Project 1: Implement Your Own Modulo Scheduler

October 2, 2023

Contents

1 Introduction

Welcome to the first project of Synthesis of Digital Circuits. The goal of this project is to practice the theory of HLS scheduling we have covered in the lectures.

Your task is to implement different scheduling algorithms. The end goal is to create a modulo scheduler that can handle complex design requirements. The project template provides you with the LLVM-generated IR files of 4 C++ HLS kernels. The framework parses these IRs, solves the ILP formulation that you will create, and creates Gantt charts of the ILP solution.

Each task in this project involves implementing one or more functions needed to create the final schedule. These include specifying the data dependency constraints, the ILP formulation, and the ILP objective function, for different scheduling methods.

The framework is written in Python, therefore your solutions will also have to be written in Python. If you are not familiar with Python or programming in general, it is highly recommended that you familiarize yourself with basic programming concepts and the usage of Python before starting this project.

You will need to submit your scheduler implementation as well as a PDF report answering the additional questions for each task.

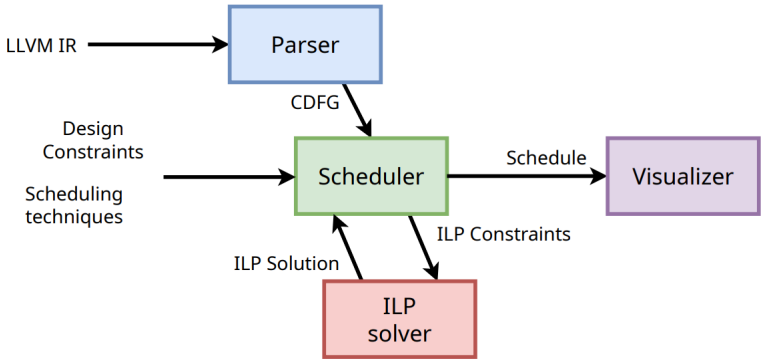
2 Scope of the Project

As a whole, the tasks in this project cover the following challenges:

- Identifying data, control and loop-carried dependencies in the IR.
- Formulating systems of difference constraints (SDC) from the dependency constraints.
- Formulating the objective functions for classic scheduling problems (ASAP, ALAP).
- Implementing heuristic resource-constraints.
- Implementing an incremental method of II identification for pipelined scheduling.

- Implementing heuristic resource-constraints with pipelined execution.

Figure 1: Overview of our scheduler.



The input to the Scheduler is the Control Flow Graph (CFG) and Control Flow Data Graph (CDFG). Both can be rendered by graphviz-dot for quick visualization.

The output of the scheduler is a Gantt chart, showing the execution cycle of each node.

The template project already calls the ILP solver and prints out the visualization. The code you must add is to convert the CDFG into the ILP formulation to be solved.

There are 4 kernels to test your code on with the tester. You should only evaluate the pipeling code on kernels 3 and 4, as kernels 1 and 2 do not contain loops.

3 Deliverables

Please add your implementations into the following files:

- proj_fld/run_SDC.py
- proj_fld/src/main_flow/scheduler.py
- proj_fld/src/main_flow/resource.py

The other files should not need any alterations to complete the tasks.

Running each of the methods for each of the kernels will generate Gantt charts and schedule files automatically. These are important for evaluating your code.

In addition, each task has several questions to be answered in a report. Please zip your final project directory and your report together, and submit the resulting file.

4 Control Data Flow Graphs

The two high level aspects of the intermediate representation (IR) generated from a C++ HLS kernel are the Control Data Flow Graph (CDFG) and the Control Flow Graph (CFG). Both are member variables of the Scheduler, and can be accessed with

```
    self.cdfg
    self.cfg
```

in any of its member functions.

