课程设计

基于 LLVM IR 的高层次综合后端实现

院　　系：　微电子学院

专　　业：　微电子科学与工程（本研贯通）

　　微电子科学与工程

姓　　名：　庄集 唐嘉恒

日　　期：　**2024 年 5 月 25 日**

目录

基于 LLVM IR 的高层次综合设计

1 设计概述

高层次综合（High-level Synthesis）简称 HLS，指的是将高层次语言描述的逻辑结构，自动转换成低抽象级语言描述的电路模型的过程。所谓的高层次语言，包括 C、C++、SystemC 等，通常有着较高的抽象度，并且往往不具有时钟或时序的概念。相比之下，诸如 Verilog、VHDL、SystemVerilog 等低层次语言，通常用来描述时钟周期精确（cycle-accurate）的寄存器传输级电路模型，这也是当前 ASIC 或 FPGA 设计最为普遍使用的电路建模和描述方法。

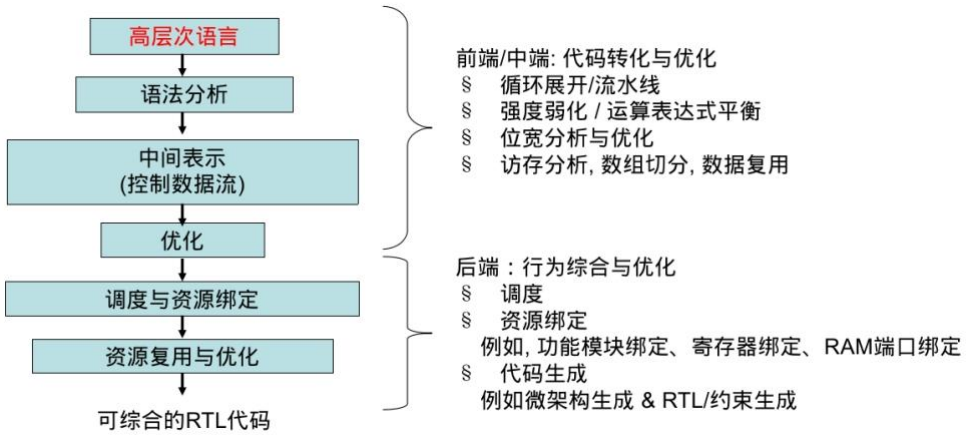


图 1：高层次综合流程

本组设计的程序主要实现了高层次综合的后端全流程，满足了本次课程设计的有关要求。该流程包括算子调度、寄存器和操作数的绑定、控制逻辑综合、函数输入的数组综合为 SRAM，最终生成可综合的 Verilog HDL 代码。此外，对于题目所给的 IR，本组设计的程序所生产的 RTL 代码通过了功能仿真，验证了程序的正确性。

由于要求中未限制资源使用，因此程序采用了 ASAP 调度算法，对于 Load 和 Store 算子，认为运算所需周期为 2，其余算子为 1；寄存器和操作数的绑定使用左边算法，并对全局变量和局部变量进行分别的考虑；控制逻辑综合采用一段式状态机加微控制器的形式实现；SRAM 使用 Vivado 生成单端口 RAM IP，在顶层模块实现了 IP 和程序生成模块的交互；通过编写 testbench 实现功能仿真，观测仿真的运行状态和结果验证程序的正确性。

