continue;  }  // 表示前驱节点是否已全部调度过  bool preProcessFinished = true;  int maxPreEndCycle = 0;  // 检查非输入节点的前驱  if (!currentNode->pre.empty()) {  for (Node\* n : currentNode->pre) {  // 存在前驱节点未调度  if (n->startCycle == -1) {  preProcessFinished = false;  break;  }  // 更新最大前驱结束周期  maxPreEndCycle = max(maxPreEndCycle, n->getEndCycle());  }  }  // 只有当前驱节点全部调度完成，且当前周期大于等于最大前驱结束周期的节点才加入当前周期  if (preProcessFinished && currentCycle >= maxPreEndCycle) {  processingNodes.push\_back(currentNode);  }  }  // 调度当前周期的节点  if (!processingNodes.empty()) {  for (Node\* node : processingNodes) {  node->startCycle = currentCycle;  g.processCount++;  }  }  currentCycle++;  }  return currentCycle - 1;  } |

* 1. **ALAP 算法实现：**

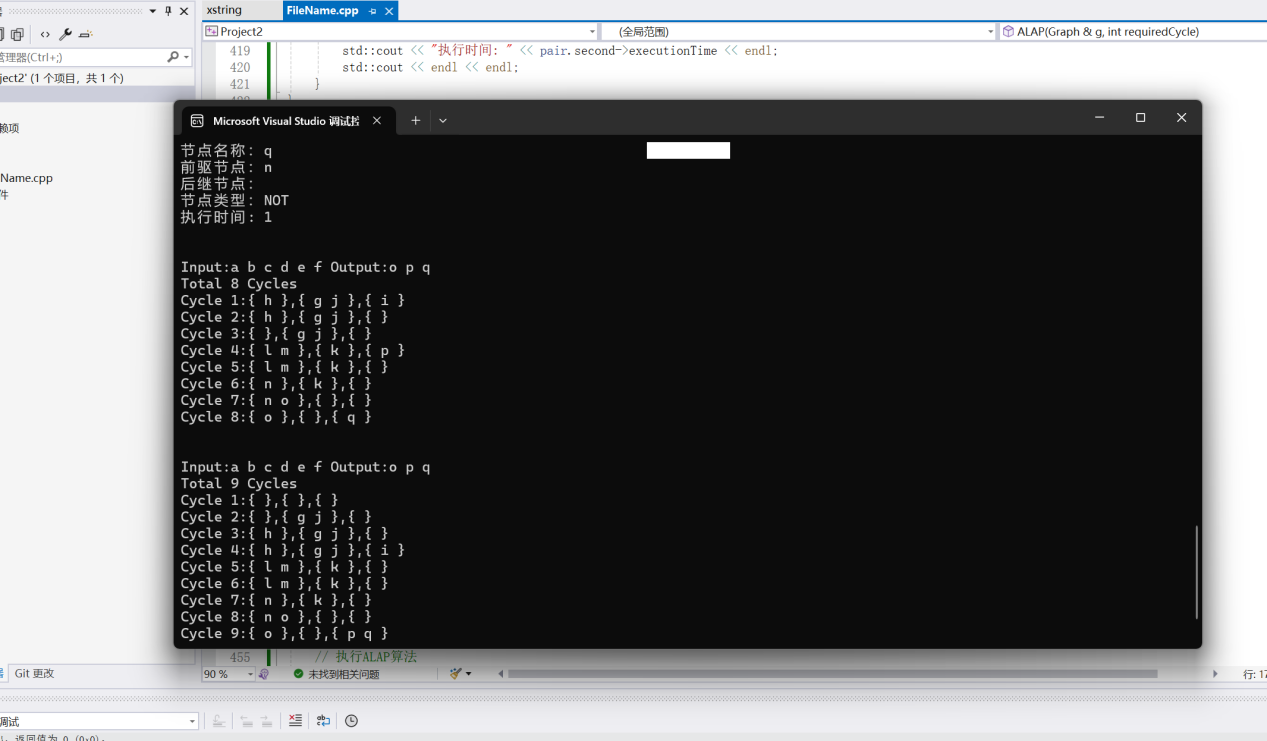
|  |
| --- |
| // ALAP算法  void ALAP(Graph& g, int requiredCycle) {  int minCycle = ASAP(g);  g.clear();  if (requiredCycle < minCycle) {  std::cout << "时间过短，无法完成" << endl;  return;  }  // 每次循环是一个周期,周期从后往前推  int currentCycle = requiredCycle;  while (g.processCount < g.size) {  // 维护当前周期的调度节点集  vector<Node\*> processingNodes;  // 查找当前周期可以调度的节点  for (auto& pair : g.nodes) {  Node\* currentNode = pair.second;  // 跳过已调度节点  if (currentNode->startCycle != -1) {  continue;  }  // 表示后继节点是否已全部调度过  bool nextProcessed = true;  int minNextStartCycle = requiredCycle + 1;  if (!currentNode->next.empty()) {  for (Node\* n : currentNode->next) {  if (n->startCycle == -1) {  nextProcessed = false;  break;  }  // 表示后继节点的最小开始时间  minNextStartCycle = min(minNextStartCycle, n->startCycle);  }  }  // 当前节点的执行时间不能超过从当前周期到后继节点最早开始周期的时间差  if (nextProcessed && currentNode->executionTime <= (minNextStartCycle - currentCycle)) {  processingNodes.push\_back(currentNode);  }  }  // 调度当前周期的节点  if (!processingNodes.empty()) {  for (Node\* node : processingNodes) {  node->startCycle = currentCycle;  g.processCount++;  }  }  currentCycle--;  }  } |

* 1. **结果输出：**
     1. 对于周期消耗为1的执行结果（ALAP的要求时间为5）



* + 1. **选做题：对于不同周期数消耗的执行结果**

**以AND：2，OR：3，NOT：1为例（ALAP的要求时间为9）：**



**九、总结及心得体会：**

通过本次实验，我深入理解了 ASAP 和 ALAP 调度算法的原理和实现方法，掌握了使用编程手段对 BLIF 文件进行解析和处理的技巧。在实验过程中，我遇到了一些问题，如 BLIF 文件解析时的边界条件处理、算法实现中的逻辑错误等，但通过查阅资料和调试代码，最终都得到了解决。

通过对不同 BLIF 文件的测试，验证了算法的正确性和通用性。同时，我也认识到在实际应用中，需要考虑更多的因素，如运算的周期数、资源约束等，以提高调度的效率和质量。

此外，本次实验也让我更加熟悉了 C++ 语言的使用，提高了我的编程能力和问题解决能力。在今后的学习和工作中，我将继续深入学习 EDA 相关知识，不断提升自己的专业素养。

1. **对本实验过程及方法、手段的改进建议：**
   1. 增强鲁棒性：

添加环路检测，避免无限循环。

基于真值表内容（如1- 1）精确判断门类型。

* 1. 扩展功能：

支持多周期门延时，提升算法实用性。

增加图形化展示，直观呈现调度结果。

**报告评分：**

**指导教师签字：**