If you iterate over the CDFG, it will give you every node in the program, but in an arbitrary order:

```
    for node in self.cdfg:
        print(node)
```

If you iterate over the CFG, it will give you every BB in an arbitrary order:

```
    for bb in self.cfg:
        print(bb)
```

The CDFG also allows you to iterate over every node, separated out by BB, by accessing its subgraph member function.

```
    for bb in self.cdfg.subgraphs():
        for node in bb:
            print(node)
```

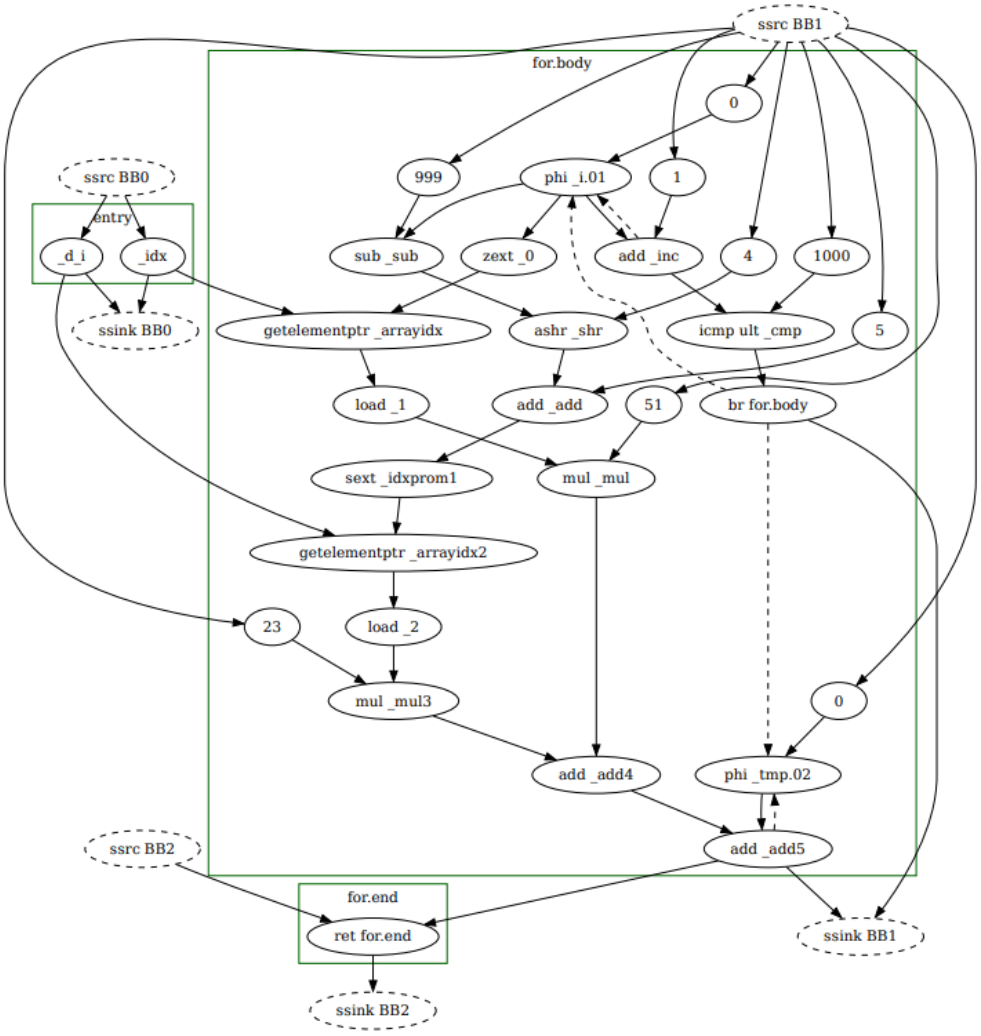
An example kernel and CDFG can be seen below.

Figure 2: A HLS Kernel.

```
int fir (in_int_t d_i[1000], in_int_t idx[1000]) {
  int i;
  int tmp=0;

  for (i=0;i<1000;i++) {
    tmp += (idx [i] * 51) + (d_i[((999-i)>> 4)+5] * 23);
  }
  return tmp;
}
```

Figure 3: The Kernel's CDFG



Here is a list of node types that we have considered in our CDFG:

- **Operator nodes:**

  They directly correspond to the LLVM instructions inside a basic block (BB), e.g., mul, add, sub, etc. (for instance, "mul mul3" in the example).

- **Memory nodes:**

  They directly correspond to the LLVM memory operations load and store. In this project, we do not distinguish memory nodes from operator nodes and treat them the same way("load 2" in the figure).

- **Control nodes:**

  They correspond to LLVM control flow instructions at the boundary of the BBs, including br and phi ("phi i.01" in the example).

- **Auxiliary nodes:**

  They are dummy nodes that are added into the graph to enforce strict separation of BBs. We introduce two auxiliary nodes: supersink and supersource. A supersource is served as the first node of BB. A supersink is served as the last node of a BB.

