Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

DYNAMO

SS 2023

Prof. Dr. Lana Josipović, Carmine Rizzi, and Jiahui Xu

Synthesis of Digital Circuits

Project 1: Implement Your Own Modulo Scheduler

1 Background

This is the description of the first project of the course 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 recreate a modulo scheduler able to handle complex design requirements.

Besides the submission for your implementation of the scheduler, please provide ***a pdf report*** for answering the additional questions in the assignments.

The following document is divided in 5 sections:

2 Scope of the Project: it provides a general view of the project

3 Implementation Notes: it contains information and assumptions of the code

4 Intermediate Representation Format: it describes the Intermediate Representation (IR)

5 How to Use the Template Project: it depicts the template project and it provides a "dummy" example

6 Your Tasks: it contains the tasks of this project

2 Scope of the Project

Starting from an optimized intermediate representation (IR), we would like to investigate the following tasks:

- 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 a heuristic procedure for resource-constrained scheduling;
- Implementing an incremental method for pipelined scheduling.
- Implementing a heuristic procedure for resource-constrained pipelined scheduling.

Figure 1 illustrates a high-level overview of our scheduler. Our scheduler receives the following inputs:

- **LLVM IR**: optimized intermediate representation. We provide the LLVM IRs corresponding to several C++ code examples. Note: the C++ code should not be used in the Scheduler.

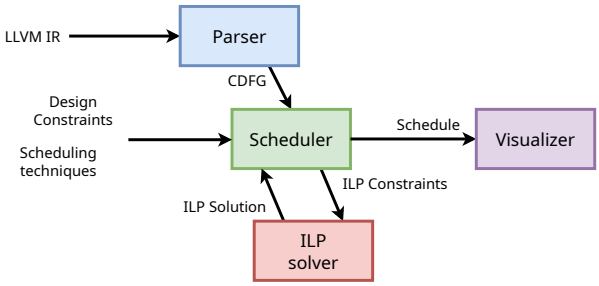

Figure 1: Overview of our scheduler.

- **Design Constraints**: user-defined constraints to limit the number of available resources
- **Scheduling Technique**: different scheduling technique that can be applied by the scheduler

The output of the scheduler is a Gantt chart representing the scheduling of the IR input. The representation used by the scheduler is a *CDFG* (Control DataFlow Graph). It is an equivalent format to the LLVM IR and can be rendered directly by graphviz-dot for quick visualization.

The scheduler consists of the following parts:

- **Parser**: it converts LLVM IR to CDFG graphical representation.
- **Scheduler (your turn to complete)**:

  (1) it analyzes the CDFG;

  (2) it generates scheduling constraints;

  (3) it interacts with the ILP solver;
- **ILP Solver**: it solves the scheduling constraints problem and it returns the scheduling solution to the Scheduler.
- **Visualizer**: it is a visualizer for analyzing the scheduling results.

In the "examples" folder, there are 4 kernels which can be used to test your code. The questions for each task are related to the scheduling of these kernels.

NOTE: in each task you should **NOT** use all the kernels. The applicable kernel depends on the task to solve.

3 Implementation Notes

In this section, we address some important *conventions* that you should follow when completing the assigments. These conventions should be taken into account for all tasks.

3.1 Common Assuptions for All Tasks

We assume the following for each *basic block (BB)* in the CDFG:

- 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 and can happen out of order).
- We assume that the critical paths of the components are the same, and all multi-cycle operations are pipelined (with $II = 1$).

We assume the following for scheduling a *control flow graph (CFG)*:

- Assume that the 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).
- For all tasks, the solution should be reported as the start cycle of each operation (with respect to the beginning of its BB).
- *Constant nodes* are considered as immediate operations (i.e. their combination delay is 0).

3.2 ILP Problem Formulation

When formulating ILP problems for CDFG scheduling, *formally* we 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 variable $sv_i(v)$.

Please follow the notation/naming above when you construct the ILP formulation.

Few additional remarks:

a) 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.

b) For each node in a BB, if it has no predecessors from the same BB, then this node is connected to the super-source; if it has no successors from the same BB, then this node is connected to the super-sink.

c) 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 removed when adding artificial nodes (i.e., *supersources* and *supersinks*).

