数字集成电路设计自动化基础

高层次综合项目报告

小组成员：

姚晓辉 21307140024

徐一丁 21307140036

刘致依 20300750091

目录

一、项目说明

1. 项目介绍

本项目是《数字集成电路设计自动化基础》课程的高层次综合课程设计（一）。该工具以课程提供的 IR 作为输入，借助课程提供的语法分析器来读取 IR 文件，然后完成调度、寄存器及操作的绑定、控制逻辑综合、生成 RTL 代码。最后，利用仿真软件测试 RTL 代码（点乘操作模块）的功能，并得到正确的结果。

2. 组内分工

姚晓辉	控制逻辑综合、RTL 生成、报告撰写
徐一丁	数据结构构建、寄存器绑定、操作绑定
刘致依	数据结构构建、调度、测试结果分析

3. 工具使用方法

以下流程在系统版本为 Windows 11 22631.3737，编译器版本为 g++ (Rev3, Built by MSYS2 project) 13.2.0 的环境下可以正常运行。

3.1 编译

打开终端，在 sources 文件夹下执行 `g++ *.cpp -o hls`，即可得到 hls.exe。编译好的 hls.exe 已经在 exe 文件夹中提供。

3.2 运行

打开终端，在 hls.exe 所在目录下执行`.\hls filename`（将 filename 替换为 IR 的文件名），即可生成对应的 RTL 代码。

以课程提供的测试 IR 为例，将 test.ll 置于 hls.exe 所在目录下，在终端中打开该目录并执行`.\hls test.ll`，即可生成 RTL 文件 dotprod.v。生成好的 dotprod.v 已经在 RTL 文件夹中提供。

二、设计说明

1. 设计思路

将程序划分为如下几步（具体过程在主函数 main.cpp 中）：

①IR 读取与数据流图生成；

②调度；

③绑定（寄存器与操作）；

④控制逻辑综合与 RTL 代码生成。

为了简化设计，有如下的规定：

①读取 IR 时，生成的图中只有数据流。规定每个基本块结束后必定是返回和跳转（分支）两种控制流之一，因此可以用一个结构体来表示基本块结束后的控制流；

②同一时刻只有一个基本块工作，其他基本块均不工作。因此调度算法得到简化，可以对每个基本块单独进行调度，无需考虑基本块间的控制流；

③基本块内可以共享寄存器与运算单元，基本块间不共享寄存器与运算单元，以简化绑定算法，绑定时无需考虑基本块间的控制流；

④每个操作只需要 1 周期来实现。

2. 主要算法

2.1 IR 读取与数据流图生成

为了防止混淆，更改 parser.h 和 parser.cpp，将课程提供的语法分析器相关的类和结构体全部置于 `namespace parser` 下。

这一过程在构造函数 `Function::Function(const std::string& fileName)` 中实现，过程如下：

IR 读取：

(1) 使用语法分析器 `parser::parser p`，读取 `filename` 文件到 `p` 中；

(2) 如果 `p` 中基本块最后一条语句不是控制流（返回/分支），则添加一条跳转语句，跳转到下一个基本块（如果已经是最后一个基本块，则跳转到基本块 0）；

(3) 读取 `p` 中的函数名、返回类型；

(4) 读取 p 中的入口参数到_statements 中；

(5) 读取 p 中的语句和控制流，将语句读取到_statements 中（读取操作数时，操作数的_name 暂时用变量名/基本块 label 表示），控制流读取到临时变量中；

(6) 建立变量名与语句编号的映射，建立基本块 label 与基本块编号的映射；

(7) 将临时变量中的控制流读取到_blockEnd 中

(8) 将操作数的_name，从变量名/基本块 label，变为语句编号/基本块编号；

数据流图生成：

(9) 生成数据流图中的点（函数入口参数除外）；

(10) 生成数据流图中的边（函数入口参数相关的数据流除外）；

(11) 为数据流图的_blocknum 赋值（基本块数）

2.2 数据流图上的调度

这里介绍数据流图 class Graph 的调度（Function::schedule()只是调用了数据流图的调度算法）。

数据流图上的调度算法在 Graph::list_schedule()中实现，使用列表调度法，但因为该项目中没有明确的延时或运算单元数目约束，实际上相当于一个 ASAP 调度算法。过程如下：

(1) 计算入度。

(2) 对于每个基本块，进行调度：

首先将块内入度为 0 的点进队。

然后，只要队列非空，就不断循环：进行出队操作，访问出队的点 u，将 u 的所有块内的出边(u, w)连接的 w 点的入度减小 1。若 w 的入度减到 0，且 w 点未访问过，则 w 点进队，并且 w 的调度周期数等于 u 的周期数+1。

