Fudan_Digital_EDA

复旦数字集成电路设计自动化项目文档

原文件说明

1. function define:

```
define int foo(int a, int b); return value could be int/void
```

2. operators:

```
// support arrays:
int a[100]; define int foo(int a[], int b[], int N);
// load:
load a value from array. a[10] cannot be used directly. load(a, 10);
// store:
save a value to array. store(a, 10, c) -> a[10] = c;
=:
   assign value to a variable, follows the static single assignment rule
+:
   c = a + b;
-:
   c = a - b;
*:
   c = a * b;
/:
   c = a / b;
==:
   cond = a == b;
<:
   cond = a < b;
>:
   cond = a > b;
```

```
>=:
   cond = a >= b;
<=:
   cond = a <=b;
br:
   br label or br cond label1 label2
phi:
   phi function, select the right value from different blocks, according to SSA rule
   phi(value1, block_label1, value2, block_label2, ...); The default block label from the definition of the function is 0.
return:
   return a value
```

3. examples

```
define int dotprod(int a[], int b[], int n)
    c = 0;

start:
    i = phi(0, 0, i_inc, calc);
    cond = i >= n;
    br cond ret calc

calc:
    ai = load(a, i);
    bi = load(b, i);
    cl = phi(c, 0, cr, start);
    ci = ai * bi;
    cr = cl + ci;
    i_inc = i + 1;
    br start;

ret:
    cf = phi(0, c, start, cr);
    return cf;
```

project要求

以上述IR作为输入，我们提供基本的IR的parser
根据上述IR，完成调度、寄存器及操作的绑定、控制逻辑综合，函数输入的数组综合为RAM存储器，需要根据load/store指令来读写存储器数据。最终生成RTL代码。基本计算操作可以调用RTL计算模块，或直接使用原始操作符
不超过6位同学为1组，后面的两个Project仍按此分组。
最终提交代码、技术报告(包括测试结果)，并注明每位同学贡献。

项目成员

成员名称	学号
郑志宇	20307130176
邱峻蓬	20307130028
任钰浩	20307130243
沈笑涵	20307130063
周翔	20307130188

项目说明

本项目实现了从中间表示IR到生成RTL语言的过程。生成了数据流图、控制流图，实现了在块内的调度，完成了寄存器以及操作的绑定，实现了控制逻辑的综合。最终生成一个RTL的代码。我们将项目文件编译成了两个可执行文件，分别是适用于windows 的hls.exe 以及适用于linux 的hls。经过测试，可执行文件功能正常。

功能说明

该工具可以读入IR文件，然后执行生成数据流图和控制流图的过程，在有限运算资源的条件下执行调度算法，然后进行寄存器以及计算资源的分配。随后完成控制逻辑的综合。根据上面实现的结果输出一个verilog的module代码。可以使用testbench进行功能测试验证。

创新点

1. 项目采用了面向对象的编程方式，项目成员实现了明确的分工，使用了工厂模式以让每个成员的工作相对独立，保证了程序运行与维护的便利性。
2. 为了程序的可拓展性和灵活性，我们留下了很多接口和类去方便进行下一步的改进。
3. 在数据流图与控制流图部分，我们舍弃了传统的指针所表示的有向图，而用数组加上哈希进行标识和索引，有更快的访问速度。
4. 在调度部分，我们除了使用`ASAP`和`ALAP`方法，还使用了列表调度法来实现对周期的调度。
5. 在寄存器绑定部分，我们考量计算机在存储变量时候的全局变量与局部变量的思想，将寄存器分为块之间传输变量的全局寄存器以及块内使用的局部寄存器，形成了一一种更简单的调度逻辑。
6. 在运算资源绑定部分，我们使用匈牙利算法还有最大匹配的方法求解了最佳的绑定结果。
7. 控制逻辑综合部分我们综合了周期的逻辑来表示寄存器的行为，以此来作为实现代码的基础。
8. 生成`verilog`代码部分我们对生成的块进行了划分，用上了前面调度得到的所有信息。譬如使用控制流图生成跳转逻辑，使用调度后的数据流图生成周期控制逻辑等等。
9. 我们还使用了`vivado`的`HLS`工具进行功能的验证，有很不错的效果。但是因为生成`verilog`的逻辑与运算的周期存在一定的差异，所以结果并非一致。

项目使用方法

环境要求

- `visual studio 2019`及以上版本能正常打开项目中的所有文件
- 注意`src`编码格式为`Unicode(UTF-8)`
- 生成一个`hls.exe` 的可执行文件能够在`windows` 下运行。如果想要生成`Linux`下的`Makefile`，可以修改一下`Makefile`文件。

使用方式

根据自己的环境编写替换`Makefile`，目前项目中的`Makefile`仅仅适用于`window`。

`Windows`：在`testfile` 文件夹中写好测试文件`file.ll` 后，在项目文件夹下运行以下命令：

```
.\hls.exe testfile\dotprod.ll
```

`Linux`：在`testfile`文件夹中写好测试文件`file.ll`后，在项目文件夹下运行以下命令：

```
./hls testfile\dotprod.ll
```

cmd中运行结果示意：

```
Result of CFG && DFG
...
...
Finsh Register Allocation and Binding
...
...
Finsh Scheduling
...
...
Finsh Calculate Allocation and Binding
...
...
Finsh Synthesize Control Logic


Finsh Generate Finate Sate Machine.


Output file: testfile/dotprod.v
```

`IR`文件格式声明

变量：

仅支持`int`变量，支持数组，定义与`c`语言类似。如`int a, int a[]`
函数定义：

```
define int foo(int a, int b);
```

返回值可以是`int`和`void`。

要注意`IR`文件不应当出现一个变量既是表达式的输入变量，又同时作为表达式的输出变量的形式，我们认为这是不符合`IR`文件的格式规范的行为。同时，我们并没有对`parser`文件进行修改，所以`parser`的一些不合理的遗留问题并没有进行针对性解决，其对格式的要求以及对变量的检查方面的工作并不完善。所以很多时候，能够解析的语句并不一定能够生成`verilog`代码。

操作定义：

`load`：

> 从数组中加载一个数据，如`b=load(a, 10)`就是加载`a[10]`到`b`

`store`:

> 将数据存储到数组：如`store(a, 10, c)`，将`c`存储到`a[10]`

`=`:

> 赋值

`+`:

> `c = a+b`

`-`:

> `c = a-b;`

`*`:

> `c = a*b;`

`/`:

> `c = a/b;`

`==`:

> `cond = a==b;`

`<`:

> `cond = a < b;`

`>`:

> `cond = a> b;`

`=`:

> `cond = a>=b;`

`<=`:

> `cond = a<=b;`

`br`:

> `br label`：无条件跳转
>
> `br cond label1 label2`：有条件跳转

`phi`:

> 从不同模块中选择不同的变量值：`phi(value1, block_label1, value2, block_label2, ...);`
>
> 函数入口的`label`默认为`0`。

`return`:

> 返回或返回值。

语言实例

```
define int dotprod(int a[], int b[], int n)
c = 0;
start:
    # i = 0:i_inc 0:calc
    i = phi(0, 0, i_inc, calc);
    # cl = c:cr 0:calc
```

```
    cl = phi(c, 0, cr, calc);
    cond = i >= n;
    br cond ret calc;
calc:
    ai = load(a, i);
    bi = load(b, i);
    ci = ai * bi;
    cr = cl + ci;
    i_inc = i + 1;
    br start;
ret:
    return cl;
```