3.3 Resource Constraints

Accelerator designs are usually subject to many constraints. In this project, we consider only the resource constraints. For resource constraints, we consider that our platform is constrainted by the number of functional units of each resource type. For instance, for a requirement that the number of multipliers and the number of adders will not exceed 3 and 4, respectively:

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

Figure 2: An example of resource constraints represented as a Python dictionary.

4 Intermediate Representation Format

The scheduler is supposed to operate on the CDFG representation of the LLVM IR code. It is **NOT** supposed to read the IR code. An example of a CDFG of an HLS kernel described in Figure 3 is illustrated in Figure 4.

The CDFG is given in Graphviz-dot format (graphviz.org).

```
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: An example of HLS kernel.

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

- **Operator nodes**: they directly correspond to 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 LLVM memory operations load and store. In this project, we do not distringuish memory nodes from operator nodes ("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. The insertion of these auxiliary nodes is done in the parser utility. 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. And they are used for encoding the following information.

- **Solid-lines**: Direct data dependencies.
- **Dashed-lines**: Backward edges represents data dependencies use in BBs that are revisited.

Additionally, the parser also generates **control flow graph (CFG)**.

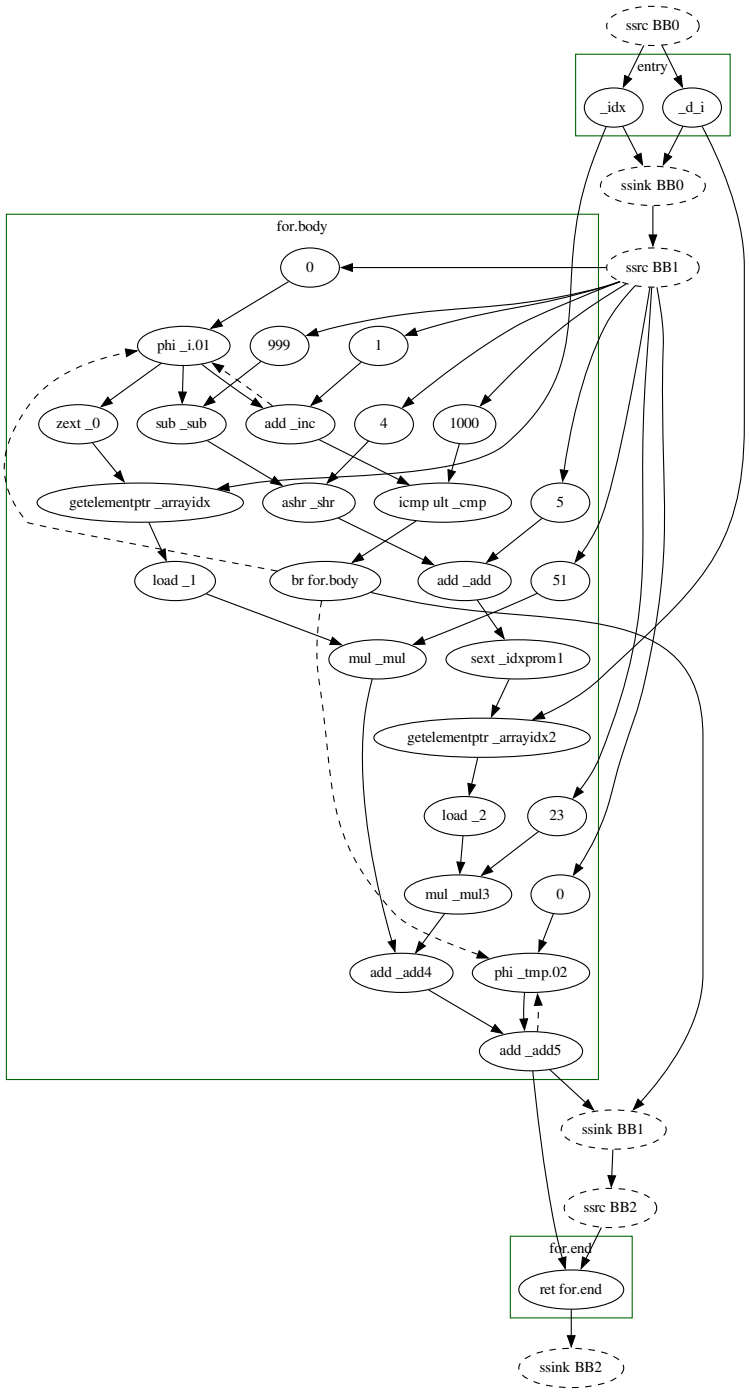

Figure 4: An example of CDFG representation of Figure 3.

5 How to Use the Template Project

We provide a *template project* for you to build your scheduler. Assume that the base folder for the project is called `sdc-sdc/`. This template project also includes necessary utilities, such as:

- A LLVM IR to CDFG converter (`sdc-sdc/src/main_flow/parser.py`).
- A library for graph operations (`sdc-sdc/src/utilities/cdfg_manager.py`).
- A library for interacting with the ILP solver (`sdc-sdc/src/utilities/ilp_manager.py`).
- Miscellaneous utilities (e.g., regex) (`sdc-sdc/src/utilities/regex.py`).

Please *do not change the files mentioned above*, and *do not import any other modules*. These utilities include *detailed descriptions of their functionalities* which you can find in the documentations in these files. Apart from these utilities, there are several incomplete files for you to complete:

- Top-level script for running your scheduler (`sdc-sdc/run_SDC.py`).
- Your implementation of the scheduler (`sdc-sdc/src/main_flow/scheduler.py`).
- Your implementation of the resource manager (`sdc-sdc/src/main_flow/resource.py`).
- List of examples to run (`sdc-sdc/filelist.lst`).

In this section, we are going to walk you through the implementation of a very basic scheduler: it creates a legal schedule (i.e., all dependencies are respected), but it does not exploit any parallelism.

5.1 An API for CDFG IR generation.

To get the CDFG, we provide you with a library for converting *LLVM-IR* to *CDFG-IR* (the library file is located in `sdc-sdc/src/main_flow/parser.py`.). To demonstrate how it works, we start by editing `sdc-sdc/filelist.lst`:

