Project 1: Implement Your Own Modulo Scheduler

March 11, 2025

Contents

1 Introduction

Welcome to the first project of Synthesis of Digital Circuits. The goal of this project is to practice the theory of High Level Synthesis (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 LVM-generated Intermediate Representation (IR) files of 4 C++ HLS kernels. The framework parses these IRs, solves the integer linear programming (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.

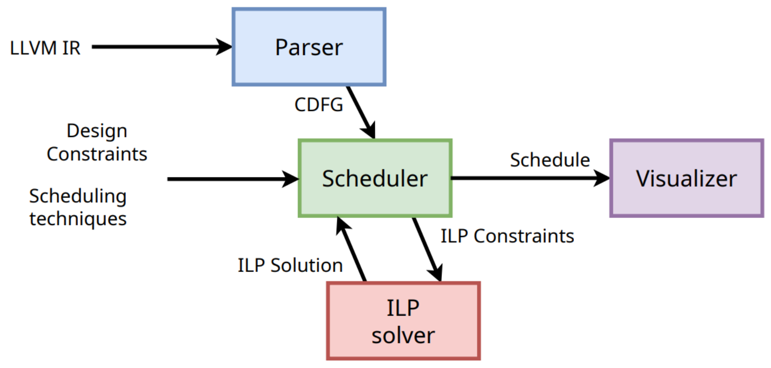
You will need to submit your scheduler implementation as well as a PDF report answering the additional questions for each task.

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 Data Flow 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 pipelining 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:

- sdc_project1/run_SDC.py
- sdc_project1/src/main_flow/scheduler.py
- sdc_project1/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 and 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. Both the CFG and CDFG are implemented as Pygraphviz graphs, which allow easy use of pre-implemented graph traversal and graph attributes.

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 of every BB by accessing its subgraph member function.

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

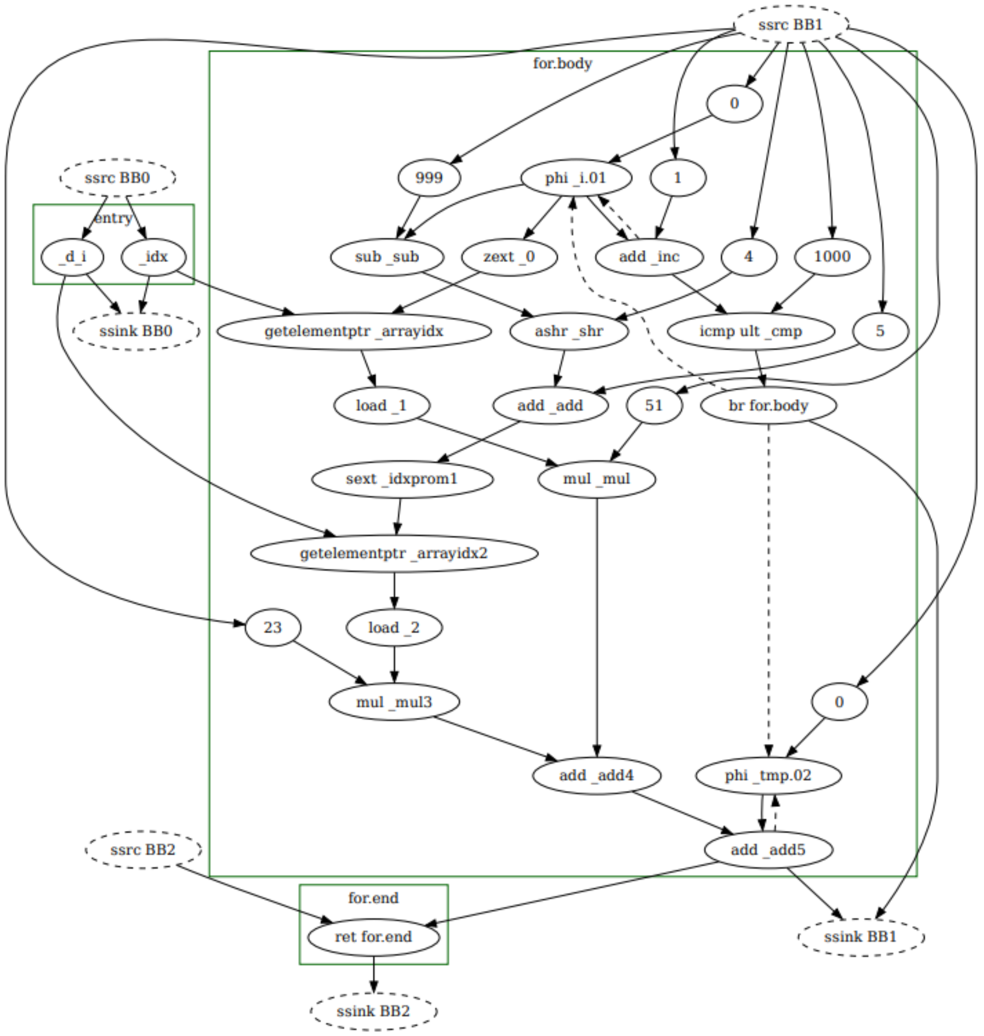
An example kernel and CDFG can be seen below.