Inside a CDFG, the edges are used to denote dependencies between different nodes.
We differentiate the type of edges by their line style, i.e., solid-lines or dashed-lines.
Solid lines denote regular data dependencies whereas dashed lines represent back edges.

5 Running the Scheduler

The scheduler is ran by executing the following terminal command in the root folder of the project:

```
python run_sdc.py
```

The only command line argument you should need is:

- `--methods`

  Specify which scheduling techniques to use.

  Allowed values:

  - asap
  - alap
  - asap_rconst
  - pipelined
  - pipelined_rconst
  - all

In addition, you must specify which kernels to schedule in "filelist.lst". By default, it contains kernel_1.

6 Using the Tester

The usage of the tester script is fairly straightforward. It can be run by executing the following terminal command, also in the root directory of the project, and will automatically evaluate your code.

```
    python tester.py
```

The tester has certain options which control its behavior. These options are:

- `--kernels`

  The space-separated list of kernels that the tester should use. Each kernel is specified using its name followed by a comma and a number indicating whether it should be used for pipelined tests. Example value: kernel 1,0 kernel 3,1

- `--methods`

  The scheduling methods that the tester should test. Multiple methods can be specified in the form of a space-separated list. Possible values: asap, alap, asap rconst, pipelined, and pipelined rconst.

  Example value: asap alap

- `--iimax`

  The pipelined scheduler test will incrementally raise the II until a solution is found but it will automatically fail if the II reaches iimax.

  Specifying a lower value will reduce the runtime but keep in mind that specifying a value lower than the lowest possible II will make the test fail even if your code is correct.

  This value is 40 by default.

- `-v or --verbose`

  Prints extra information regarding the tests being run. Off by default.

- `--no-name-check`

  Makes the tester not check if the supernodes were named appropriately. Off by default.

- `--no-snode-conn`

  Makes the tester not check if the supernodes were connected to other nodes. Off by default.

- `--no-print-suppress`

  The tester suppresses print statements called from within the code being tested by default. Specifying this option disables this behavior.

7 Task 1: Inserting Artificial Nodes

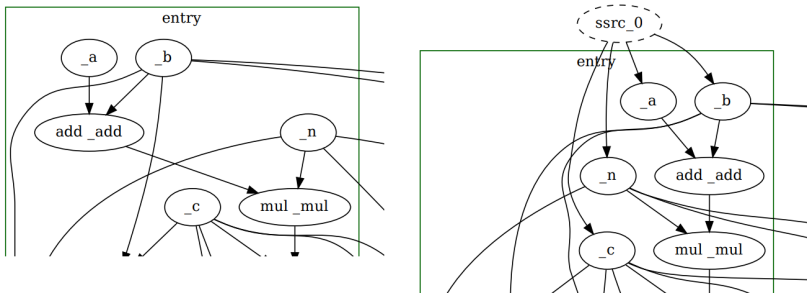
The first task is to write the member function

```
    def add_artificial_nodes(self):
```

in the Scheduler class. You can add auxiliary member functions to the Scheduler class if you'd like, but you don't need to change the rest of the existing code.

In order to ensure strict separation of BBs, we need to add 2 artificial nodes per BB: A supersource, and a supersink.

Figure 4: Kernel 1's BB0 after adding supersource



You'll need a bbID, bbLabel, and a name variable to add the artifical nodes.

```
    #todo: somehow get the bbID
    #todo: somehow get the bbLabel

    supersource_name = f"ssrc_{bbID}"
    supersink_name = f"ssink_{bbID}"
```

Please use the given supersource and supersink name, as it helps the tester.

bbID and bbLabel can be accessed from a BB through its id and bbID attributes, which are stored in its member variable dictionary attr.

```
    for bb in self.cfg:
        print(bb.attr["id"])
        print(bb.attr["bbID"]
```

Despite their index names, "id" returns a numeric id of the BB, and "bbID" returns a label string, such as "entry" in Fig. 4.

The numeric id of the BB is also stored on every node, also on its member variable dictionary attr.

```
    for node in self.cdfg:
        print(node.attr["id"])
```

7.1 Adding the nodes

Please use the following member function to add the artificial nodes:

```
    self.cdfg.add_node(supersource_name, id=bbID, bbID=bbLabel, type="supersource",
    label=supersource_name)
    self.cdfg.add_node(supersink_name, id=bbID, bbID=bbLabel, type="supersink", label=
    supersink_name)
```

The first argument is the name which will show up on the Gantt chart, and the last argument is what will show up on the printed CDFG.