```
- dummy
+ kernel_1
```

We implement the basic scheduler by completing the code in `sdc-sdc/src/main_flow/scheduler.py`. We first need to add a new scheduling technique (for the following, we refer this basic technique as *"naive"*):

```
- scheduling_techniques = ["asap", "alap", "pipelined"]
+ scheduling_techniques = ["asap", "alap", "pipelined", "naive"]
```

The top-level script (`sdc-sdc/run_SDC.py`) can generate the CDFG representation from LLVM-IR. To generate the CDFG for the `kernel_1` kernel, simply execute this script in your prompt:

```
$ python3 ./run_SDC.py
```

Now you should be able to see the PDF file located in `sdc-sdc/examples/kernel_1/parser_output.pdf`.

To use the scheduler, you need to generate an object of the class "Scheduler". Please add the following lines in your `sdc-sdc/run_SDC.py` script:

```
                ssa_parser.draw_cdfg("{0}/{1}/parser_output.pdf".format(base_path,
    example_name))

+               ######################## Naive ######################
+               scheduling_type = "naive"
+               scheduler = Scheduler(ssa_parser, scheduling_type, log=log)
+
                ######################## ASAP ########################
                # TODO: write your code here
```

5.2 A Library for Graph Operations

In this section, we describe the graph utilities library. These utilities are used to retrieve information from the CDFG and *add/remove* nodes from the CDFG. We can add a few options in the initializer function inside the scheduler class (in sdc-sdc/src/main_flow/scheduler.py).

```
    # set solver options
    self.ilp = ILP(log=log)
    self.constraints = Constraint_Set(self.ilp, log=log)
    self.opt_fun = Opt_Function(self.ilp, log=log)

+   # This is how you retrieve information
+   for operation_node in get_cdfg_nodes(self.cdfg):
+     print(f'Name of the node: {operation_node}')
+     print(f'Type: {operation_node.attr["type"]}')
+     print(f'Latency: {get_node_latency(operation_node.attr)}')

    # define ilp variable per each node
    # TODO: write your code here
    pass
```

To use the scheduler, it must be initialized in sdc-sdc/run_SDC.py