项目的结构

```
├── picture # README文档的说明图片
├── projectfile # 项目要求文档与参考资料
|   ├── read.md # 项目要求文件
##### 测试文件目录
├── testfile
|   ├── SRAM.v
|   ├── dotprod.v
|   ├── gcd.v
|   ├── tb_gcd.v
|   ├── tb_dotprod.v
|   ├── dotprod.ll
|   ├── gcd.ll
##### 源文件
├── src
|   ├── main.cpp
|   ├── parser.h
|   ├── parser.cpp
|   ├── HLS.h
|   ├── HLS.cpp
|   ├── dataflowgraph.h
|   ├── dataflowgraph.cpp
|   ├── controlflowgraph.h
|   ├── controlflowgraph.cpp
|   ├── schedule.h
|   ├── schedule.cpp
|   ├── leftAlgorithm.h
|   ├── computeresource.h
```

```
|   ├── Hungarian_algorithm.h
|   ├── control_logic.h
|   ├── cycleTable.h
|   ├── FSMachine.h
|   ├── FSMachine.cpp
#####
├── README.md # 项目文档
├── Makefile # windows平台下的makefile
├── hls.exe # windows平台下编译的结果，使用mingw32_make
├── hls # Linux下编译结果，使用cmake
```

项目技术细节

项目顶层文件

此部分由郑志宇同学维护，接口与功能与小组成员共同商议确定。最终实现了项目的并行推进与项目成员对函数的独立维护。

`HLS`类是项目的顶层模块，它的每一个函数都实现了一个特定的功能：

`├── HLS.h`

`├── HLS.cpp`

```cpp
    // 实现图的生成
    void generate_CFG();
    //设置测试时间
    void setTestTime();
    // 遍历所有节点的算法
    void travelaround();
    void travelback();
    // 实现调度算法
    void perform_scheduling();
    // 实现寄存器分配和绑定
    void perform_register_allocation_and_binding();
    // 实现运算资源的分配与绑定
    void perform_calculate_allocation_and_binding();
    // 控制逻辑综合方法
    void synthesize_control_logic();
        // 生成状态机
    void genFSM();
```

```
18      // 输出verilog文件
19      void outputfile();
```

其中，函数`travelaround()`和`travelback()`是郑志宇同学给出的按照拓扑排序数据流图的方法，用作参考，并不参与项目功能的实现。其中，只介绍顺序遍历`travelaround()`，`travelback()`完全类似。`travelaround()`的具体的实现使用队列的数据结构，首先初始化节点的入度，取出所有入度为`0`的节点放入队列，入度为`0`代表节点并不依赖于同一个数据流图的其他节点，所以可以直接进入队列进行调度。

而在主循环的过程是节点出队列，对节点进行操作，然后将后续节点的入度减小，并检查后续节点此时的入度。为`0`时进入队列。

- 因为`phi`操作的特殊性，`phi`操作的入度理论上应该是`1`，但是实际上`phi`操作在数据流图中可能依赖多个变量，这样的可能导致实际上会有`phi`操作的节点的入度在小于`0`的时候才入栈。不能保证其入度恰好为`0`。虽然可以通过令`phi`操作的入度等于其依赖变量数目来完善这一部分程序，但逻辑上还是并没有进行调整。

这一步的遍历是遍历项目的数据流图的基本操作。

Part 1生成数据流图以及控制流图

此部分由郑志宇同学完成。

```
├── HLS.cpp
```

```
|    ├──  generate_CFG();
```

```
├── controlflowgraph.h
```

```
├── controlflowgraph.cpp
```

```
├── dataflowgraph.h
```

```
├── dataflowgraph.cpp
```

函数及文件说明

```
void HLS::generate_CFG() {
    // 通过IR生成数据流图以及控制流图
    CFG = ControlFlowGraph(parsered);
}
```

这个函数在`HLS`中利用解析出的文件生成了数据流图和控制流图。控制流图中是包含数据流图以及跳转关系的节点，而数据流图由操作节点以及其依赖关系构成。在生成数据流图的时候，控制流图也顺便完成了生成。

数据流图实现的接口：

```
// CreateGraph
//
// 用于标记节点是否被访问
std::vector<int> Mark;
// 用于标记节点当前的入度，为0表示可以被直接访问
std::vector<int> InVertex;
// 用于标记节点当前的出度
std::vector<int> OutVertex;
void Initialize();
DataFlowGraph(basic_block& bb);

// operation
// 创建边
void CreateEdge(int from, int to);

// 修改流图的最短周期
int getPeriod() const;
void setPeriod(int T);
// get
std::vector<node>& get_opList();// 节点列表
std::vector<BranchEdge>& get_Branches();// 输出边列表
std::vector<InputEdge>& get_inputList();// 块的输入变量
std::vector<OutputEdge>& get_outputList();// 块的输出变量列表
std::string& get_label();// 模块名
std::vector<int>& ToVertex(int from);// 节点的出度节点
std::unordered_map<std::string, int>& myOutvartable();// 输出变量的哈希表
```

主要介绍后面的方法的意义：

`void Initialize()`：初始化入度和出度以及节点的访问标记，以便于开始新一轮的遍历操作。

`std::vector<node>& get_opList()`：操作使用`vector`存储，这样的好处是方便快速索引而且访问速度快

`std::vector<BranchEdge>& get_Branches()`：块内的跳转逻辑，这部分能够得到清晰的块之间的跳转关系，`BranchEdge`标识了块的跳转方向和跳转条件。

`std::vector<InputEdge>& get_inputList()`：块的输入变量组成的`vector`，主要信息有变量来自哪个块以及变量名。

`std::vector<OutputEdge>& get_outputList()`：块的输出变量组成的`vector`，主要信息是外部的块需要哪些变量。

`std::string& get_label()`：获得块的标签。

`std::vector<int>& ToVertex(int from)`：获得节点`from`的出度节点的序号组成的`vector`，`vector`中的每一个`index`都是节点`from`出度的节点。可以用于拓扑排序中寻找出度的节点。

`std::unordered_map<std::string, int>& myOutvartable()`：这是一个哈希表，其最大的意义在于可以从一个变量的名称直接索引到输出这个变量的操作节点的`index`。可以实现节点的快速查找。可以用于反向拓扑排序中寻找入度的节点。

控制流图实现的接口：

```
    // 控制流图的生成
    ControlFlowGraph(parser& p);
    // 获得CFG图中的DFG节点
    std::vector<graph_node>& getDFGNodes();
    // 获得DFG在节点向量中的下标
    int getIndex(std::string label);
    //获取下一个块所指向的块的下标
    std::vector<int> NextNode(std::string label);
    std::vector<int> NextNode(int index);
    // 返回信息的接口
    std::string getfuncname();
    std::vector<var>& getvar();
    int getRet_type() const;
```

主要介绍一些方法的意义：

`std::vector<graph_node>& getDFGNodes()`：这个方法返回了一个`DFG`节点构成的`vector`，可以实现节点的快速索引。

`int getIndex(std::string label)`：可以实现从标签索引到`DFG`的节点的`index`的操作。

技术细节

控制流图与数据流图的意义在于为后面的调度的实现创建了条件，提供很多方便的接口让后面的实现能够访问。

本部分使用了一些访问速度比较快的结构来实现一部分功能，有效提升了运行的效率：

1. 哈希表索引节点
2. 数组实现有向图

本部分在创建数据流图的时候，在首尾各创建了一个虚拟的节点，首节点意义是作为外部变量输入的索引位置，而末节点的意义是作为需要输出到外部的变量的索引位置。利用哈希表在所有的数据流图中匹配需要的信息并进行更新和存储最终生成了这样的一个完整的数据流图。然后是在创建控制流图的时候为了输出三个输入变量生成了一个名为`fiction_head`的虚拟节点。这个节点的主要意义也在与索引到头部的节点，在生成的`verilog`的状态机中，这个节点是标志着函数未开始运行的节点。在接收到`ap_start`的信号之后就会进入到运行状态。