The "type" of the node is used to get its latency during scheduling, and other possibilites are other operations such as add, mult, etc.

You only need supersource and supersink for this task, however.

And remember, we went a supersource and supersink node for each BB.

7.2 Identifying which nodes to connect

Any node with no predecessors in the same BB must be a successor of the supersource, and any node with no successors in the same BB must be a predecessor of the supersink.

The predecessors and successors of a node can be found using:

```
    #predecessors
    for pred in self.cdfg.in_neighbors(node):
        print(pred)

    #successors
    for succ in self.cdfg.out_neighbors(node):
        print(succ)
```

However, back-edges only have a data dependency when the BB is pipelined, and so we ignore these connections when deciding whether to connect to the artificial nodes.

You can get the edge between two nodes with the following function:

```
    edge = self.cdfg.get_edge(node, successor)
```

You can tell if an edge is a back-edge by checking if its style attribute is "dashed".

```
    isBackEdge = edge.attr["style"] == "dashed"
```

7.3 Adding the edges

Once you have identified which nodes should be connected to a supersource or supersink, please use the following member function to add the corresponding edges.

```
    #if you don't have the supernode variables saved
    for node in self.cdfg:
        if shouldBeConnectedToSupersource(node):
            #todo: get bbID from this node
            self.cdfg.add_edge(f"ssrc_{bbID}", node)
        if shouldBeConnectedToSupersink(node):
            #todo: get bbID from this node
```

```
            self.cdfg.add_edge(node, f"ssink_{bbID}")

    #if you have the supernode variables
    for node in self.cdfg:
        if shouldBeConnectedToSupersource(node):
            self.cdfg.add_edge(correct_supersource, node)
        if shouldBeConnectedToSupersink(node):
            self.cdfg.add_edge(node, correct_supersink)
```

7.4 Testing

The testing script won't work until you have completed Task 2.

Use the following code to print out the CDFG to examine it manually.

```
    self.cdfg.layout(prog='dot')
    self.cdfg.draw('output.pdf')
```

This is already present in the file.

8 Task 2: ASAP Scheduling

8.1 Add Every Node to the ILP Formulation

In order for a node to be scheduled, the first step is to add it to the ILP formulation. The formulation is accessible as a member variable in the Scheduler class:

```
    self.ilp
```

The member function to add a scheduling variable to the ilp is:

```
    self.ilp.add_variable(f"sv{node}", lower_bound=0,var_type="i")
```

Again, please follow the scheduling variable naming convention.

The lower bound sets the lowest possible start time of a node, and we are only allowing integer start times, so the var_type is "i".

8.2 Write Data Dependency Constraints Function

Encoding the dependencies is done by altering the set_data_dependency_constraints member function in the Scheduler.

For each BB, please encode all the data dependency constraints between nodes in that BB. Two nodes in two different BBs should not have their data dependency added to the ILP, as the scheduling of each BB is done independently.

Additionally, nodes connected by a back-edge only have a data dependency when the BB is pipelined. At this stage of the project, we do not yet want to add a constraint for these connections.

Section 7.2: Identifying which nodes to connect, has the code necessary to identify data dependencies between nodes.

For a node B that depends on node A, B cannot start before A finishes.

To add this constraint to the ILP, you must encode it in an inequality with the start times of A and B on the left hand side, an inequality sign, and the latency of A on the right hand side.

The valid inequality signs are less than or equal, equal, and greater than or equal.

Constraints are added through the following function, which is assumed to be called from the Scheduler. add_constraints is a member function of the Constraints class, which is instantiated as a member variable in the Scheduler.

```
    self.constraints.add_constraint(lhs_dictionary, inequality_sign, rhs)
```

The inequality_sign variable should be a string:

```
    #less than or equal
    inequality_sign = "leq"
    #equal
    inequality_sign = "eq"
    #greater than or equal
    inequality_sign = "geq"
```

The right hand side (rhs) variable must be an integer, and in this case should be the latency of node A, which can be obtained through the following function:

```
    rhs = get_node_latency(A.attr)
```

get_node_latency is a non-member function imported from cdfg_manager. Notice that it takes the member dictionary attr as input, not the node itself.