```
                    continue
                ssa_parser.draw_cdfg("{0}/{1}/parser_output.pdf".format(base_path,
    example_name))

+               ###################### Naive ####################
+               scheduling_type = "naive"
+               scheduler = Scheduler(ssa_parser, scheduling_type, log=log)
```

In this way, we can print out the *Name*, *Type*, and *Latency* attributes of the CDFG nodes (recall that dashed edges are back edges). Then we re-run sdc-sdc/run_SDC.py:

```
$ python3 ./run_SDC.py
Name of the node _add
Type: add
Latency: 1

Name of the node _b
Type: argument
Latency: 1

Name of the node _a
Type: argument
Latency: 1
...
```

Similarily, we can print out the *predecessor*, *successor*, and *type* of the CDFG edges. Again, we directly write the changes to sdc-sdc/src/main_flow/scheduler.py.

```
                    print(f'Type: {operation_node.attr["type"]}')
                    print(f'Latency: {get_node_latency(operation_node.attr)}\n')

+               # loop through the entire graph, and access attributes of edges
+               for cdfg_edge in get_cdfg_edges(self.cdfg):
+                       print(f'For Edge: {cdfg_edge[0]} -> {cdfg_edge[1]}')
+                       if cdfg_edge.attr["style"] == "dashed":
+                               print('Edge type: back edge.\n')
+                       else:
+                               print('Edge type: normal edge.\n')
+
                # define ilp variable per each node
                # TODO: write your code here
```

Then we re-run sdc-sdc/run_SDC.py:

```
$ python3 ./run_SDC.py
...
For Edge: _add -> _mul
Edge type: None

For Edge: _b -> _add
Edge type: None

For Edge: _b -> _sub
```

```
Edge type: None
...
```

5.3 A Library for Interacting with the ILP Solver

We provide you a library for interacting with the *ILP solver*. The library file is located in sdc-sdc/src/utilities/ilp_manager.py (again, you do not have to change the content of this file).

You can create an ILP model by interacting with an `ILP` *object*, a `Constraint_Set` *object*, and an `Opt_Function` *object*. In the following, we refer to the collection of these three objects as an *ILP tuple*. These objects are already defined in the *initializer function* of the scheduler class in sdc-sdc/src/main_flow/scheduler.py.

```
    # set solver options
    self.ilp = ILP(log=log)
    self.constraints = Constraint_Set(self.ilp, log=log)
    self.opt_fun = Opt_Function(self.ilp, log=log)
```

The functionality of ILP tuple is as follows:

- `self.ilp`: it is used to interact with the ILP formulation
- `self.constraints`: it is used to describe the constraints of the ILP formulation
- `self.opt_fun`: it is used to describe the optimization function of the ILP formulation.

We need more context interacting with the ILP solver library, which is described in the following subsection.

5.4 A Dummy Scheduler

In this tutorial, we guide you through the implementation of a very basic scheduler: it schedules all nodes one after another, i.e., it serializes all the operations, no spatial parallelsm is exploited.

The resulting schedule is reported as a list of *scheduling variables(sv)*, one per each node in the CDFG. We need to create one ILP variable per each CDFG node. We start by changing the initializer function in the scheduler class inside sdc-sdc/src/main_flow/scheduler.py.

```
                # define ilp variable per each node
-               # TODO: write your code here
+               for n in self.cdfg:
+                       self.ilp.add_variable(f'sv{n}', lower_bound = 0,  var_type="i")
                pass

        # function to set scheduling technique
```

Note: the scheduling variable of each CDFG node should be prefixed with *sv* (this is your first task). This **naming convention has to be respected**, since the schedule visualization tool (`print_gantt_chart`, which will be described later) uses this naming convention to know which ILP variables represent the scheduled time of different operations.

We start the scheduling by creating a basic topological order of the nodes in the `create_scheduling_ilp` function of the sdc-sdc/src/main_flow/scheduler.py:

```
                elif self.sched_tech == "pipelined":
                        # TODO: write your code here
                        pass
+               # This is just a dummy scheduling for your reference,
+               # you can delete this part afterwards
+               elif self.sched_tech == "naive":
+                       # We first obtain a Topological sort
+                       ordering = get_topological_order(self.cdfg)
+                       print(ordering)
+
                self.set_opt_function()
```

Note: By scheduling one CDFG node after the other using a topological order, we guarantee that all data dependencies of the program are respected.

We can add ILP constraints that enforce an order between adjacent nodes in the topologically ordered list. For a pair of adjacent nodes n, v we enforce a relation on the scheduling variables of these two CDFG nodes:

$$sv(n) + Lv(n) \leq sv(v), \tag{1}$$

which means, for every operation in the a CDFG, the *start time of this node* must be greater than or equal to the *end time* of the previous node in this topological ordered list.

Recall that, in Section 3, we have mentioned that we assume that the execution of different BBs are completely sequentialized. Therefore, when we create ILP constraints, we consider the topological order obtained per each BB. This can be done by extracting the order of operations in individual BBs from the full order.

For adding the constraints to sequentialize BBs, we continue to modify the `create_scheduling_ilp` function of the sdc-sdc/src/main_flow/scheduler.py:

```
        # This is just a dummy scheduling for your reference, you can delete this part
    afterwards
        elif self.sched_tech == "naive":
            # 1. We first obtain a Topological sort.
            ordering = get_topological_order(self.cdfg)
+           # 2. We assume that different BBs are completely disjoint.
+           #    Therefore we perform naive scheduling with respect to individual BBs
+           list_of_bbs = set(get_cdfg_nodes(self.cfg))
+           for bb in list_of_bbs:
+               # filter the topological ordering within a BB
+               ordering_in_bb = [ n for n in ordering if n.attr['bbID'] == bb ]
+               # For every two adjacent nodes in the topological order of this BB
+               for src, dst in zip(ordering_in_bb[0:-1], ordering_in_bb[1:]):
+                   src, dst = self.cdfg.get_node(src), self.cdfg.get_node(dst)
+                   # The later node cannot start before the earlier node finishes.
+                   # We create an ILP constraint every two adjacent nodes.
+                   self.constraints.add_constraint({f'sv{src}' : -1, f'sv{dst}' : 1},
+                       "geq", get_node_latency(src.attr) )
        self.set_opt_function()
```

Here we would like to highlight the syntax for adding constraints to the ILP problem. Defining an ILP constraint requires the following arguments in the class method `self.constraints.add_constraint(dict_of_variables, type_of_inequality, constant)`:

a) Dict of variables: each key-value pair is an ILP variable and its constant coefficient

b) Type of inequality: a string in either "eq" ($=$), "leq" (le), or "geq" (ge)

c) Constant: each ILP constraint can have up to 1 constant term (defaults to 0)

An ILP problem also needs an *objective function* (or *optimization function*), we can simply minimize the sum of the start time of the last node of each BB in topological order. For this, we would like to modify the `set_opt_function` function in sdc-sdc/src/main_flow/scheduler.py:

```
        elif self.sched_tech == 'naive':
                # we minimize the last node in the topological order
                ordering = get_topological_order(self.cdfg)
-               pass
+               list_of_bbs = set(get_cdfg_nodes(self.cfg))
+               for bb in list_of_bbs:
+                       # filter the topological ordering within a BB
+                       ordering_in_bb = [ n for n in ordering if n.attr['bbID'] == bb ]
+                       self.opt_fun.add_variable(f'sv{ordering_in_bb[-1]}', 1)
        else:
                self.log.error(f'Not implemented option! {self.sched_tech}')
                raise NotImplementedError
```

Now to test if your basic scheduler works as intended, change the sdc-sdc/run_SDC.py script. `create_scheduling_ilp` creates the ILP constraints and objective function according to your implementation. `solve_scheduling_ilp` calls the ILP solver. We provide a class method `print_`

`gantt_chart` for generating a time-operation diagram for debugging purposes. We also provide a class method `print_scheduling_summary` for generating a textual report.