Part 2完成周期的调度

此部分由邱峻蓬同学完成。

`├── HLS.h`

```
// 实现调度算法
void perform_scheduling();
```

`├── schedule.h`

`├── schedule.cpp`

本部分根据解析获得的数据流图和控制流图的结构，对于各运算操作进行周期的调度，考虑在运算资源约束下的最小延时的调度策略。需要注意的是块与块之间是独立考虑的，即不同数据流图之前的周期是分开调度的。

其基本思想是根据采用列表调度法，根据数据流图的结构，首先进行`ASAP`和`ALAP`调度，根据这二者的调度结果作为列表调度法优先选取同一层级的操作的顺序的标准。然后再次进行`ASAP`调度，在每次周期迭代中考虑满足硬件约束的可调度操作并按照之前的规则优先选择，调度在该周期中。

本算法考虑了多周期运算操作的调度问题。

函数及文件说明

根据数据流图中运算操作的宏定义：

```
constexpr auto T_ASSIGN = 1; // 赋值操作;
constexpr auto T_ADD = 2;    // 加法操作
constexpr auto T_SUB = 2;    // 减法操作
constexpr auto T_MUL = 5;    // 乘法操作
constexpr auto T_DIV = 40;   // 除法操作
constexpr auto T_LOAD = 3;   // 载入操作
constexpr auto T_STORE = 3;  // 存储操作
constexpr auto T_BR = 1;     // 分支操作
constexpr auto T_LT = 2;     // 小于操作
constexpr auto T_GT = 2;     // 大于操作
constexpr auto T_LE = 2;     // 小于等于操作
constexpr auto T_GE = 2;     // 大于等于操作
constexpr auto T_EQ = 2;     // 等于操作
constexpr auto T_PHI = 2;    // Phi 操作
constexpr auto T_RET = 1;    // 返回操作
```

数据流图中节点的调度结果：

调度结果存储在数据流图的`node`类的成员`T_start`和`T_end`中，最后在数据流图中的`Period`中得到数据流图的完整运行周期数。

硬件类`Hardware`：

我们定义了默认运算资源的个数。

```
private:
```

```
    int adder;
    int mul;
    int div;
    int sram;
    int adder_available;
    int mul_available;
    int div_available;
    int sram_available;

public:
    Hardware()
    {
        adder = 2;
        mul = 1;
        div = 1;
        sram = 1;
        adder_available = adder;
        mul_available = mul;
        div_available = div;
        sram_available = sram;
    }
```

周期类`Period_Rec`：

存储`ASAP`调度和`ALAP`调度结果，第一个`int`为运算操作开始的周期，第二个`int`为运算操作结束的周期，`vector`下标为对应运算操作的索引。

```
private:
    std::vector<std::pair<int, int>> ASAP_RES;
    std::vector<std::pair<int, int>> ALAP_RES;
```

方法说明：

```
bool meet_resources_constraint(
    std::map<int, struct Hardware> &rec,
    int i,
    DataFlowGraph &DFG
);
void reset(
    Hardware &hardware,
    int i,
```

```
    DataFlowGraph &DFG
);
int max(int a, int b);
int min(int a, int b);
void ASAP(DataFlowGraph &DFG, Period_Rec &REC);
void ALAP(DataFlowGraph &DFG, Period_Rec &REC);
bool cmp(
    const std::pair<int, int> &a,
    const std::pair<int, int> &b
);
void improved_table_schedule_forDFG(DataFlowGraph &DFG);
void improved_schedule_forCFG(ControlFlowGraph &CFG);
```

`bool meet_resources_constraint(std::map<int, struct Hardware> &rec, int i, DataFlowGraph &DFG)`：判断当前运算操作是否满足硬件约束，即是否可被调度。若可以被调度，则消耗一个对应的硬件资源。

`void reset(Hardware &hardware, int i, DataFlowGraph &DFG)`：根据运算操作释放相应运算单元。

`void ASAP(DataFlowGraph &DFG, Period_Rec &REC)`：实现基于`ASAP`方法的多周期运算操作的调度操作，主要基于图的拓扑排序操作，结果存于`Period_REC`类`REC`中。

`void ALAP(DataFlowGraph &DFG, Period_Rec &REC)`：实现基于`ALAP`方法的多周期运算操作的调度操作，需要进行一次`ASAP`调度获得最长调度周期后进行，主要基于根据出度的图的拓扑排序操作，结果存于`Period_REC`类`REC`中。

`bool cmp(const std::pair<int, int> &a, const std::pair<int, int> &b)`：`ALAP`和`ASAP`的调度结果差存于`vector`中，该函数对于`vector`内数据进行排序。

`void improved_table_schedule_forDFG(DataFlowGraph &DFG)`：根据`DFG`图的结构实现列表调度法。

`void improved_schedule_forCFG(ControlFlowGraph &CFG)`：遍历`CFG`图中的各个`DFG`，分别进行周期调度。

技术细节

1. 多周期的`ASAP`和`ALAP`调度方法在具体实现中需对原始拓扑排序流程进行改进，对拓扑排序前一层级的运算操作的所剩周期进行记录，以此判断是否该运算操作已执行完毕可依此减少后级节点的入度。
2. 由于数据流图和数据流图的头尾虚节点的特殊数据结构，为避免调度结果出错，需在遍历迭代前首先于第0周期对头/尾虚节点进行独立处理，设置虚节点的周期并减少后级/前级节点的入度/出度。
3. 列表调度法不同于`ASAP`调度方法，由于优先选取规则的存在，运算操作的实际开始调度周期与其拓扑排序不存在必要的联系。因此在实现算法过程中不采用`std::queue`数据结构而改用`std::map`（因为运算操作的索引唯一），`std::map`中存储待被调度的运算操作。每一次周期迭代中，首先根据排序结果选取容器中元素，判断是否满足硬件约束，进行调度。再考虑当前周期是否有操作已执行完毕，恢复硬件资源并减小图中对应后序节点入度，将入度为0节点置入容器中，结束当前周期迭代。

Part 3完成寄存器的绑定

此部分由任钰浩同学完成。
本部分根据周期调度的结果计算变量的生存周期，从而将变量与寄存器绑定。需要注意的是块与块之间是独立考虑的。

其基本思想是根据调度结果，从输出开始往前遍历，生成每个操作数的生存周期。然后根据生存周期重叠的变量无法共享寄存器，不重叠的变量可以共享寄存器的原则进行寄存器分配。这实际上是一个区间染色问题，可以通过左边算法来实现快速求解最优解。

函数及文件说明

`├── HLS.h`

```
// 每个块的寄存器分配结果
std::vector<std::vector<std::pair<std::string, int>>> REG;
```

`├── HLS.cpp`

```
// 执行寄存器分配和绑定
perform_register_allocation_and_binding();
```

`├── leftAlgorithm.h`

生存周期结构体的定义

```
struct varPeriod {
    std::string var;//变量名
        int startp;//起始周期
        int stopp;//结束周期
};
```

这个结构体是为了提炼出`CFG`中对寄存器分配有用的信息，使之后的左边算法编写只用聚焦与这个结构体，可以在`CFG`完成前就开始编写，提高项目的并行度。

根据调度结果得到生存周期

```
std::vector<varPeriod> graph2VarPeriods(DataFlowGraph& DFG);
```

左边算法分配寄存器

```
std::vector<std::pair<std::string, int>>
leftAlgorithm(std::vector<varPeriod> V);
```