Figure 2: An HLS Kernel.

```c
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:**

These 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:**

  These 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:**

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

- **Auxiliary nodes:**

  These are artificial nodes that are added into the graph to enforce strict separation of BBs. We introduce two auxiliary nodes: supersink and supersource. Each path through a BB should start at a supersource node, and ends at a supersink node (In the example, ”ssrc BB1” is a supersource node while ”ssink BB1” is a supersink node.)

Inside a CDFG, the edges are used to denote dependencies between different nodes.

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).

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 skip the check for supernode naming conventions. Off by default.

- `--no-snode-conn`

  Makes the tester skip the check for connections to and from supernodes. 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 Implementation Notes and Conventions

The following section formally specifies the assumptions made and conventions followed, for references purposes.

Many of these details are replicated in the relevant sections, but can be viewed here in full.

7.1 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 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).

7.2 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 $sv_i(v)$.

8 Python Tips

8.1 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]
```

8.2 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"
```

9 Task 1: Inserting Artificial Nodes

Your task for this project is to create a scheduler where each BB executes sequentially, e.g. where instructions from the next BB can only begin when every instruction from the current BB have completed. This means that you should schedule each BB separately, and that the schedule for each BB should begin at time 0. Our ILP solver, which schedules the instructions according to our constraints, needs to know this starting time. Rather than adding a starting time constraint to every instruction, this constraint is added to an artificial supersource node. (We only ask you to add the constraint itself in Task 2.) By making every path through the BB begin with this supersource node, you enforce that every instruction must start after it.

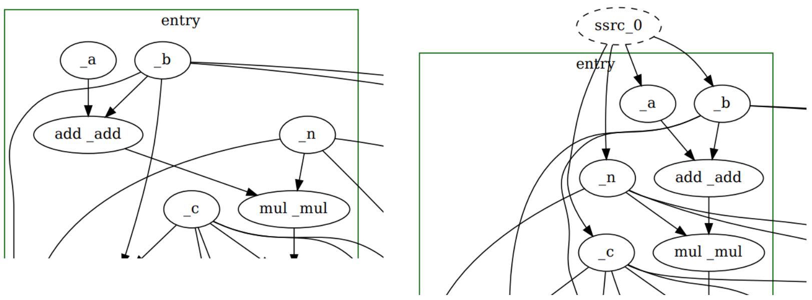
You should also add an artificial supersink node, which will provide you with the latency of the BB. By making every path through the BB end at the supersink node, you enforce that it must be the last instruction to execute. The execution time of the supersink node is therefore the BB's latency. The latency is important information for ALAP scheduling, which you will look at in Task 3.

Your first task is to implement the insertion of the supersource and supersink nodes for each BB.

Sections 9.2 to 9.4 walk you through each step of this process.

The member function **add_artificial_nodes** in the Scheduler class is responsible for adding these nodes. The function call and definition are already present, but you must add the body of the function. 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.

Figure 4: Kernel 1's BB0 after adding supersource



9.1 Hints

Here are some useful lines of code that will help in Sections 9.2 to 9.4.

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

```
#todo: somehow get the numericBBID

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

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 label attributes, which are stored in its member variable dictionary attr.

```
for bb in self.cfg:
    numericBBID = bb.attr["id"]
    bbLabel = bb.attr["label"]
```

"id" returns a numeric id of the BB, and "label" 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"])
```

9.2 Adding the nodes

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

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

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

9.3 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.

A reminder that 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).

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"
```

9.4 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)
```

9.5 Running and Testing

Once you have implemented the code, you can run the scheduler by calling

```
python run_sdc.py
```

In future tasks, you will be able to check if the resulting schedule is correct by calling