The left hand side dictionary (lhs_dictionary) has a separate entry per scheduling variable. The key should be the scheduling variable, and the valid values are either 1 or -1.

You must choose the correct values for each scheduling variable to encode the correct constraint.

```
    #instantiate an empty dictionary
    lhs_dictionary = {}
    #to add start time of A to inequality
    lhs_dictionary[f"sv{nodeA}"] = 1
    #or instead to add the negative start time of A to inequality
    lhs_dictionary[f"sv{nodeA}"] = -1
```

8.3 Call the Data Dependency Constraints Function

The set_data_dependency_constraints member function function must be called from the create_asap_scheduling_ilp function.

While this is a trival step for ASAP scheduling, the create scheduling functions will become slightly more complex, so it is helpful to know this step exists.

8.4 Set the Objective Function

Every scheduling variable you wish to consider in the objective function must be added, along with a coefficient. Variables with a positive coefficient will be minimized, and variables with a negative coefficient will be maximized.

Please alter the set_asap_obj_function in order to do so. The function call is already written for you, in the set_obj_function function.

The member variable obj_fun of the Scheduler contains the member function add_variable, which you should make use of.

```
    self.obj_fun.add_variable(f"sv{node}", coeff}
```

8.5 Testing

Run run_sdc and the tester to double check the Gantt charts to see if the resulting schedules satisfy your goal of scheduling every node ASAP.

8.6 Report Questions

1. Is the obtained latency always minimal?
2. Why should the scheduling problem be solved individually for each BB?

9 Task 3: ALAP Scheduling

ALAP differs from ASAP in two ways: it has a different objective function, and each BB must be constrained to a maximum latency.

9.1 Read the ALAP Function

The ALAP function is a non-member function in run_sdc. It runs ASAP scheduling first, and then asks the scheduler for the execution cycle of the supersinks.

Does this help with constraining the maximum latency of each BB?

9.2 Retrieve the Execution Cycle of the Supersinks

Please alter the get_sink_sv function, which is a member function of the Scheduler.

You need it to return a data structure with the information for writing the maximum latency constraint.

You'll need to be able to access the result of the scheduling variable for a given node. The function to do so is:

```
self.ilp.get_operation_timing_solution(node)
```

9.3 Write the Maximum Latency Constraints Function

Please alter the add_sink_sv_constraints function, to add a maximum value constraint to the scheduling variable of the supersinks. This should be based on the supersink's scheduling variable's value in the ASAP execution.

This is one way to add the maximum latency constraint to each BB.

9.4 Create ALAP Schedule Constraints

Use the set_data_dependency_constraint function and the add_sink_sv_constraints function in the create_alap_scheduling_ilp function to create the ALAP constraints.

9.5 Set the Objective Function

Alter the set_alap_objective_function in order for the ILP to find an ALAP solution.

Choose the correct coefficients so that the execution cycle of each node is maximized.

9.6 Testing

Call run_sdc and the tester, and view the Gantt chart. Does your code past the tests? Are the nodes scheduled properly on the Gantt chart?

9.7 Report Questions

1. Comment on the achieved latency with respect to the ASAP latency.
2. Comment on the slack that you observe. Is this information useful? How could you use it?
3. For the provided examples, which solution is more desirable in terms of area-performance tradeoff, ASAP or ALAP?

10 Task 4: ASAP with Resource Constraints

Please alter the add_resource_constraints member function in Resources.

The parameter

```
self.resource_dict
```

contains which resources are constrained as keys, and the maximum number as values.

You can check which type of operation a node is from its type attribute:

```
for node in self.cdfg:
    print(node.attr["type"])
```

10.1 Sort the Operation Nodes

We have provided you with the sorting function

```
get_topological_order(self.cdfg)
```

which is a non-member function imported from cdfg_manager. The cdfg is also available as a member variable of the Resources class.

This will give you an approximation of which order the operations will happen in.

While it will work well on some kernels, it will perform badly on other kernels. The effects of this may not become apparent until you reach pipelining, but make a mental note now that you may need to alter this ordering later.

10.2 Constrain the Operation Nodes

Assuming pipelined operators, the number in the resources dictionary affects how many operations can start in the same cycle, not how many can run concurrently.

Based on your ordered operations, constrain the scheduling variables so this maximum value is respected.