该报告共分为设计概述、设计流程、功能仿真、总结四部分，后续部分将更详细地介绍程序的流程、算法细节以及实验过程。

2 设计流程

2.1 设计总体流程

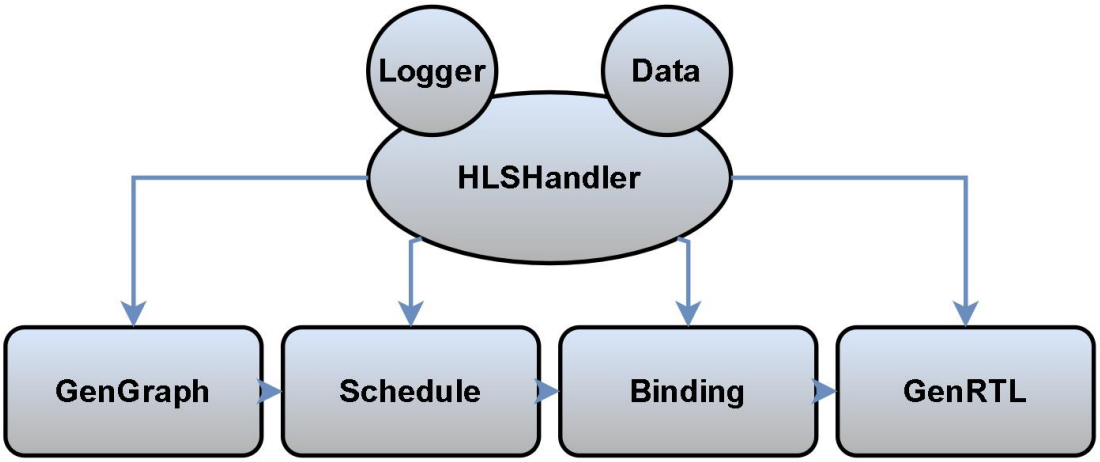


图 2：设计总体流程图

HLSHandler 统筹整个设计的工作进程，Logger 目录下导入 easylogger 库主要用于记录程序各部分运行时间和运行中间结果统计。程序运行时，首先根据 parser 得到的数据运行 GenGraph 目录下的 GenGraphGroup 生成后续运行所需的数据结构，然后调用 Schedule 下的 ASAP 调度算法进行算子调度，进而调用 Binding 类进行寄存器和操作数的绑定，最后调用 GenRTL 类生成 Verilog RTL 代码到指定目录。

2.2 数据结构

```
1. class Statement
2. {
3. public:
4.     int type;
5.     int num_oprands;
6.     std::string var;
7.     std::vector<std::string> oprands;
8.     int sch;                              //算子调度周期
9.     int indegree;                         //算子入度
10. };
```

```
1. class Graph

2. {
3. public:
4.     Graph(int port, Parser& p);
5.     ~Graph();
6.     int port, num_node;                   //算子端口号，算子节点数
```

```
    std::string name;                    //Basic Block 名称
     int sch_total;                      //算子总调度周期
     std::vector<int> outports;
     std::vector<std::vector<int>> matrix;      //算子的邻接矩阵
     std::vector<Statement> statements;         //算子的语句
     std::map<std::string, std::pair<int, int>> vars_life;
                                                //变量生命周期
     std::map<std::string, std::string> var_reg;//变量及其寄存器
     void show_info();
protected:
     void set_edge(int from, int to);
     void set_matrix();
     void set_indegree();
     void set_outport(Parser& p);
     void set_port2index(Parser& p);
     std::string get_port_index(std::string port);
private:
     std::map<std::string, std::string> port2index;
};
```

```
class GraphGroup
{
public:
     GraphGroup();
     ~GraphGroup();
private:
     std::string function_name;
     int ret_type;
     std::vector<var> function_params;
     std::vector<std::string> global_vars;
     int regs_num;
     std::vector<Graph> graphs;
     std::vector<std::vector<int>> graph_matrix;
};
```

GraphGroup 将所有基本块封装到其成员变量 vector<Graph> graphs 中，并由 Paerser 直接得到函数名、函数参数、返回类型、基本块之间的连接方式。

Graph 是对基本块的封装，由 Parser 得到基本块的端口号、块名称、算子连接矩阵、输出端口号、将所有算子封装到 vector<Statement> statements 中。

Statement 是对算子的封装，由 Parser 得到算子的类型、左右操作数、如果是 phi 指令，进一步获取算子的入度。

对于数据结构中其他的成员变量，如算子的调度周期 Statement::sch、基本块中变量的生命周期 Graph::vars_life,寄存器的数量 GraphGroup::regs_num 等，将由后续的调度后绑定操作得到。