(3) 将调度信息输出，便于调试。

2.2.1 算子绑定到周期 - 资源约束下最小时间

原理：

使用列表调度算法，使用资源全局数组变量 resource 表示每个算子所有的资源。总体原理与 ASAP 方法类似，不同之处在于每个算子被调度（pop 出调度队列）

之前要检查对应的资源，如果算子的对应资源不足，则 push 回队列末侧等待下一轮检查。

功能：

打包成一个函数（list_shcedule()）放在 Graph 类里，使用时运行类的实例化对象的函数即可给 Graph 对象中所有结点的属性 cycle 赋值。赋值的是该算子在每个块内的相对调度周期，例如，每个块内第一个运行的算子对应的周期被赋为 1。

实现：

在 Graph 类里增加一个 init_stage()函数，根据定义的全局变量 cycle（表示每个运算运行的周期数）来初始化图中表示运行阶段的变量。

在 Graph 类里增加一个 cal_indegree()函数来计算所有结点的入度。这里的入度是指同一块内其他结点指向某一结点的入度。

在 Graph 类里增加 list_schedule()函数的声明，然后在类外定义。

首先调用之前介绍的两个函数进行初始化；

声明函数所需的队列、时钟周期计数等变量；

在调度队列放入入度为 0 的结点，然后进入主循环；

主循环即表示时钟计数的增加，每个循环代表一个时钟周期。在每个循环弹出队列中初始有的元素并进行判断，资源充足则开始运行，否则将结点 push 回队列。运行的结点即可将该循环对应的周期去赋值 CycleID，并设置开始标志为 1；

增加周期并遍历所有结点，更新结点的所在周期数，并判断是否结束（finish=1），对于运行结束的结点，就更新该结点的所有边的终结点的入度并将相应结点放入调度队列，同时释放资源并复原结点信息。

结果：

运行 Graph 类实例化对象中的 list_schedule()函数并打印 Graph 类中所有结点的 _Cycle 属性，得到结果与预期一致。

```
5 to 8
indegree is 0
blocknum is 1
time is 2
6 to 7
indegree is 0
blocknum is 1
time is 1
7 to 8
indegree is 0
blocknum is 1
time is 2
8 to 9
indegree is 0
blocknum is 1
time is 3
9 to
indegree is 0
blocknum is 1
time is 4
10 to 0
indegree is 0
blocknum is 1
time is 0
11 to
indegree is 0
blocknum is 2
time is 0
```

其他尝试：

尝试用控制流或判断块跳转变量表示块之间跳转以整体控制，并写出一个版本实现。但是考虑到最终循环的跳出需要通过预先运行函数代码计算一遍，这种“上帝视角”在实际的 HLS 中不常用，也就是循环不会在算子周期分配时就展开，故最终还是选择得到块内的相对运行周期，块间跳转留给后续步骤。

2.3 数据流图上的绑定

这里介绍数据流图 `class Graph` 的绑定（`Function::bind()`只是调用了数据流图的绑定算法并统计绑定信息）。

2.3.1 寄存器绑定

在 `Graph::regBind(const vector<int>& blockMaxTime, const`

vector<vector<Vertex*>>& vertexLists)中实现。

绑定过程（使用左边算法）：

遍历每个基本块，进行如下操作：

①首先，分析每个点对应数据的生存周期（存储、返回、分支除外），将分析结果存入 regTable[b]中。

②然后，使用数组 regNum 记录占用情况，数组下标为寄存器编号，数组元素为占用情况（数组元素为 0 代表未被占用）。

③遍历每个周期，按时序分配寄存器：来到周期 t，若有寄存器被占用，将占用情况减小 1，然后遍历每个算子，首先获取未被占用的寄存器编号，然后，如果该算子生存周期恰好从 t 开始，则分配一个寄存器，寄存器的占用情况即为生存周期数。

④分配后，将寄存器信息汇总到_registers 中。

最后，将寄存器绑定信息输出，以便调试。

2.3.2 运算单元绑定

在 Graph::opUnitBind(const vector<int>& blockMaxTime, const vector<vector<Vertex*>>& vertexLists)中实现。

绑定过程和寄存器绑定类似（使用左边算法）：

遍历每个运算类型和每个基本块，进行如下操作：

①首先，分析生存周期（返回和分支除外），生存周期即为各个算子被调度的周期，将分析结果存入 opUnitTable[b]中。

②然后，使用数组 opUnitNum 记录占用情况，数组下标为运算单元编号，数组元素为占用情况（数组元素为 0 代表未被占用）。

③遍历每个周期，按时序分配运算单元：来到周期 t，若有运算单元被占用，将占用情况减小 1，然后遍历每个算子，首先获取未被占用的运算单元编号，然后，如果该算子生存周期恰好为 t，则分配一个运算单元，占用情况为 1。