10.3 Testing

Call run_sdc and the tester for resource constraining, and look at the Gantt charts. Did your code pass the tests? Are your resource constraints respected?

10.4 Report Questions

1. Vary the "add", "mul" and "zext" resource constraints from 1 to 3 (try all possible constraint combinations). Highlight interesting cases and compare area and latency with the results of Task 1. How do the resource constraints affect the schedule?

2. Which resource(s) influence the schedule and with which maximum number of allowed operations?

11 Task 5: Pipelined Scheduling

The ILP solver only considers the starting time of each node in a single iteration. In order to pipeline a BB, we must find the minimum II which does not violate our inter-iteration data dependencies.

There are two ways to do this. Firstly, we could add the II to the objective function as an additional variable to minimize. Or we can iteratively run the ILP, each time specifiying the II to a single value, until the ILP is able to find a valid solution.

The issue with the first solution is that the minimum II may require that individual nodes start later than for an unpipelined approach. The ILP will consider this a trade-off to make, depending on how the II is weighted in the objective function, and may return an II that is larger than the minimum possible II.

We will therefore take the second approach. This means the objective function is unchanged: it still wants to schedule each node ASAP, just now with more constraints.

Since the II is fixed for each ILP iteration, it does not needed to be added to the ILP formulation and instead can be hard-coded into the right hand side of the constraints.

11.1 Implement the inter-iteration Data Dependency Constraints function

Alter the set_pipelining_constraints functions to encode the inter-iteration data dependency constraints.

A inter-iteration data dependency is represented by a back-edge on the CDFG. As a reminder, you can find out if nodes are connected by a back-edge by:

```
edge = self.cdfg.get_edge(node, successor)
isBackEdge = edge.attr["style"] == "dashed"
```

11.2 Building the constraint

Remember, the constraint for a cross-dependency data dependence is that

1. The finishing time of the data generating node in the FIRST iteration
2. must be before or at the same time as
3. The starting time of the data consuming node in the SECOND iteration.

The notes discuss dependencies separated by multiple iterations, but for now you can assume dependence in the next iteration, e.g. the iteration distance is always 1.

The constraints should be added by altering the set_II_constraints member function in Scheduling.

11.3 Create Pipelining Schedule Constraints

Call functions needed to add the intra-iteration and inter-iteration constraints to the ILP formulation from the create_pipelined_scheduling_ilp member function in the Scheduler.

11.4 Find Loop BB

Alter the find_loop_bb function to return the bbID (numeric) of the BB that is being pipelined.

In this project, the BB being pipelined is always the BB with the longest latency.

11.5 Testing

Call run_sdc and the tester for pipelining, and open the Gantt charts to double check your solution.

Remember, only kernels 3 and 4 have loops, so don't run the pipelining method on kernels 1 or 2.

11.6 Report Questions

1. Assuming N loop iterations, what is the total latency of the program? Comment on the latency and area.
2. What are the achievable IIs?

12 Task 6: Resource Constraints with Pipelining

There is no simple way to generate a schedule which respects inter-iteration resource constraints.

For this project, we will iteratively generate schedules and then check if our constraints are respected. This is therefore a two step process.

In order to validate the inter-iteration constraints, a Module Reservation Table (MRT) can be used.

The first step is to find a candidate schedule using ILP. Both intra-iteration and inter-iteration data dependency constraints are considered. Resource constraints are added, but only for intra-iteration. To repeat for emphasis: The ILP step considers resource usage collisions inside an iteration, but not between multiple iterations.

The construction of an MRT follows rules that mirror resource usage in a pipelined schedule. It is used to observe a given schedule's resource requirements.

If the desired resource constraints are not respected, a new schedule must be made.

The simplest way to do this is to increase the minimum II and try again.

The MRT is made up of II columns and one row for each individual resource unit.

If a resource type is not constrained, operations of that type can be ignored.

MRT construction has the following rules:

- All operations must be placed in the MRT.
- Each cell can contain only one operation.
- Each operation must be inserted in a row of a unit that can execute it.