左边算法将区间图中的区间按其左边（区间起点）排序，然后从排序的队列中，取出一个区间，逐个从剩下的区间里根据左边顺序，逐个找到与前面区间不重叠的区间，给相同“着色”（分配寄存器），重复上述操作，知道所有区间（变量）都被正确“着色”（分配寄存器）
实际操作中是定义一个`endpos`，将`startp`大于`endpos`的变量分配给该寄存器，然后刷新`endpos`为该变量的`stopp`，知道没有变量可以分配，给出新的寄存器然后将`endpos`重置为0。

技术细节

1. 有的数据不用分配寄存器，所以并不需要在`graph2VarPeriods`中转化为`varPeriod`，这类数据分两类，一是常数，而是该函数的输入，需要在`graph2VarPeriods`函数中识别并将其排除

常数：

对于常数，默认将不会进行寄存器的分配。

函数输入：

这些数据的特点是它们都来自于`fiction_head`

2. 上述提到寄存器分配时块与块之间是独立考虑的，但为了使得块中的数据在未使用前可以保留，不被下一个块的寄存器分配冲掉，我们参照计算机函数调用的思想，设计了一个`mem`寄存器用来存储每个块执行后输出的变量。显然这样的方法比每个变量用一个寄存器所使用的寄存器还多，这并不是一个优秀的方法，该项目中是为了模拟实现左边算法的寄存器分配才出此下策。之后改进时可以替换为更好的算法。

Part 4完成计算资源的绑定

此部分由沈笑涵同学完成。
本部分使用了`HLS.h`中各块内的寄存器与变量的绑定结果`REG`、和各块`DFG`中存储`node`计算结点的`opList`信息。

其基本思想是利用匈牙利算法，按照各块的拓扑排序遍历结果，对同一时刻入度为0的所有结点进行硬件资源的匹配。其中，生成的代价矩阵值为匹配该计算结点所额外增加的硬件代价。通过对矩阵的多次迭代，得到代价最低的硬件资源匹配结果，并按照硬件资源分配结果绑定输入输出寄存器。

函数及文件说明

`├── HLS.h`

```
//计算资源（包括加法器、乘法器和除法器）
std::vector<computeresource> COR;
//计算资源匹配结果（匹配的是node结点的编号和计算资源COR的序号）
std::vector<std::vector<std::pair<int, int>>>  CSP;
```

`├── HLS.cpp`

```
void HLS::perform_calculate_allocation_and_binding();
```

`├── computeresource.h`

`├── Hungarian_alogrithm.h`

计算资源类的定义

该部分完成了绑定的计算资源的基本信息的说明，包括计算资源的类别、输入端绑定寄存器、输出端绑定寄存器、以及相关绑定操作的方法定义

```
class computeresource{
    int flag;// 声明计算资源是加法器or乘法器or除法器
    std::vector<int> Ainputregisters;// 左边输入寄存器
    std::vector<int> Binputregisters;// 右边输入寄存器
    std::vector<int> outputregisters;// 输出寄存器
    computeresource(int flag1, int outputreg);// 构造函数
    void setinputAregisters(int reg);// 绑定计算资源左输入寄存器
    void setinputBregisters(int reg);// 绑定计算资源右输入寄存器
    void setoutputregister(int reg); // 绑定计算资源输出寄存器
    bool findareg(int reg);// 查找计算资源左输入端绑定寄存器
    bool findbreg(int reg);// 查找计算资源右输入端绑定寄存器
    bool findoutreg(int reg);// 查找计算资源输出端绑定寄存器
}
```

主要介绍一些定义的想法：

`std::vector<int> Ainputregisters`：计算资源输入端、输出端绑定的寄存器一般不止一个，因此通过vector向量存储；

查找每一个块中计算结点`node`与寄存器绑定结果

该部分完成了对每一个`DFG`中的`node`结点和寄存器绑定结果的提取，以及该块内的寄存器绑定结果的查找，可以实现通过输入变量名查找绑定的寄存器编号

```
std::vector<graph_node>& DFGS = CFG.getDFGNodes();
int findregister(std::vector<std::pair<std::string, int>> REGi,
std::string val);
```

实现将块内`node`结点与计算资源的绑定

该计算资源的实例化结果和绑定结果分别存储在`HLS.h`中新定义的两个变量中

```
std::vector<computeresource> COR;
std::vector<std::vector<std::pair<int, int>>>  CSP;
```

函数接口：

```
//实现计算资源与计算结点的绑定
std::vector<std::pair<int, int>> bindcomputeresource(
    DataFlowGraph& DFG,
    std::vector<std::pair<std::string, int>>REGi,
    std::vector<computeresource>& CORE
);
//实现将输出寄存器与计算资源输出端绑定
void bindoutputregister(
    DataFlowGraph& DFG,
    std::vector<std::pair<std::string, int>>REGi,
    std::vector<computeresource>& CORE,
    std::vector<std::pair<int, int>>CSPi
);
```

技术细节：

1.初始时生成一个队列，队列中压入目前状态下的所有入度为0的所有结点（即没有数据依赖的所有结点），并按照所需计算资源的种类（加法器、乘法器、除法器）分类，并将结果存储在三个`vector`迭代器中

2.分别对三个`vector`迭代器进行操作，统计每个计算结点在不同编号的对应计算资源绑定的代价，从而生成匈牙利算法中的代价矩阵。这里，我考虑的代价为该计算资源为了绑定某一计算结点所额外增加的输入端数据选择器的输入个数：如果某一计算资源的两个输入变量分配的寄存器均未与该计算资源相连，那么其代价为2；若输入变量所在寄存器中有一个与该计算资源相连，那么代价为1；如果该计算结点两个输入寄存器均与该计算资源绑定，那么代价为0。该操作在以下函数中生成：

```
std::vector<std::vector<int>>creatematrix(
    DataFlowGraph& DFG,
    std::vector<int>list,
    std::vector<std::pair<std::string, int>>REGi,
    std::vector<computeresource>& CORE,
    int flag,
    Hardware& hardware
);
```

3.利用匈牙利算法的步骤对矩阵进行操作，得到最大匹配数，并按照该匹配结果对这些计算结点进行计算资源的匹配。该操作在以下函数中实现：

```
//生成最大匹配
int maxcompair(std::vector<std::vector<cost_matrix_node>>&
matrix2);
```

4.当某一时刻的所有计算结点均完成计算资源的匹配后，将这些计算结点标记为`VISITED`，并对计算资源绑定输入寄存器，同时将其后序计算结点的入度减一。再次重复步骤一，压入当前入度为0的所有计算结点，并按照以上流程操作，直至当前块内所有计算结点均完成计算资源的绑定。以上所有操作在以下函数中实现：

```
std::vector<std::pair<int, int>> Hungarian(
    DataFlowGraph& DFG,
    std::vector<int>&list,
    std::vector<std::pair<std::string, int>>REGi,
    std::vector<computeresource>& CORE,
    std::vector<std::vector<int>>matrix,
    int flag,
    Hardware& hardware,
    int& k
);
```

5.匈牙利算法实现计算资源的绑定后，通过对绑定结果遍历，完成对各个计算资源的输出寄存器绑定。该操作在以下函数中实现：

```
void bindoutputregister(
    DataFlowGraph& DFG,
    std::vector<std::pair<std::string, int>>REGi,
    std::vector<computeresource>& CORE,
    std::vector<std::pair<int, int>>CSPi
);
```