④分配后，将运算单元信息汇总到_units 中。

2.3.3 寄存器和运算单元绑定

在 Graph::bind()中实现。

首先统计每个块中算子的最大周期，统计每个块中的节点，然后调用上述两个绑定函数即可。

2.4 控制逻辑综合与 RTL 生成

这里主要介绍控制逻辑的实现。

2.4.1 基本块间的状态机

规定每个周期仅有一个基本块激活，所以可将模块的状态分为空闲、完成、块 n 激活、块 n-1 激活、块 n-2 激活、……、块 0 激活。用独热码为状态编码，以便直接得到各个状态的信号，以 4 个基本块的模块为例：

```
assign {idle, complete, bb3_en, bb2_en, bb1_en, bb0_en} = state;
```

`state` 为 6 bit 的变量，从低位到高位分别表示块 0 激活、块 1 激活、块 2 激活、块 3 激活、完成、空闲。同时，还需要记录上一个状态 `last_state`，以便 phi 操作使用。

状态转移如下：

若状态为空闲，则等到 `start` 信号为 1 时将输入信号寄存并激活基本块 0；

若状态不是空闲，则需要根据基本块结束后的控制流，生成次态。如果是有条件分支，则根据条件是否满足，选择对应的次态；如果是无条件分支（跳转），则激活跳转目标对应的基本块；如果是返回，则次态为完成。