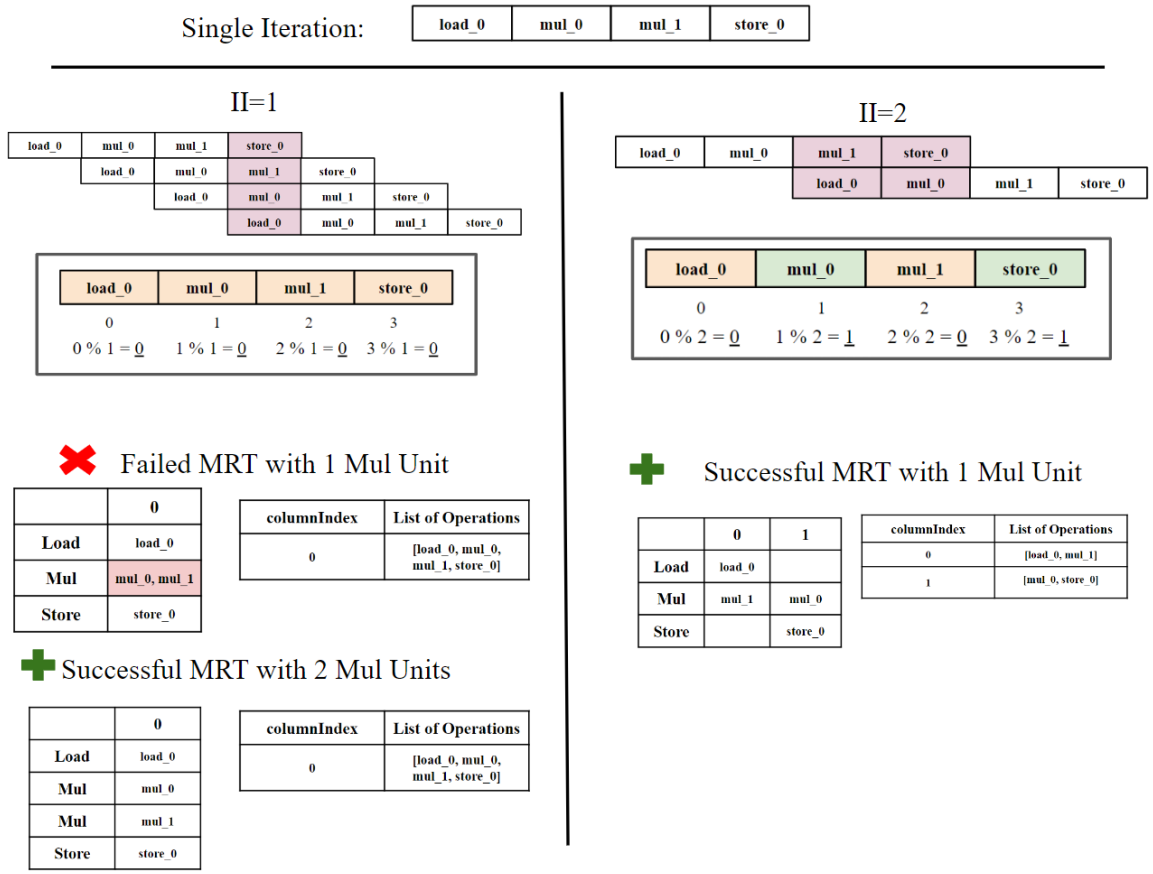
- The column is chosen by the formula: columnIndex = sv_node % II

Hence, the name "Modulo Reservation Table".

If any of the previous rules cannot be respected, the resource constraints check fails and the II must be increased.

In Fig. 5, there are three examples which showcase the result of the MRT construction.

Figure 5: Sample Modulo Reservation Tables



12.1 Test a Schedule using a Modulo Reservation Table

Please alter the check_resource_constraints_pipelined function in the resource manager.

It should return false if the MRT construction fails, and true if it succeeds.

In practice, the easiest way to build an MRT is to create a dictionary of lists.

Then for each node in the graph:

1. Use the calculated columnIndex as the key.
2. And add the node's type to the list at that key.

Then, once finished, check if the list at each columnIndex has more than the maximum number of each constrained resource type. If it does, the minimum II must be increased and scheduling re-ran.

Sample dictionaries can be seen in Fig. 5

Remember, the function to get the result of the scheduling variable for a given node is:

```
self.ilp.get_operation_timing_solution(node)
```

12.2 Alter Node Ordering

Now that you have resource constrained pipelining, you may see kernel 3's II has increased quite a lot.

Investigate why, and see if you can fix this by reordering the operations for resource constraining.

There are many ways to convert a graph to a heuristic operation order, so there is no single correct solution to this.