Part 5完成控制逻辑综合

这部分由周翔同学完成。

```
├── cycleTable.h
```

```
├── control_logic.h
```

函数及文件说明

```
// 控制逻辑综合方法
    void synthesize_control_logic();
```

这个函数利用周期调度、寄存器绑定以及计算资源绑定的结果，汇总每个块、每个周期下运行的所有操作语句`std::vector<Statement>`。其中`Statement`是一个结构体，记录下了对应语句的信息（在后面说明）。

寄存器**Register**类

```
public:
    //寄存器的下标，与任钰浩的pair的索引值相对应
    int reg_index;
    // data中会存储当前周期下寄存器存储的值
    bool getData(int cycle, varPeriod& data);
```

其中，`varPeriod`是任钰浩同学定义的一个结构，内含变量名`var`、变量起始活跃时间`startp`与终止活跃时间`stopp`。

`bool getData(int cycle, varPeriod& data)`：输入周期数cycle，通过遍历寄存器中存储的所有变量对应的活跃周期，选择出该周期下该寄存器存储的变量名（包含在data中）。

选择器**Mux**类

```
public:
    int mux_index;                  //选择器下标
    bool chooseReg(int cycle, DataFlowGraph dfg,
std::vector<Register> REGs,
    std::vector<std::pair<std::string, int>> REGi,
    std::vector<std::pair<int, int>> CSP,
    std::vector<computeresource> com,
    Register& reg, std::string& _var);
```

`bool chooseReg（）`：默认`reg`下标为-1、`_var`为`NULL`作为未找到的结果。挑选出当前周期`cycle`、当前模块`dfg`下，选择器输入端所选取的寄存器`reg`，以及寄存器中存储的变量`_var`。这里要注意的是在不同块中活跃的寄存器可能不同，如果直接访问将会导致溢出错误，因此需要先结合寄存器绑定结果`REGi`，挑选出当前块中会使用到的所有寄存器编号，记为`std::vector<int> curDFGinput;`。之后，利用Register类的`getData()`函数判断当前周期寄存器`i`是否存有数据（记为`v`）。如果存有数据，由于一个寄存器可能会与多个计算

资源相连，无法保证`v`一定是在选择器连接的计算资源中被使用的，因此还需要结合计算资源绑定结果，找到相关周期的节点语句node进行判断，得到最后的结果。

经过小组讨论，`Register`类与`Mux`类仅用于表示连接的结构，可以反映门级连接，但实际上完成`PJ`的要求是生成`RTL`代码，并不需要使用到上述两个类。

控制器类

```
public:
    void generateCycles(std::vector<std::pair<std::string, int>> _REG, CFG);
    std::vector<Cycle> getCycle();
```

其中，`Cycle`是一个结构，定义如下：

```
struct Cycle {
    std::vector<Statement> Statements;
};
struct Statement {
    std::vector<std::string> vars;  //输入变量名
    std::vector<int> regs;          //变量对应的寄存器编号
    int optype;                     //操作类型
    int compute_resource_index;     //绑定的计算资源标号
    int outreg;                     //传出的寄存器编号
    std::vector<std::string> label; //为phi操作而设，记录数据来自哪一个块
};
```

`void generateCycles(std::vector<std::pair<std::string, int>> _REG)`：根据寄存器绑定、计算资源绑定结果，生成一个向量`std::vector<Cycle> C;`，`C[i]`对应周期`i`执行的所有`Statement`。首先要确定`C`的大小。经过与小组成员的协商，由于寄存器绑定的结果是针对每个块而言的，每当传入一个新的块时，所有的寄存器都会被覆盖掉，因此需要事先将本块中需要用到的从其他块中传入的变量导入到对应的寄存器中。
因此，这里将`C`的大小设置为块内语句总周期数加`1`，其中第零周期将郑志宇同学生成的`CFG`中的`MemMap`中存储的相关变量数据导入到绑定的寄存器中，最后一个周期均为`BR`指令或`RETURN`指令，在`BR`指令所处周期执行将本块会传出的数据导入到`MemMap`中的操作。
另外，`node`节点包含沈笑涵同学的计算资源绑定结果，以及不使用计算资源的节点。后者需要特别处理，否则会报溢出错误。

`std::vector<Cycle> getCycle()`：用于返回最终生成的周期表`std::vector<Cycle> C`。

其他函数

```
void Pair2Register(
    DataFlowGraph &DFG,
    std::vector<std::pair<std::string, int>> REG,
    std::vector<Register>& Regs
);
```

这一函数用于将任钰浩同学的寄存器绑定结果`vector<pair<string, int>>`结构转化成`vector<Register>`结构。

技术细节

1. 在`chooseRegs()`与`generateCycles()`函数的编写时，比较容易出现堆栈溢出的问题，这就需要判断每个块内是否真正使用了相应的硬件资源。
2. 为了方便生成Verilog代码的工作，创建了一个Cycle结构，里面存放着对应周期执行的所有操作信息，并且保存了STORE、RET等操作的输出情况。
3. 为了在分块实现寄存器绑定的基础上，保证数据的正确性，这里在开始与结束阶段各加了一拍，实现了寄存器数据的导入与传出。

Part 6生成`verilog`代码

这一部分由郑志宇同学与任钰浩同学共同完成，郑志宇同学负责了数据流图的生成，所以负责状态机的跳转部分以及`module`块的生成。任钰浩同学负责寄存器的绑定部分，所以负责寄存器的行为的综合。

```
├── HLS.cpp

|    ├── genFSM();

├── FSMachine.h

├── FSMachine.cpp
```

生成`module`以及状态机的跳转逻辑

这一部分由郑志宇同学完成

函数及文件说明

```
void HLS::genFSM() {
    outputFSM = FSMachine(CFG);
}
```

接口说明

```
std::unordered_map<std::string, std::string>& getStateMap() {
    return stateMapping;
}
// 生成module
void IOdefinationAppend(int ret_type,std::vector<var>& vars);
// 生成FSM
void FSMgener(ControlFlowGraph &CFG);
```

技术细节

这一部分的技术细节并未有很多体现，主要还是使用了工厂模式实现用类来代理实现生成输出函数。

基本流程是：

- 实现输入输出的实例化，创建`module`
- 实现状态机的编码，创建状态机寄存器
- 实现状态机的`if`语句以及`always`块的生成

输入的实例化：

我们参考了`vivado`的`HLS`工具生成的`verilog`文件的输入来实例化数组的行为。一个数组对应的是内存中的一块区域，为了在内存中读取或者写入数据，我们需要一个读信号以及一个写信号。此外，我们还需要读入`SRAM`中数据的一个输入信号以及向`SRAM`中写入数据的一个写入信号。综合出来如下所示的信号：

```
(
    input [31:0] var_q0, // 读入数据
    output [31:0] var_ad0, // 写入数据
    output [31:0] var_addresss0, // 偏移地址
    output var_ce0, // 读入使能
    output var_we0 //写入使能
);
```