2.4.2 基本块内的控制逻辑

基本块内有一个计时器，记录基本块运行的时钟周期。如果基本块激活，则计时器也激活。根据计时器的值，可以得到对应 MUX 和计算单元的控制信号。

若基本块激活，且计时器达到最大值，则输出一个完成信号以便上述状态机判断基本块是否计算完成。

RTL 代码生成的具体实现位于 RTL.cpp 文件中。为了便于后续仿真测试，令 RAM 的地址为 4 bit。

3. 数据结构定义

3.1 IR 代码中函数的表示

3.1.1 操作数 struct Operand

这里的“操作数”既可以是常数，也可以是基本块的编号（跳转或 phi 语句），还可以是语句的编号（语句对应的变量）。

数据成员如下：

(1) `string _name` 操作数的名称，以字符串形式存储

此处的“名称”是一个常数或基本块/语句的编号，不是 IR 中的变量名。

(2) `bool _isConst` 表示操作数是否为常数

3.1.2 语句 struct Statement

为了方便表示，函数的入口参数也作为一条“语句”，其所在基本块的序号为-1。

数据成员如下：

(1) `string _name` 语句对应的变量名

(2) `int _blockID` 基本块的序号（-1 表示输入变量）

(3) `int _type` 计算类型

(4) `vector<Operand> _operands` 操作数

(5) `bool _isArray` 是否为数组

(6) `int _reg` 绑定的寄存器编号

(7) `int _opUnit` 绑定的计算单元编号

3.1.3 基本块结束的信息（返回或分支） struct BlockEnd

数据成员如下：

(1) `bool _isReturn` 是否返回

若该变量为假，不返回，则该基本块结束的控制流为分支。

(2) `Operand _returnValue` 返回值

可以返回变量，也可以返回常数。

(3) `int _branchCond` 分支条件对应语句的编号

若小于 0，则为无条件分支（即跳转）。

(4) `int _branchTrue, _branchFalse` 条件为真、条件为假的分支目标

分支目标为基本块的序号。如果是无条件分支，则跳转到`_branchTrue` 对应的基本块。

3.1.4 函数 `class Function`

表示 IR 代码中的函数。除返回和跳转（分支）外，函数的一条语句（Statement）对应一个操作，同时也对应数据流图中的一个算子（一个节点）。

主要的 public 接口如下（get 和 set 函数在这部分省略）：

(1) `Function(const string& filename)`

构造函数，借助解释器读取 IR 文件，具体过程在后面介绍。

(2) `void schedule()`

调度，向屏幕中输出“正在调度”，并调用数据流图的调度算法。

(3) `void bind()`

绑定，向屏幕中输出“正在绑定”，并调用数据流图的绑定算法。绑定之后，为函数的每一条语句统计绑定的寄存器和运算单元信息，

private 成员如下：

(1) `string _name` 函数名

(2) `int _retType` 返回类型

(3) `vector<BlockEnd> _blockEnd` 基本块结束的信息（返回或分支）

下标为基本块的序号。

(4) `vector<Statement> _statements` 函数的所有语句（包括入口参数）

下标为语句的序号。

(5) `Graph _graph` 数据流图

3.2 数据流图的表示

3.2.1 节点 `struct Vertex`

主要的 public 接口如下：

(1) `Vertex(int type, int identifier, int time)`

构造函数，构造一个操作类型为 type，对应语句编号为 identifier，所属周期为 time

的节点。

(2) `void addInEdge(Edge* edge)`

增加一条入边。

(3) `void addOutEdge(Edge* edge)`

增加一条出边。

数据成员如下：

(1) `int _type` 操作类型

(2) `int _identifier` 点对应语句的编号

(3) `int _time` 点所处周期数

(4) `int _visited` 是否被访问过

(5) `int _indegree` 入度（只统计基本块内的点）

(6) `int _outdegree` 出度（只统计基本块内的点）

(7) `int _blockID` 属于子块的编号

(8) `vector<Edge*> _inEdge` 入边集合

(9) `vector<Edge*> _outEdge` 出边集合

3.2.2 边 `struct Edge`

主要的 public 接口如下：

(1) `void addFromVertex(Vertex* f)`

设置边的起始点。

(2) `void addToVertex(Vertex* t)`

设置边的终止点。

数据成员如下：

(1) `Vertex* _fromVertex` 起始点

(2) `Vertex* _toVertex` 终止点

3.2.3 图 `class Graph`

主要的 public 接口如下（get 和 set 函数在这部分省略）：

(1) `bool bind()`

寄存器和运算单元的绑定，具体算法在后面介绍。

(2) `void list_schedule()`

调度，具体算法在后面介绍。

(3) `void display()`

输出图的信息（用于调试），包括各个点的出边，入度，基本块的序号，调度的时间。

(4) `void cal_indegree()`

计算各个点的入度（只统计基本块内的点）。

private 函数如下：

(1) `bool regBind(const vector<int>&, const vector<vector <Vertex*>>&)`

寄存器绑定函数，第一个参数为各个基本块执行时间的最大值，第二个参数为各个基本块的节点。

(2) `bool opUnitBind(const vector<int>&, const vector<vector <Vertex*>>&)`

运算单元绑定函数，第一个参数为各个基本块执行时间的最大值，第二个参数为各个基本块的节点。

数据成员如下：

(1) `int _blocknum` 图内基本块的数量

(2) `vector<Vertex*> _vertexList` 图中所有点

(3) `vector<Edge*> _edgeList` 图中所有边

(4) `vector<vector<vector<int>>> _registers` 共享寄存器的连接信息

`_registers[`块编号`][`寄存器编号`][`周期数`]`的值为输入算子编号。

使用 public 接口访问时，`getRegisters()[`块编号`][`寄存器编号`][`周期数`]`的值为输入算子编号。

(5) `vector<vector<vector<int>>> _units[15]` 共享计算单元的连接信息

15 为操作的种类数。

`_units[`计算类型`][`块编号`][`计算单元编号`][`周期数`]`的值为算子编号。

使用 public 接口访问时，`getUnits(`计算类型`)[`块编号`][`计算单元编号`][`周期

数]的值为算子编号。

(6) vector<int> _blockMaxTime 基本块执行时间的最大值

时间从 0 开始。若总周期数为 1，则时间最大值为 0；若基本块内无计算单元，则时间最大值为-1。

3.3 控制逻辑综合与 RTL 生成

相关的类、函数和枚举定义在 namespace RTL 下。

3.3.1 RTL 模块 class Module

主要的 public 接口如下：

(1) Module(const Function& function)

构造函数，构造一个模块类，对应的函数为 function，并统计函数的入口参数（即模块的输入信号和连接的 RAM）。

(2) void generateRTL()

将 RTL 代码生成到_moduleRTL 字符串中，并写入文件。

private 函数如下：

(1) void moduleDeclaration()

将模块声明写入_moduleRTL 字符串中。

(2) void portsDeclaration()

将各个端口的声明写入_moduleRTL 字符串中。

(3) void varsDeclararion()

将各个变量的声明写入_moduleRTL 字符串中。

(4) void stateMachine()

将状态机代码写入_moduleRTL 字符串中。

(5) void moduleIO()

根据函数入口参数，处理模块的输入输出（数组：输出地址等信号，非数组：将输入信号寄存），将对应代码写入_moduleRTL 字符串中。

(6) void moduleReturn()