2.3 ASAP 调度

调度的任务是要完成将操作算子分配到时钟周期从而在状态机内部建立起一个微控制器，使用 ASAP 调度算法前，我们需要确定各算子的运算周期。

表 1：算子运算周期

算子	ASSIGN	ADD	SUB	MUL	DIV	LOAD	STORE	BR
周期	1	1	1	1	1	2	2	1
资源	LT	GT	LE	GE	EQ	PHI	RET	
数量	1	1	1	1	1	1	1	

ASAP 调度算法的核心代码如下，其原理是对算有算子进行分层级的拓扑排序，按照拓扑排序顺序入队，同时确定其所在周期为当前出队算子时钟周期与其运算周期之和。调度算法的结果存储在数据结构中每个 Graph 里对应 Statement 的数据成员 sch 中，表示该算子在当前基本块内开始的周期。

```
1. void Schedule::ASAP(Graph& bb)
2. {
3.     int bbTime = 0;
4.     std::vector<bool> visited(bb.num_node, false);
5.     std::vector<float> indegree(bb.num_node, 0);
6.     std::queue<int> nodes;
7.     while (!nodes.empty())
8.     {
9.         int node = nodes.front();
10.        nodes.pop();
11.        bbTime = bb.statements[node].sch + node.cal_time;
12.        visited[node] = true;
13.        for (int i = 0; i < bb.num_node; i++)
14.        {
15.            if (bb.matrix[node][i] == 1)
16.            {
17.                indegree[i]--;
18.                if (!visited[i] && indegree[i] == 0)
19.                {
20.                    nodes.push(i);
21.                    bb.statements[i].sch = bbTime;
22.                }
23.            }
```

```
        }
        bb.sch_total = bbTime;
    }
```

2.4 寄存器绑定

高层次综合要尽可能实现寄存器的复用。我们需要根据数据流判断该数据的生命周期，生命周期存在重叠的数据无法共享寄存器。

本设计将变量分为**函数参数变量、全局变量、局部变量、常量**。对于函数参数变量，由于其一定会作为模块的输入，故不需要额外寄存器；对于出现在多个基本块之间的变量，称为全局变量，为节约程序运行时间，我们将这类全局变量的每个变量用一个寄存器单独存储；对于只出现一个模块的变量，称为局部变量，通过分析局部变量的生命周期，利用左边算法进行寄存器共享；对于 IR 中间表示中出现的常量，在生成 RTL 时可以直接使用该常量表示，故可以认为常量的寄存器就是常量本身。

```
Binding::Binding(GraphGroup& graph_group) {
    // 1. 找到每个 BB 的全局变量、常量、函数参数、局部变量
    bfs(graph_group);
    graph_group.set_global_vars(global_vars);
    regs_num = global_vars.size();
    // 2. 对局部变量，进行生命周期分析，绑定到局部寄存器
    for (int i = 0; i < graph_group.size(); i++)
    {
        Graph& bb = graph_group.get_graph(i);
         local_binding(bb);
     }
     graph_group.set_regs_num(regs_num);
     // 3. 对全局变量，直接绑定到全局寄存器
     for (int i = 0; i < graph_group.size(); i++)
     {
         Graph& bb = graph_group.get_graph(i);
         global_binding(bb);
     }
}
```

Binding 算法的核心如上，首先，通过广度优先搜索找到各个类型的变量，并将局部变量存储到 Graph 类中；其次，对每个基本块的局部变量进行生命周期分析并绑定；最后对每个基本块的全局变量进行寄存器绑定。寄存器绑定的结果存储到基本块 Graph 类的 map<string, string> var_reg 成员变量中。

2.5 RTL 代码生成

2.5.1 端口定义

本设计认为，任何使用 IR 高层次综合得到的模块都具有时钟、复位、开始标识信号、结束标识信号这几个端口。开始标识是一个输入端口，置于高电平即可使模块开始工作；当模块计算完成后则将结束标识信号置于高电平，直到复位或者下次运行任务的开始。

IR 函数的标量整型形参对应模块的一个 32 位宽的输入端口。对于向量整型形参，认为使用了但端口 ram，包括 32 位宽的输入输出数据端口、写使能端口、读使能端口、地址位端口（程序默认地址位宽为 16 位）。IR 函数的整型返回值对应模块的一个 32 位宽的输出端口；void 类型的返回值则没有数据输出端口。