而一个`int`型的变量则会综合出一个输入端：

```
(
    input [31:0] var
);
```

这样我们就完成了函数输入变量在对应的`module`中的实例化。

函数的输出类型则决定了是否会多出一个输出端口：

```
(
    output [31:0] ap_return
);
```

`module`的运行需要一系列的其他参数，在这里逐一说明：

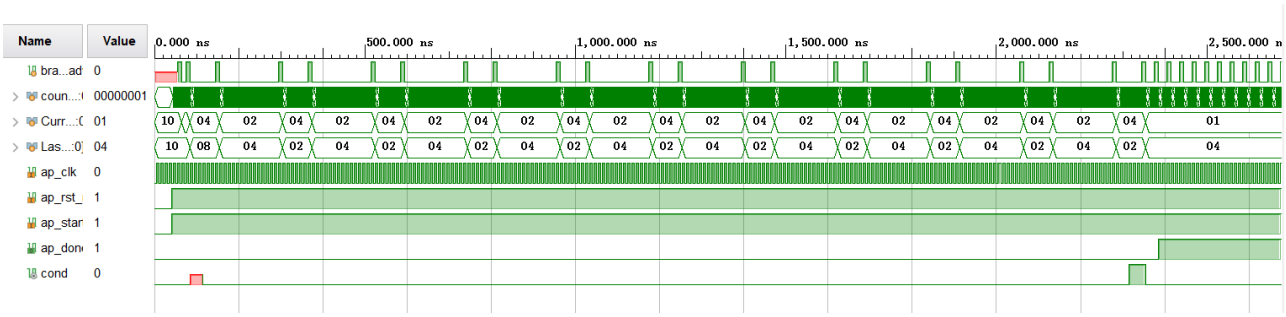
```
(
    input   ap_clk, // 时钟信号
    input   ap_rst_n, // rst_n信号
    input   ap_start, // 程序的开始信号
    output  reg ap_done // 程序结束信号
);
```

代码块的状态编码

对 N个代码块的状态编码主要思路就是生成一个 N维的 0组成的字符串，然后将其中一位修改为1即可完成对每个状态的唯一编码。编码完成之后，使用`std::unordered_map<std::string, std::string>& getStateMap()`函数将块的标签与块的状态对应上。

状态机跳转的实现

状态机跳转示意图：



生成一个`always`块的语句，对于每个块都生成一个`if`逻辑语句块，进行跳转条件的设置，状态机对时钟`ap_clk`的`posedge`敏感。在接收到跳转信号`branch_ready`以及跳转条件`cond`之后进行跳转。不跳转`branch_ready`为 0，无条件跳转只需要`cond`为 1即可，为真为假跳转则需要`cond`的指示来实现。

实现寄存器与连线的行为综合

这部分由任钰浩完成

接口说明

```
void CounterGener(
    std::vector<std::vector<Cycle>> &Cycles,
    ControlFlowGraph &CFG
);
void perPeriodGener(
    std::vector<std::vector<Cycle>>& Cycles,
    ControlFlowGraph& CFG
);
void regDefGener(
    std::vector<std::vector<std::pair<std::string, int>>>& REG
);
```

块内运行周期计数器

根据我们构建项目的思想，程序在一个块内运行的周期数必然是固定的，这个逻辑被综合成了一个`always`逻辑块。它对`ap_clk`的上升沿敏感，每次上升沿到来时，判断当前的周期数是否到达了调度的周期，如果是，那么`counter`被置零，且产生一个状态机可以跳转的`branch_ready`信号。否则`counter <= counter + 1`，继续计量周期。

寄存器综合

可以将**op**分为以下几类

1. 计算类：`OP_ASSIGN`（赋值操作）`OP_ADD`（加法操作）`OP_SUB`（减法操作）`OP_MUL`（乘法操作）`OP_DIV`（除法操作）`OP_LT`（小于操作）`OP_GT`（大于操作）`OP_LE`（小于等于操作）`OP_GE`（大于等于操作）`OP_EQ`（等于操作）
2. 访存类：`OP_STORE`（存储操作）`OP_LOAD`（载入操作）
3. 跳转类：`OP_BR`（跳转操作）
4. phi类：`OP_PHI`（phi操作）
5. 返回类：`OP_RET`（return操作）

每类`op`的寄存器综合形式如下

1. 计算类：
   输入进行`op`对应运算符操作后存入输出寄存器，需要注意的是输入不一定是寄存器有可能是常数或函数输入这些不需要分配寄存器的数据，需要进行判断。（为了表示运算的周期，我们在运算的开始周期执行寄存器赋值，之后的运行的周期闲置）。示例如下：

```
    32'd7: begin
        reg_1 <= reg_4 * reg_1
    end
    32'd8: begin
    end
    32'd9: begin
    end
```

2. 访存类：
   在起始周期将`load`或`store`的使能信号置1，地址寄存器存入相应的地址（对于`store`，在这个周期还需要将数据存入写寄存器中），在结束周期将`load`或`store`信号置0，对于`load`，在这个周期将要写如的数据存入对应寄存器。示例如下：

- `store`

- ```
  'd6: begin
          b_we0 <= 1;
          b_address0 <= reg_1;
          b_ad0 <= reg_2;
      end
      32'd7: begin
      end
      32'd8: begin
          b_we0 <= 0;
      end
  ```

- `load`
- ```
  'd4: begin
          b_ce0 <= 1;
          b_address0 <= reg_1
      end
          32'd5: begin
      end
          32'd6: begin
          b_ce0 <= 0;
          reg_1 <= b_q0
      end
  ```

3. 跳转类：
   将该块中所有要输出的数据存入对应的`mem`寄存器中。示例如下：

```
    32'd14: begin
        Mem_i_inc <= reg_3
        Mem_cr <= reg_1
    end
```

4. `phi`类：
   根据上一个块来确定数据选取。示例如下：

```
    32'd1: begin
        if(LastState == state_0)
            reg_1 <= 0
        elseif(LastState == state_calc)
            reg_1 <= reg_1
        if(LastState == state_0)
            reg_2 <= reg_2
        elseif(LastState == state_calc)
            reg_2 <= reg_3
    end
```

5. 返回类：
   无操作。

连线综合

返回结果连线

示例如下：

```
    assign ap_return = reg_1;
```

`cond`连线

实际上是一个根据`CurrentState`的多路选择器。示例如下：

```
    assign cond = ((CurrentState == state_start) & reg_3) ||
        ((CurrentState == state_cal) & reg_3);
```

项目测试

测试文件 1 `dotprod.v`

IR文件由课程提供

```
define int dotprod(int a[], int b[], int n)
c = 0;
start:
```

```
    # i = 0:i_inc 0:calc
    i = phi(0, 0, i_inc, calc);
    # cl = c:cr 0:calc
    cl = phi(c, 0, cr, calc);
    cond = i >= n;
    br cond ret calc;
calc:
    ai = load(a, i);
    bi = load(b, i);
    ci = ai * bi;
    cr = cl + ci;
    i_inc = i + 1;
    br start;
ret:
    return cl;
```

测试文件由郑志宇同学提供：

```
├── testfile
│   ├── dotprd.v
│   ├── tb_dotprd.v
```

测试结果：

实例化`SRAM`

为了从数组中读数据，我们初始化了两个`SRAM`，`SRAM_a`，`SRAM_b`，这两个`SRAM`的`module`为生成的`module`的访问数组的功能提供了便利。在这里为了方便起见，我们取`n=10`，然后在`SRAM_a`，`SRAM_b`中存入了数据：