处理模块返回值，将对应代码写入_moduleRTL 字符串中。

(7) void blockCount(int blockID)

将基本块的计时器的代码写入_moduleRTL 字符串中。

(8) void blockReadWrite(int blockID)

将基本块的 IO 读写（即对数组 RAM 的读写）代码写入_moduleRTL 字符串中。

(9) void opUnit(int blockID, int type, int unitID)

将计算单元的 RTL 代码（例如加法器、乘法器）写入_moduleRTL 字符串中。

(10) void blockReg(int blockID, int regID)

将寄存器的 RTL 代码写入_moduleRTL 字符串中。

(11) void moduleEnd()

将"endmodule"写入_moduleRTL 字符串中。

数据成员如下：

(1) const Function& _function 模块对应的函数

(2) string _moduleRTL 模块的 RTL 代码

(3) vector<int> _input 函数入口参数对应语句的编号

3.3.2 辅助函数

(1) int blockCountWidth(int cycle)

根据基本块周期数，计算块内计时器位宽（即 log2(cycle)向上取整）。

(2) string port(int portType, int width, const string& name)

端口的字符串，例如"input [31:0] a"。

(3) string var(int varType, int width, const string& name)

变量的字符串，例如"wire [31:0] a"。

(4) string reg(const Function& f, int statementID)

语句对应寄存器的 RTL 变量名（例：基本块 0 寄存器 1 对应变量名"bb0_reg1"）。

(5) string operandReg(const Function& f, const Operand& a)

操作数对应寄存器的 RTL 变量名（例：操作数为基本块 1 的寄存器 2，对应变量名"bb1_reg2"；操作数为常数 10，对应"10"）。

(6) string opName(const Function& f, int statementID)

string opName(int blockID, int type, int unitID)

运算单元名（例：基本块 0 加法器 1 对应运算单元名"bb0_add1"）。

(7) `string operatorStr(int type)`

运算符号（例：乘法为`"*"`，加法为`"+"`）。

(8) `string state(int blockNum, int blockEnable, bool idle, bool complete)`

状态编码（例：`"5'b00001"`）。

三、测试结果

使用编译好的 hls.exe 为课程提供的测试 IR（向量点乘运算，文件名为 test.ll）生成 RTL 代码，保存到 dotprod.v 中。

然后，手工编写一个仿真文件（RTL 文件夹下的 sim_dotprod.v），其中

a 数组的前 16 个数据为{1, 2, 3, 4, 5, 6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}；

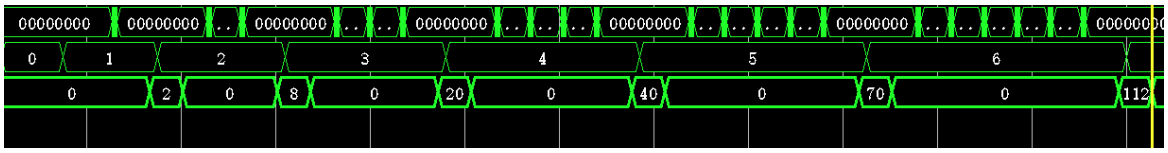
b 数组的前 16 个数据为{2, 3, 4, 5, 6, 7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}。

输入变量 n 从 0 逐一增加到 16，对应的返回值应为

0, 2, 8, 20, 40, 70, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122。

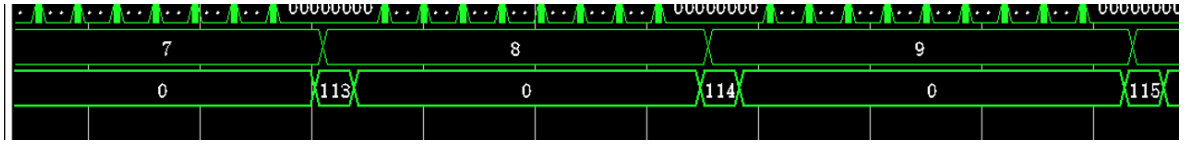
借助 Vivado 的仿真功能，可以得到模块的功能仿真结果。

n 从 0 到 6 的仿真结果：



结果为 0，2，8，20，40，70，112，符合预期。

n 从 7 到 9 的仿真结果：



结果为 113，114，115，符合预期。

n 从 10 到 16 的仿真结果在 RTL 文件夹中提供，结果仍然符合预期。

四、总结

我们设计了一个简单的高层次综合工具，可以将课程提供的 IR 经过调度、寄存器及操作数绑定、控制逻辑综合、RTL 代码生成，得到最终的 RTL 文件，并且使用仿真工具，验证了测试模块（向量点乘）功能的正确性。