输出结果如下（可以运行代码，结果在根目录/result/dotprod.v 中）

```
1.  input clk,
2.  input rst,
3.  input start_sig,
4.  // sram a
5.  input  [32-1:0] a_in,
6.  output reg [32-1:0] a_out,
7.  output reg    a_wr_en,
8.  output reg    a_rd_en,
9.  output reg [16-1:0] a_addr,
10.  // sram b
11.  input  [32-1:0] b_in,
12.  output reg [32-1:0] b_out,
13.  output reg    b_wr_en,
14.  output reg    b_rd_en,
15.  output reg [16-1:0] b_addr,
16.  input  [32-1:0] n,
17.  output reg [32-1:0] result,
18.  output reg    done_flag
```

2.5.2 非端口寄存器声明

模块的非端口寄存器分为以下几类：

1. 操作数绑定的寄存器。在之前寄存器和操作数绑定时，可以确定每个操作数绑定到的寄存器，从而也可以确定所需要的总寄存器数量。

2. 状态机寄存器，包括 state，sub_state，pre_state，branch_state。其中 state 表示当前所在基本块代表的状态；sub_state 表示当前基本块内的调度周期；pre_state 记录上一个基本块代表的状态；branch_state 表示即将跳转到的目的基本块状态。这些寄存器的位宽根据具体的状态数来确定。

```
// regs
reg [32-1:0]reg0;
reg [32-1:0]reg1;
reg [32-1:0]reg2;
reg [32-1:0]reg3;
reg [32-1:0]reg4;
reg [32-1:0]reg5;
reg [32-1:0]reg6;
reg [32-1:0]reg7;
// state registers
reg [3-1:0]state;
reg [3-1:0]prev_state;
reg [3-1:0]branch_state;
// sub state
reg [3-1:0]sub_state;
```

2.5.3 状态机编写

本设计采用一段式状态机编写，核心处理状态转换和状态内各个算子对应的RTL 代码。

综合得到的状态机分成两层：第一层将基本块映射为状态，同时加上头尾两个开始和结束状态；第二层是一个微控制器，根据各基本块所需要的调度周期确定微控制器的状态数量，每个周期对该周期运行的算子进行综合。对第一层状态机的做法如下：

1. 头部的空基本块对应的状态机在开始标识信号被置高之前空转，接收到开始识信号后将状态切换到第一个基本块对应的状态

2. 尾部的空基本块对应着状态机的最大状态值，在该状态下，状态机将结束，在下一个周期回到第一个状态。

3. 除了首尾空基本块对应的状态，状态机每进入一个新的状态前会将状态

4. 寄存器 state 赋值为目标状态寄存器 branch_state 的内容，并将前次状态寄存器 prev_state 赋值为当前的状态值，同时将子状态寄存器 sub_state 赋值为 0。

对于每个状态的算子，需要插入到微控制器对应的周期处，下面介绍对一些关键算子的映射。

1. br：先将分支名映射为状态值，然后赋值给 branch_state 寄存器。有条件的 br 需要综合为条件运算符，无条件的 br 则采用直接赋值。

2. load：由于 load 操作需要经过两周期（使能+得到操作数）完成，因此需要在 load 指令所在子状态的下一个子状态综合一段操作 ram 数据输入端口，并把使能信号调低的代码。

3. store：store 指令与 load 指令类似，不再赘述。

4. phi：phi 操作需要通过 prev_state 寄存器选择数据的来源，因此将 phi 操作符综合为一段 case 语句。

3 功能仿真及分析

为了验证所生成 RTL 代码的正确性，本设计使用 Vivado 生成了两个但端口 ram IP，并编写了顶层模块 top.v 以实现 ram IP 数据的初始化以及 ram IP 和 dotprod.v 模块的连接。同时，编写了顶层模块的仿真文件 tb_top.v，用 vivado 进行了功能仿真并得到对应的波形。

对于题目所给的 LLVM IR，在仿真时，我们设定以下参数，其中 i=0~9

表 2:模块参数设定

参数	a[i]	b[i]	n
数值	i	i	10

分析 LLVM IR 可知，改程序等价于运行如下公式

$$\mathrm{cr} = \sum_{i=0}^{n-1} a_i * b_i$$

3.1 顶层模块编写

在顶层模块中，我们引入 ram_init，a_ram_in，a_addr_in 等信号作为输入，以便实现 ram 的初始化。代码的逻辑是首先将 ram_init 置高电平，然后依次传入（地址，数据）对从 0-9，然后将 ram_init 置为低电平，并将 start_sig 置为高电平，此后状态机正常运行并实现和 ram 的交互，观察输出信号 done_flag 和 result 是否符合预期。