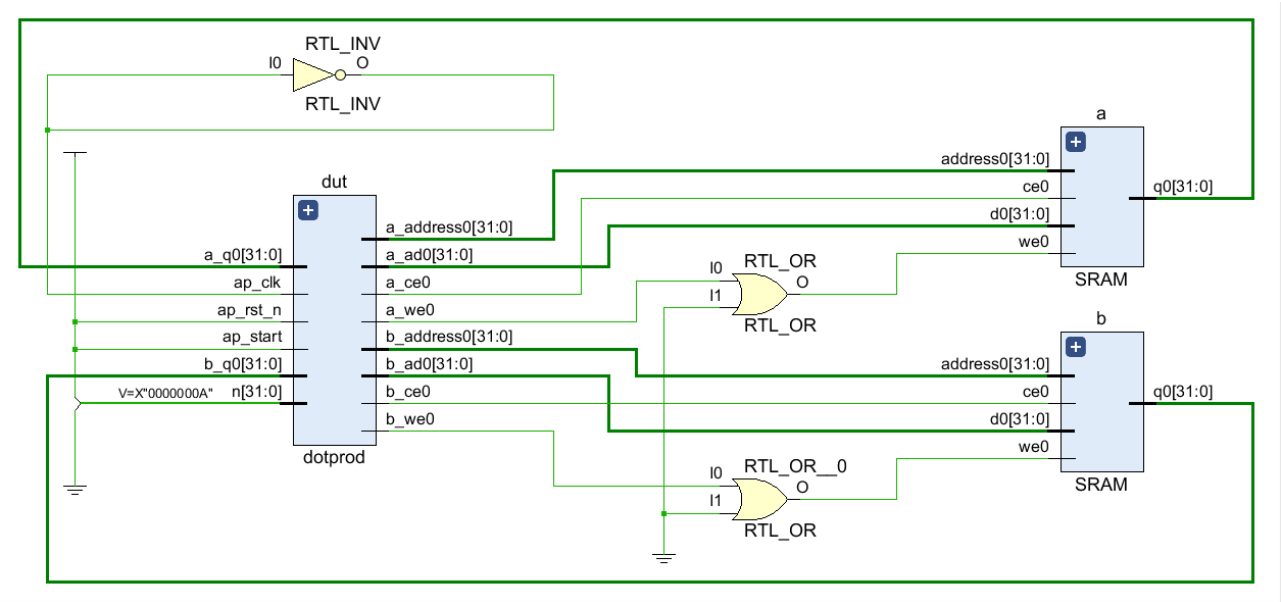
```
a={1,2,3,4,5,6,7,8,9,10};
```

```
b={10,9,8,7,6,5,4,3,2,1};
```

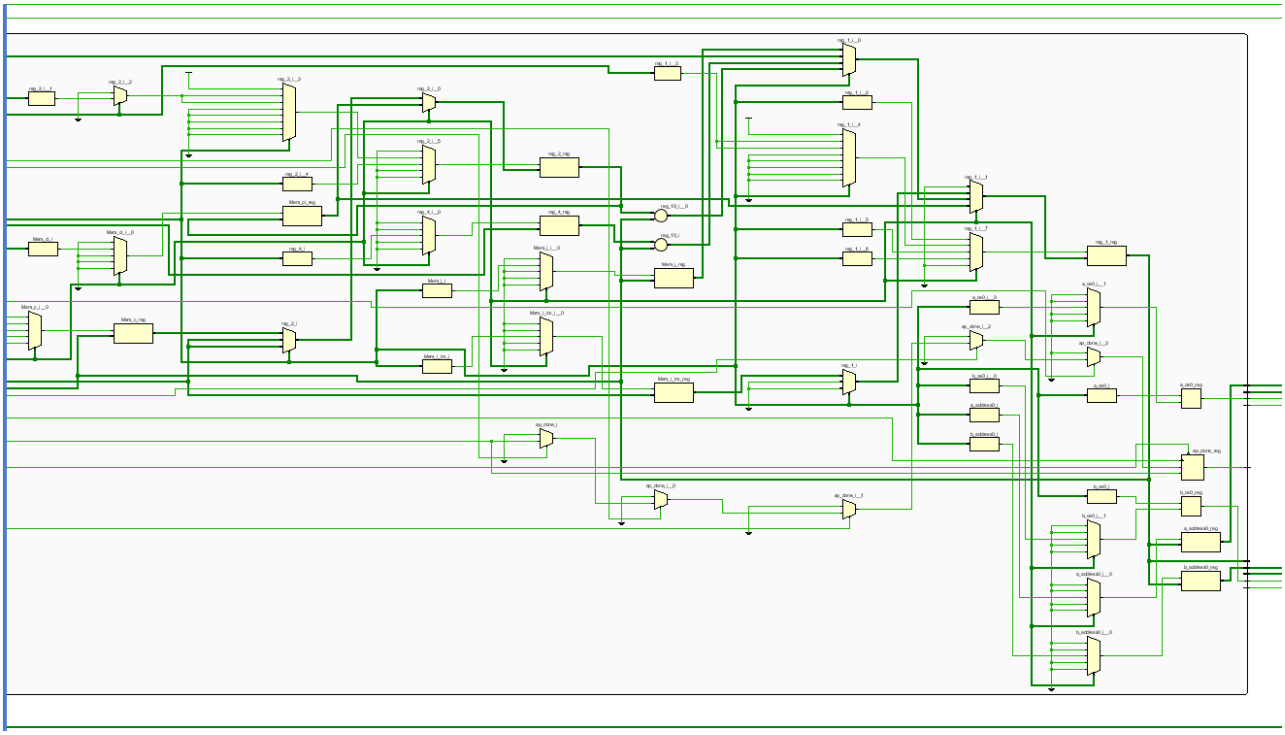
```
n=10;
```

我们对程序的预期结果是`220`；

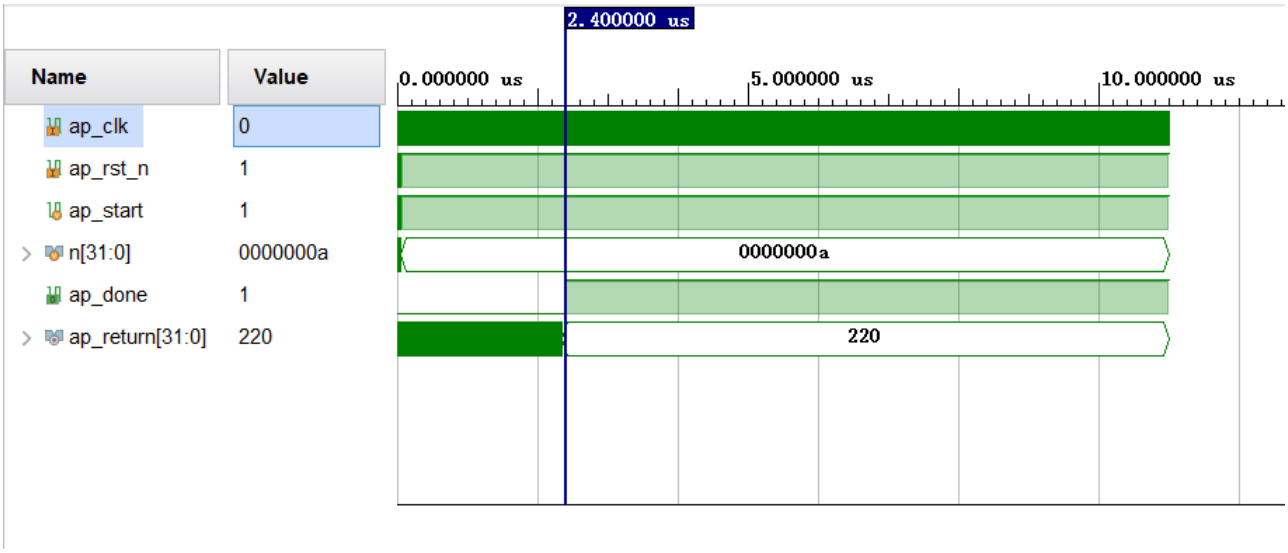
RTL电路图



部分内部结构示意：



测试的结果波形：



测试的结果如上所示，我们在`testbench`中模拟了一个`SRAM`去为`a_q0`，`b_q0`进行赋值，然后我们接收来自我们的状态机的输入的使能信号，地址信号以及写入的数据，最后完成了此程序的仿真。可以看到，在`ap_done`信号出现的时候，`ap_return`的结果已经正确而且稳定地出现。这与我们的预期结果相符。因为进行了一些寄存器分配，所以`ap_return`的值并不是存储在一个固定的寄存器中，只有在最终才会由分配好的寄存器进行传出。