```
python tester.py
```

but this will not 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 code is already present in the file, along with a call to exit execution. The PDF will be saved in the directory you call the script from, which should be the top level directory.

Before you move on, you should make sure each BB has a supersource and supersink, and that they are connected correctly.

10 Task 2: ASAP Scheduling

As Soon As Possible (ASAP) scheduling does exactly what it says: every instruction is scheduled to execute as soon as it can.

In order to implement this, you need to do three things:

1. Constrain the supersource nodes to start at time 0.
2. Constrain each instruction to only execute after its predecessors have finished.
3. Minimize each instruction's execution time, so that they execute ASAP.

Each instruction is scheduled by our ILP solver, so you must add these constraints to the ILP formulation.

Sections 10.1 to 10.4 walk you through each step of implementing ASAP scheduling.

10.1 Add Every Node to the ILP Formulation

In order for a node to be scheduled, you must add it to the ILP formulation. This is to construct the list of instructions that need to be scheduled. You will add the constraints themselves later, in Section 10.2.

Please implement this in the body of **add_nodes_to_ilp**, which is a member function of the scheduler.

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}",var_type="i")
```

This line of code makes use of the fact that nodes return their name if used as a string. Again, please follow the scheduling variable naming convention.

Since we are only allowing integer start times, the var_type is ”i”, for integer.

The easiest way to constrain the starting time of the supersource is with the **add_variable** function itself. For the supersource nodes, please instead use:

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

The lower bound sets the lowest possible start time of a node.

10.2 Add the Data Dependency Constraints

For each BB, please encode all the data dependency constraints between the 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.

Encoding the dependencies should be done by in the body of the **set_data_dependency_constraints** member function in scheduler.py (the Scheduler).

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.

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.

A reminder that 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).

Section 9.3: Identifying which nodes to connect, has the code necessary to identify data dependencies between nodes.

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 has been instantiated as a member variable in the Scheduler for you.

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

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

We can insert the node variable itself into the f string, as it will auto-access the name of the node. The valid inequality signs are less than or equal, equal, and greater than or equal.

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(nodeA.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.

10.3 Call the Data Dependency Constraints Function

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

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

10.4 Set the Objective Function

You must also add every scheduling variable to the objective function, along with a coefficient. Variables with a positive coefficient will be minimized, and variables with a negative coefficient will be maximized.

ILP solvers can also solve weighted equations, where some variables more heavily effect the objective function and are prioritised more. Since we have no trade-offs to make, the value of the coefficient does not matter, only its sign.

Please alter the **set_asap_objective_function** in order to do this. 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}
```

10.5 Testing

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

10.6 Report Questions

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

11 Task 3: ALAP Scheduling

As Late As Possible (ALAP) scheduling does the opposite of ASAP scheduling: every instruction is scheduled in the last "allowed" cycle. "Allowed" means that the latency of the BB is not changed, e.g. the last instruction to execute happens in the same cycle as ASAP scheduling.

You will need to add an additional constraint to the supersink nodes to constrain the total latency, and then alter the objective function to instead maximise each instruction's execution time.

Sections 11.1 to 11.5 walk you through the process of implementing ALAP scheduling.

11.1 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. The supersink execution cycles are then used to constrain the latency of each BB.

11.2 Retrieve the Execution Cycle of the Supersinks

Please implement the **get_sink_svs** member function so that it returns a dictionary with the supersinks' names as keys, and their svs as values. This specific data-structure is requested as it is depended on by the tester.

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)
```

11.3 Add the Maximum Latency Constraints

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.

11.4 Create ALAP Schedule Constraints

Call both the **set_data_dependency_constraints** function and the **add_sink_sv_constraints** function in the **create_alap_scheduling_ilp** function to create the ALAP constraints.

11.5 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.

11.6 Testing

Call run_sdc and the tester, and view the Gantt chart. Make sure your code passes the tests and that the nodes are scheduled correctly before moving on.

11.7 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. Comment on the area-performance trade off offered by ASAP vs. ALAP. First discuss the general case, and then contrast the ASAP solution to the ALAP solution for each of the provided examples.

12 Task 4: ASAP with Resource Constraints

ASAP with resource constraints builds on your existing ASAP scheduling implementation. You must now also add resource constraints to certain operations, which will alter the final schedule. To allow parameterization of the constraints, the maximum allowed resource usages are passed as a dictionary to the functions you must implement.

Sections 12.2 and 12.3 will walk you through implementing the required functions.

12.1 Hints

You will need to alter the **add_resource_constraints** member function in resource.py

The parameter

```
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"])
```

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

12.2 Sort the Operation Nodes

To constrain resource usage requires an order of the the constrained instructions, e.g. you need to first order all of the add operations.