值得注意的是，本次设计中认为 load 和 store 指令的执行周期为 2，但通过绘制时序图（图 3）可以发现，若 ram 也采用时钟上升沿，则第三个周期 dotprod.v 才能将 ram 中的数据存储到寄存器。因此，我们将 ram 的时钟更改为下降沿，这样在第二个时钟上升沿寄存器就能顺利存储来自 ram 中的数据（图 4）。

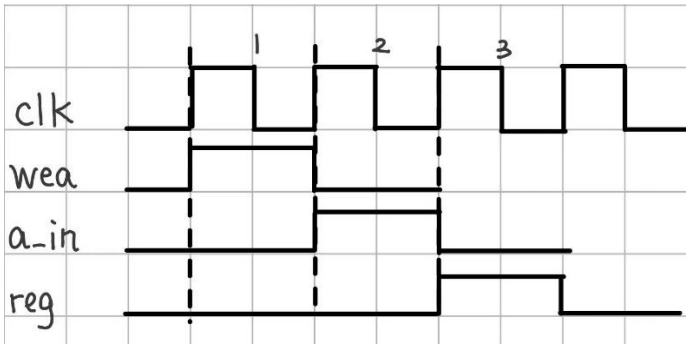


图 3：时钟均为上升沿时序分析

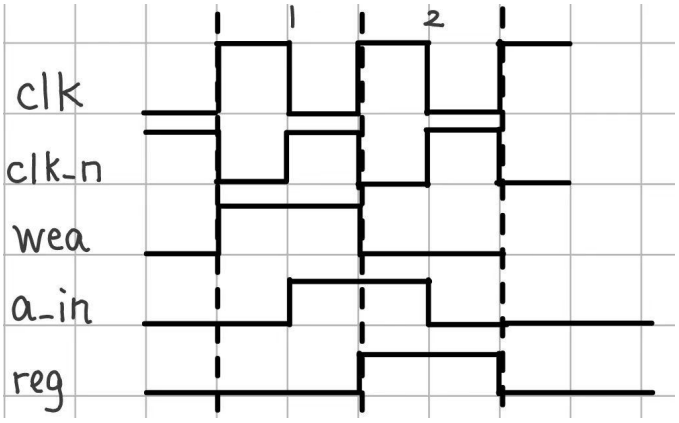


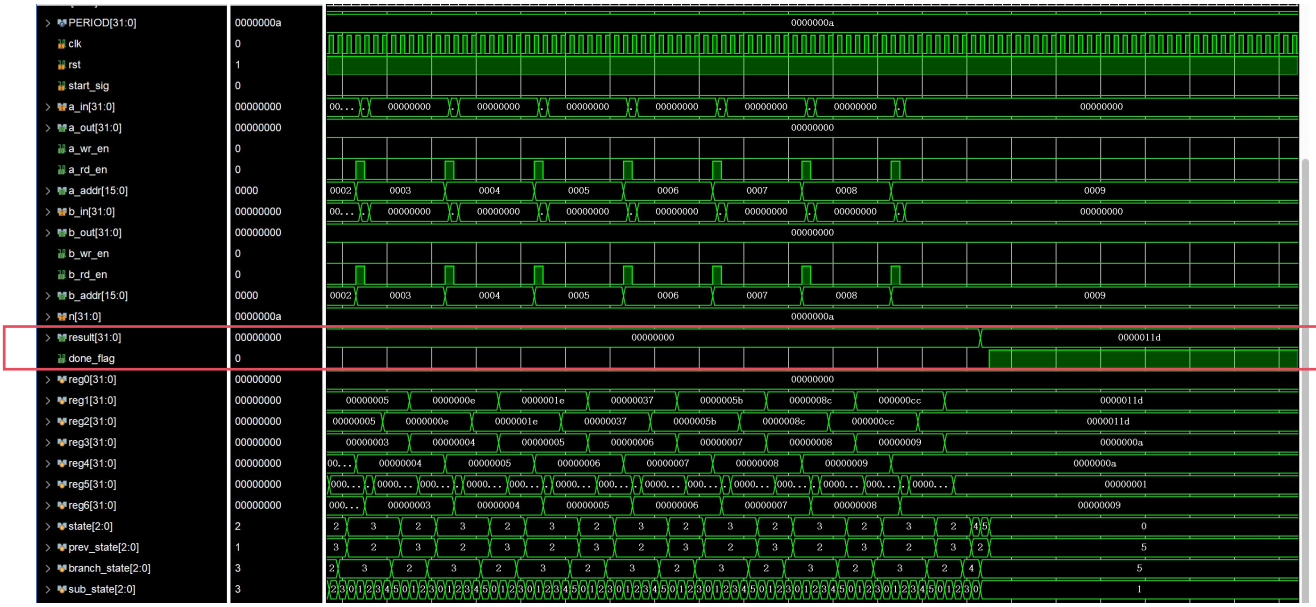
图 4：ram 使用下降沿时序分析

```
1. module top(
2. input sys_clk,
3. input sys_rst_n,
4. input ram_init,
5. input start_sig,
6. input [32-1:0] n,
7. input [32-1:0] a_ram_in,
8. input [32-1:0] b_ram_in,
9. input [16-1:0] a_addr_in,
10. input [16-1:0] b_addr_in,
11. output [32-1:0] result,
12. output done_flag
13.     );
14. //wire define
15. wire sys_clk_n;
16. wire [32-1:0] a_in;
17. wire [32-1:0] a_out;
18. wire a_wr_en;
19. wire a_rd_en;
20. wire [16-1:0] a_addr;
21. wire [32-1:0] b_in;
22. wire [32-1:0] b_out;
23. wire b_wr_en;
24. wire b_rd_en;
25. wire [16-1:0] b_addr;
26. wire wea;
27. wire web;
28. wire [32-1:0] dina;
29. wire [32-1:0] dinb;
30. wire [16-1:0] addra;
31. wire [16-1:0] addrb;
32.
33. assign sys_clk_n = ~sys_clk;
```

```
34. assign wea = ram_init || !a_rd_en;
35. assign web = ram_init || !b_rd_en;
36. assign dina = ram_init ? a_ram_in : a_out;
37. assign dinb = ram_init ? b_ram_in : b_out;
38. assign addra = ram_init ? a_addr_in : a_addr;