测试文件 2 `gcd.v`

IR文件由周翔同学提供

```
├── testfile

|   ├── gcd.ll
```

```
define int gcd(int a, int b)
    c = a;
    d = b;

start:
    a1 = phi(c, 0, divisor, cal);
    b1 = phi(d, 0, remainder, cal);
    a_LE_b = a1 >= b1;
    br a_LE_b cal exchange;

cal:
    divisor = phi(b1, start, a1, exchange);
    larger = phi(a1, start, b1, exchange);
    remainder = larger - divisor;
```

```
15      cond = remainder == 0;
16      br cond ret start;
17
18  exchange:
19      br cal;
20
21  ret:
22      return divisor;
```

测试文件由周翔同学提供：

```
├── testfile

|   ├── gcd.v

|   ├── tb_gcd.v
```

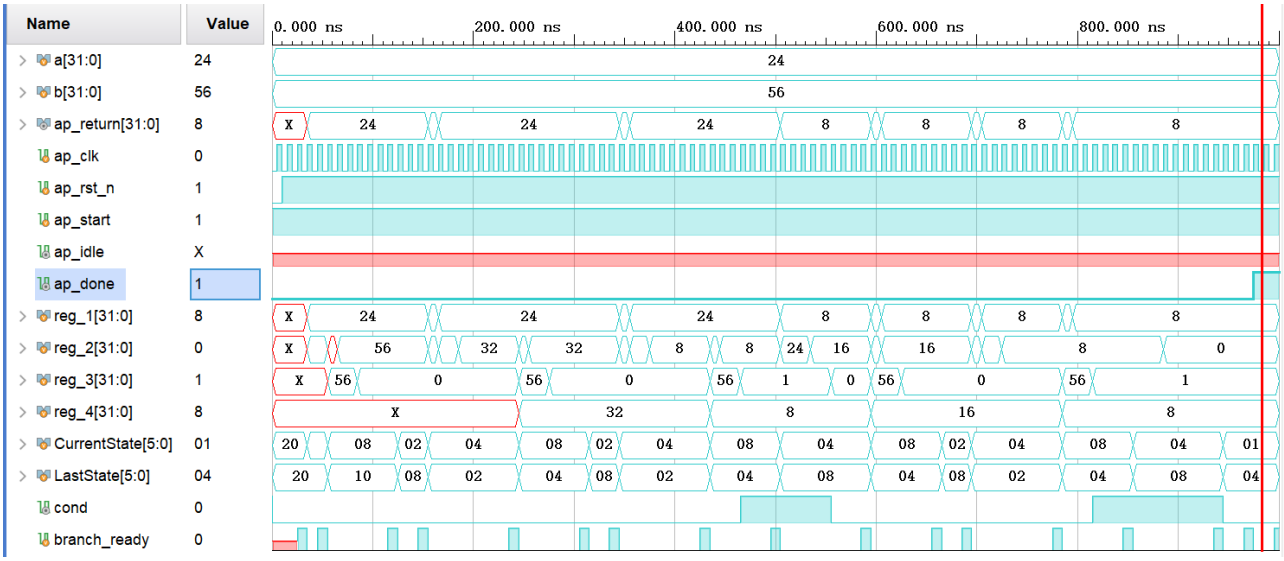
文件说明：

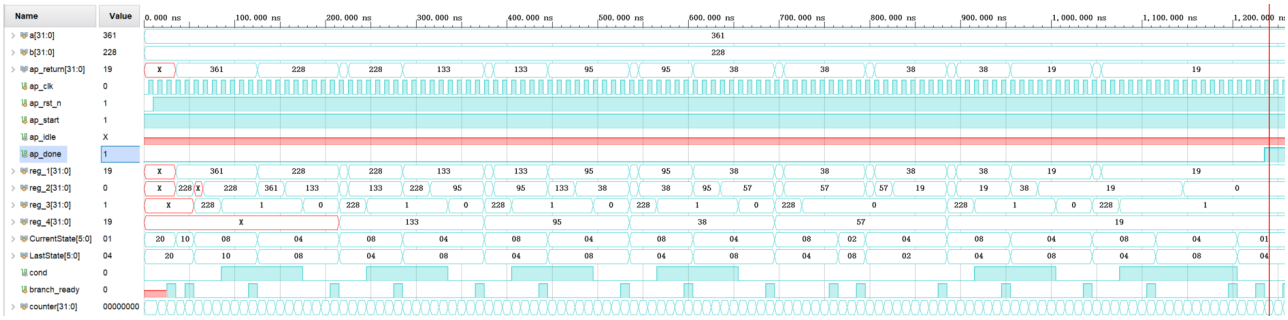
使用更相减损法计算两个数`a`、`b`的最大公因数。`exchange`块中只执行br操作，是为了在`cal`块中准确得到`a1`与`b1`中的较大值。

测试结果：

输入**A**	输入**B**	预期输出	实际输出
24	56	8	8
361	228	19	19

测试的结果波形：





测试的结果如上所示。可以看到，在`ap_done`信号出现的时候，`ap_return`的结果已经正确而且稳定地出现，与我们的预期结果相符。因为进行了一些寄存器分配，所以`ap_return`的值并不是存储在一个固定的寄存器中，只有在最终才会由分配好的寄存器进行传出，因此我们关心的只是在`ap_done`信号跳变为`1`的时候`ap_return`的输出结果。

测试文件 3 `Sum.v`

IR文件由周翔同学提供

```
├── testfile
```

```
│   ├── Sum.ll
```

```
define int Sum(int a[], int b[], int n)
    c = 0;
start:
    i = phi(0, 0, i_inc, calc);
    sum = phi(c, 0, temp, calc);
    cond = i >= n;
    br cond ret calc;
```

```
8
9  calc:
10     ai = load(a, i);
11     temp = sum + ai;
12     store(b, i, temp);
13     i_inc = i + 1;
14     br start;
15
16 ret:
17     num = n - 1;
18     res = load(b, num);
19     return res;
```

测试文件由周翔同学提供：

```
├── testfile
```

```
|   ├── Sum.v
```

```
|   ├── tb_Sum.v
```

文件说明：

使用`store`指令将`a`数组前`i`个元素的和存到`b`数组的第`i`个位置，最终输出`a`数组元素之和，即`b[n]`存储的结果。

测试结果：

这里设置输入：

```
n = 10;
```

```
a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

预期输出为：

```
b = { 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 }.
```

测试的结果波形：

测试的结果如上所示。可以看到，`b_ad1`的结果符合预期。在`ap_done`信号跳变为`1`时，`ap_return`的结果为`55`，即数组`a`元素之和，符合预期。

项目总结

我们组员之间通过共同合作完成了这一个项目，虽然一开始这个项目很难入手，但是在我们搭好的架构与多次共同的商讨中我们最终得到了一个相对完善的解决方案。这个解决方案很好地解决了生成`verilog`的问题。同时我们将一些中间的执行过程的输出体现在`cmd`中可供参考。此外，我们测试了自己书写的一些程序，均有非常好的效果。我们同时也发现了`parser`文件的不完善而导致的解析中会产生一些空节点的问题。这个问题需要另外的解决，我们考虑尽可能不动`parser`文件，这样的结果是对`IR`文件的输入的要求更加严格了。如果再做进一步的精进的话，可以从降低对`IR`文件的要求入手，合并部分逻辑，封装部分冗余代码以及优化寄存器分配的方案等。