12.3 Testing

Call run_sdc and the tester to evaluate your code. Are the pipelined resource constraints respected?

12.4 Observe the Gantt Charts

Is the II larger than you would expect by looking at the Gantt chart? Remember that the chart only shows 2 iterations, and there may be a resource collision in a later iteration that required a higher II. See if you can find an avoided resource collision that isn't obvious from the final Gantt chart.

12.5 Report Questions

- Does the MRT heuristic compute the optimal scheduling? Motivate your answer.
- Which feature of the MRT heuristic has the most significant effect on the scheduling?
- How could the MRT heuristic be improved?
- What change did you make to the operation ordering heuristic to reduce the II?
- Write down some observations from Section 12.4 Observe the Gantt Charts.

13 Submission

If your code has passed all of the tests, and you have answered all of the questions in your report, then congratulations! You have finished this project.

Please zip your final project directory and PDF report and submit the resulting file.

14 Implementation Notes and Conventions

The following section formally specifies the assumptions made and conventions followed.

14.1 Assumptions

- The execution latencies of all operators are given and fixed.
- Memory accesses are treated exactly like normal operators (i.e., you can assume that memory accesses are independent, can happen out of order, and the amount of memory ports are not limited i.e. there is no limit on how many memory operations can happen simultaneously).
- We assume that the critical paths of the components are the same, and all multi-cycle operations are pipelined (with II = 1).
- BBs always happen sequentially, i.e., a BB starts only after its active predecessor completes (e.g., if BB1 is followed by BB2, all operations from BB1 must execute before all operations of BB2). This is valid even if there are operations in a BB that theoretically could run before one of their BB's predecessors concludes while honoring the constraints.
- For all tasks, the solution should be reported as the start cycle of each operation with respect to the beginning of its BB. In other words, the Gannt chart/schedule of each BB should start from the time index 0.
- Constant nodes are considered as immediate operations (i.e. their combinatorial delay is 0).

14.2 ILP Problem Formulation

When formulating ILP problems for CDFG scheduling, we formally consider the following definitions:

- V_{bb} is the set of vertices representing Basic Blocks (BBs).
- V_{op} is the set of vertices representing operations.
- E_c is the set of edges representing control signals.
- E_d is the set of edges representing data signals.
- $sv(n)$is the starting execution time of operation n.
- $\forall v \in V_{op}, sv(v) \in \mathbf{N}$
- Given a CDFG $G(V_{bb} \cup V_{op}, E_c \cup E_d)$, each node $v \in V_{op}$ is associated with a set of scheduling variables $sv_i(v)$.

Additionally:

- Besides the standard CDFG dependency edges (i.e., E_c and E_d), we add two artificial nodes: super-source and super-sink nodes to indicate the start and end of a BB.
- For each node in a BB, if it has no predecessors in the same BB, then this node is connected to the super-source; if it has no successors in the same BB, then this node is connected to the super-sink.
- Supernodes of different BBs are not connected to each other.
- A back edge is a CDFG edge representing a loop-carried dependency (i.e., an operation from one loop iteration requires the result of another operation from the previous loop iteration). Back edges are ignored when adding artificial nodes (i.e., supersources and supersinks).

14.3 Resource Constraints

As any person familiar with VLSI design can attest, there are many constraints to be considered when designing digital circuits, HLS or not.

In this project, we only consider resource constraints specified as the number of functional units available. For instance, if the design is constrained to have 3 multipliers, up to 3 mul operations can be scheduled at the same time index.

The resource constraints are passed to the scheduler in the form of a Python dictionary. An example constraint with 3 multipliers and 4 adders is given below.

Figure 6: Sample Resource Constraint

```
{'mul' : 3, 'add' : 4}
```

15 Python Tips

15.1 Filtering Arrays

It is sometimes necessary to obtain only the items of an array that fulfill a certain condition.

```
bb0_nodes = [node for node in self.cdfg if node.attr["id"] == 0]
```

This structure can also be used to call a function on each element directly, without building an intermediate list.

```
[print(node) for node in self.cdfg if node.attr["id"] == 0]
```

15.2 F Strings

Python does not allow concatenation of strings with other datatypes without conversions, but F strings allow easy insertion of variable values into strings.

```
bbID = 0
bbName = f"bb_{bbID}"
#bbName has the value "bb_0"
```