39. assign addrb = ram_init ? b_addr_in : b_addr;
40. 
41. dotprod dotprod_0(
42.  .clk(sys_clk),
43.  .rst(sys_rst_n),
44.  .start_sig(start_sig),
45.  // sram a
46.     .a_in(a_in),
47.  .a_out(a_out),
48.  .a_wr_en(a_wr_en),
49.  .a_rd_en(a_rd_en),
50.  .a_addr(a_addr),
51.  // sram b
52.     .b_in(b_in),
53.  .b_out(b_out),
54.  .b_wr_en(b_wr_en),
55.  .b_rd_en(b_rd_en),
56.  .b_addr(b_addr),
57.  .n(n),
58.  .result(result),
59.  .done_flag(done_flag)
60. );
61. blk_mem_gen_0 ram_a(
62.   .clka(sys_clk_n),    // input wire clka
63.   .ena(1),      // input wire ena
64.   .wea(wea),       // input wire [0 : 0] wea
65.   .addra(addra),  // input wire [15 : 0] addra
66.   .dina(dina),    // input wire [31 : 0] dina
67.   .douta(a_in)  // output wire [31 : 0] douta
68. );
69. blk_mem_gen_0 ram_b(
70.   .clka(sys_clk_n),    // input wire clka
71.   .ena(1),      // input wire ena
72.   .wea(web),       // input wire [0 : 0] web
73.   .addra(addrb),  // input wire [15 : 0] addrb
74.   .dina(dinb),    // input wire [31 : 0] dinb
75.   .douta(b_in)  // output wire [31 : 0] doutb
76. );
77. endmodule
```

3.2 testbench 测试结果

如图，可以看到 result 的最终结果为 285,符合预期！



4 总结

此次设计实现了完成了高层次综合后端全流程，程序采用 ASAP 调度算法，通过广度优先搜索分析各类型变量，通过左边算法实现寄存器和操作数的绑定，通过 Vivado 得到 ram IP 并进行封装和仿真，成功检验了程序功能的正确性。

附录：小组成员分工说明

本次设计由两人共同完成，工作量相同。