The constraints themselves depend on the number of resources available:

1. With a single unit, the second operation must start after the first, the third operation must start after the second, and so on.
2. With two units, the third operation must start after the first, the fourth operation must start after the second, and so on.

This pattern continues with more units.

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 ordering is correct, but both overly restrictive and random: it guarantees that an instruction appears after all instructions it is dependent on. However, instructions without any dependencies may appear at any point in the ordering. 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.

12.3 Constrain the Operation Nodes

A sample constraint for when operation B must start after operation A is:

```
sv_B - sv_A >= 1
```

Use this type of constraint to enforce the maximum number of each type of unit specified by the dictionary.

Make sure to only add constraints between instructions in the same BB. It might be easiest to loop over the topological ordering separately for each BB, or you could try track them all simultaneously.

12.4 Testing

Call run_sdc and the tester for resource constraining, and look at the Gantt charts. Make sure your code passes the tests and that your resource constraints are respected before moving on.

12.5 Report Questions

1. Vary the resource constraints to generate several different schedules per kernel (to a maximum of 3 schedules per kernel). Provide visualizations of each schedule, and compare area and latency with the results of Task 1.
2. For each kernel, specify a set of resource constraints which produce the same schedule as from Task 1. Explain why the schedule is unchanged. If this is not possible, explain why not.

13 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 specifying 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 need to be added to the ILP formulation and instead can be hard-coded into the right hand side of the constraints. To repeat for emphasis: the II is not added to the list of variables that the ILP solver can change to find a solution, and so is not added to the formulation. Instead, the constant on the right-hand side of each individual constraint is dependant on the II, and the ILP will fail to find a valid solution if the II is too low.

Sections 13.1 to 13.3 will walk you through each step of implementing pipelined scheduling.

13.1 Add the Inter-Iteration Data Dependency Constraints

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"
```

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.

13.2 Call the Pipelining Schedule Constraints Function

Call the 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.

13.3 Find Loop BB

For different kernels, the BB to be pipelined has a different ID. The **find_loop_bb** function is used to identify which BB to pipeline. The call is already present, but you must add the function body. In this project, the BB being pipelined is always the BB with the longest latency.

13.4 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.

13.5 Report Questions

1. For kernel 3 and kernel 4, specify a formula to calculate the kernel latency. The formula should be a function of N, the number of loop iterations.
2. Comment on the area-performance trade-off that pipelining offers us. Discuss in the general case, and on the provided example kernels.
3. What are the achievable IIs?

14 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, we will use a Modulo Reservation Table (MRT).

Sections 14.1 and 14.2 describe how to build and use an MRT. The use case is given first, and then the specifics of constructing the table.

Section 14.3 walks you through the specific steps you will need to follow in order to check inter-iteration resource constraints using an MRT.

14.1 Background Information: Using an MRT

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 rules result in a valid MRT, then the candidate schedule respects the resource constraints.

The first step of the two-step process is to generate a schedule. The candidate schedule is created as normal, which for this project is through ILP. Both intra-iteration and inter-iteration data dependency constraints are added to the formulation. Resource constraints are also added, but only for intra-iteration. To repeat for emphasis: The schedule generation step should consider resource usage collisions inside an iteration, but not between multiple iterations.

The second step of the two-step process is to use the MRT to observe the candidate schedule's resource usage, and validate the schedule. If the desired resource constraints are not respected, a new schedule must be made.

The simplest way to do this is to repeat the first step and generate a new schedule, this time with a larger II. More complex methods can alter the execution cycle of the instructions to reduce collisions, but we will not consider them in this project.

14.2 Background Information: Constructing an MRT

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

Unconstrained operations are ignored, as they cannot effect schedule validity.

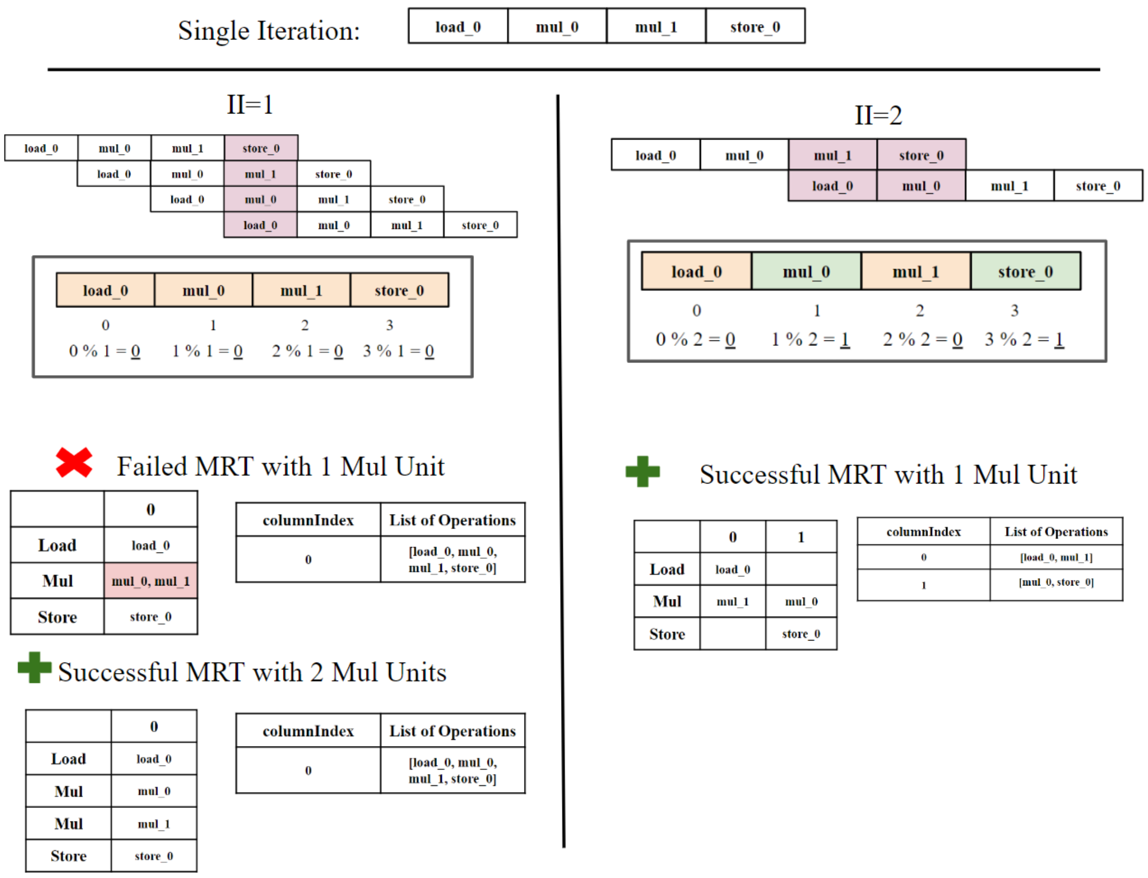
MRT construction has the following rules:

- All constrained operations must be placed in the MRT.
- Each cell can contain only one operation.
- An operation can only be inserted in a row that can execute it e.g. an add row for an add operation.
- The column is chosen by the formula: columnIndex = sv_node % II

Hence, the name "Modulo Reservation Table". If any of the 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



14.3 Find a Valid Schedule using a Modulo Reservation Table

The **check_resource_constraints_pipelined** member function is responsible for validating inter-iteration resource constraints. The definition and function call are provided, but you must implement the body of the function. Please implement MRT construction in order to check the validness of the candidate schedule. It should return true if the MRT construction succeeds, and false if it fails.

We recommend to construct the MRT using a dictionary of lists, but you can could also create a custom data-structure.

If using a dictionary of lists: For each node in the graph calculate its columnIndex using the MRT formula. Then, using the columnIndex as the key, add the node's type to the list. Once finished, check if the list at each columnIndex has more than the maximum number of each constrained resource type. If it does, the check has failed, and you should return false.

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)
```

14.4 Alter Node Ordering

Now that you have resource constrained pipelining, you will see the default resource constraints in run_SDC (1 of each resource) cause the II of kernel 3 to increase significantly. Investigate why.

Try to fix the problem by changing the order of operations used for creating resource constraints. There are many ways to generate a topological ordering of a graph, so there is no single correct solution to this.

14.5 Testing

Call run_sdc and the tester to evaluate your code. Make sure the inter-iteration resource constraints are respected before moving on.

14.6 Report Questions

1. Does the MRT heuristic compute the optimal scheduling? Explain why or why not.
2. One input to the MRT is both non-deterministic and highly influential. Identify the input and and discuss the impact of this non-deterministic behavior.
3. Describe any experimental change you made to the MRT. Visualize at least 2 MRT generated schedules for kernel 3, and discuss why they are different.

15 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.