```
        ###################### Naive ######################
        scheduling_type = "naive"
        scheduler = Scheduler(ssa_parser, scheduling_type, log=log)
+       scheduler.create_scheduling_ilp()
+       status = scheduler.solve_scheduling_ilp(base_path, example_name)
+       chart_title = "{0} - {1}".format("Naive", example_name)
+       scheduler.print_gantt_chart(chart_title,
+           "{0}/{1}/{2}_{1}.pdf".format(base_path, example_name, scheduling_type))
+       scheduler.print_scheduling_summary(
+           "{0}/{1}/{2}_{1}.txt".format(base_path, example_name, scheduling_type) )

        ###################### ASAP ######################
        # TODO: write your code here
```

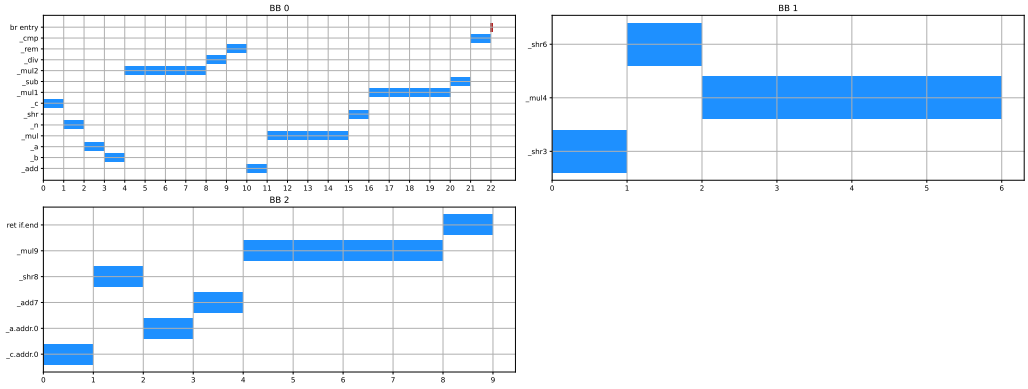

Figure 5: Resulting schedule produced from the dummy scheduling.

Figure 5 illustrates the visualized schedule (sdc-sdc/examples/kernel_1/naive_kernel_1.pdf) produced by our dummy scheduler. For instance, "*_mul9*" is scheduled in cycles 4 to 7 of BB2 scheduling since it takes 4 clock cycles to be executed. On the other hand, "*_add7*" takes only one clock cycle to be executed (clock 3). In this example, adjacent nodes will never be scheduled in the same cycle, unless the prior node in topological order has zero latency. You may view the scheduling summary of the produced schedule (sdc-sdc/examples/kernel_1/naive_kernel_1.txt):

```
sv(_c) @ bb(0) := 0.0
sv(_n) @ bb(0) := 1.0
sv(_add) @ bb(0) := 10.0
sv(_mul) @ bb(0) := 11.0
sv(_shr) @ bb(0) := 15.0
sv(_mul1) @ bb(0) := 16.0
sv(_a) @ bb(0) := 2.0
sv(_sub) @ bb(0) := 20.0
sv(_cmp) @ bb(0) := 21.0
sv(br entry) @ bb(0) := 22.0
sv(_b) @ bb(0) := 3.0
sv(_mul2) @ bb(0) := 4.0
sv(_div) @ bb(0) := 8.0
sv(_rem) @ bb(0) := 9.0
sv(_shr3) @ bb(1) := 0.0
sv(_shr6) @ bb(1) := 1.0
sv(_mul4) @ bb(1) := 2.0
sv(_c.addr.0) @ bb(2) := 0.0
sv(_shr8) @ bb(2) := 1.0
sv(_a.addr.0) @ bb(2) := 2.0
sv(_add7) @ bb(2) := 3.0
sv(_mul9) @ bb(2) := 4.0
sv(ret if.end) @ bb(2) := 8.0
II := N/A
```

6 Your Tasks

Deliverables

When you finish the project, you have to submit to us the entire completed `sdc-sdc/`project, together with your report (please make sure that you have answered all questions described in **"Questions to Answer for the Report"** section in each Task).

ONLY the following python scripts should be changed:

- `sdc-sdc/run_SDC.py`
- `sdc-sdc/src/main_flow/scheduler.py`
- `sdc-sdc/src/main_flow/resource.py`

Besides the implementations for the scheduler class, resource class, and top-level script, you should keep the generated Gantt diagrams and textual scheduling summaries in their respect example directories (these files are **NECESSARY** for submission, the way of generating these files is described in Subsection 5.4, using class methods `print_gantt_chart` and `print_scheduling_summary` respectively). For instance:

- `sdc-sdc/examples/kernel_(1|2|3|4)/(asap|alap|pipelined)_kernel_(1|2|3|4).pdf`: Gantt diagrams.
- `sdc-sdc/examples/kernel_(1|2|3|4)/(asap|alap|pipelined)_kernel_(1|2|3|4).txt`: Scheduling summaries.

Additionally, for kernels that are scheduled with resource constraints (the way of specifying the resource constraints is introduced in Subsection 3.3), please also include the assumed resource constraints in the file names. For instance, if you assume resource constraints of *3 adders* and *2 multipliers* for *resource constrained ASAP scheduling* of kernel 1, you need to specify this information in the file names as follows:

- `sdc-sdc/examples/kernel_1/asap_kernel_1_resource_add_3_mul_2.pdf`: Gantt diagrams.
- `sdc-sdc/examples/kernel_1/asap_kernel_1_resource_add_3_mul_2.txt`: Scheduling summaries.

Task 0: Inserting Supernodes

Please complete the `add_artificial_nodes` function in the Scheduler class in sdc-sdc/src/main_flow/scheduler.py. This function should insert the supersource and supersink nodes into a given CDFG. Follow the description in Section 3 to complete this task.

Hint: To add a node in the CDFG, you can use the following syntax:

```
self.cdfg.add_node(node_name, id=bb_id, bbID=bb_label, type=node_type, label=node_name)
```

- `node_name`: the *name* of the node, which can be `ssink_bb0`.
- `bb_id`: the *numeric id* of the BB of the node (e.g., it will be `0` for the super sink node in BB 0, this information is available in the CFG).
- `bb_label`: the *textual label* of the BB of the node (this information is available in the CFG, by accessing the attribute a CFG node using: `bb.attr["bbID"]`. E.g., it could be called `for.body` for a BB that is the loop body).
- `node_type`: the type of the node (i.e., in this case, **it can only be** either `supersink` or `supersource`).
- `label`: the *label* of the node, you can set it to be the same as `node_name`.

After adding these supernodes into the CDFG, you also need to appropriately connect these supernodes with existing nodes, following the rules described in Section 3. To add an edge in the CDFG, you can use the following syntax:

```
self.cdfg.add_edge(predecessor, successor)
```

Task 1: ASAP Scheduling

For the given CDFG, create a scheduler that schedules each BB using ASAP scheduling.

You can implement this scheduling algorithm by completing the following "TODOs" in `sdc-sdc/src/main_flow/scheduler.py`:

a) Complete `set_data_dependency_constraints` to generate constraints from the data dependencies edges.

b) Complete `create_scheduling_ilp` to call `set_data_dependency_constraints` for the ASAP scheduling algorithm.

c) Complete `set_opt_function` to generate the objective function for the ASAP scheduling algorithm.

Additional remarks

a) This should be solved per each BB individually.

b) This is typically not solved using an ILP formulation.

Questions to Answer for the Report

- Is the obtained latency always minimal?
- Why should the scheduling problem be solved per each BB individually?

Task 2: ALAP Scheduling

For the given CDFG, create a scheduler that schedules each BB using ALAP scheduling.

You can implement this scheduling algorithm by completing the following "TODOs" in `sdc-sdc/src/main_flow/scheduler.py`:

a) Complete `set_data_dependency_constraints` to generate constraints from the data dependencies edges.

b) Complete `create_scheduling_ilp` to call `set_data_dependency_constraints` for ALAP scheduling algorithm.

c) Complete `set_opt_function` to generate the objective function for ALAP scheduling algorithm.

Additional remarks

a) ALAP scheduling needs time *bounds* to limit its scheduling execution. One possibility for obtaining the *bound* is to run ASAP for the given BB, and use $sv(ssink)$, where *ssink* is the super-sink of the BB.

Questions to Answer for the Report

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

Task 3: ASAP Scheduling Under Resource Constraints

For the given CDFG, create a scheduler that minimizes the latency of each BB while honoring the provided resource constraints. Start from the scheduling technique completed in Task 1 and include the resource constraints. The order that the resource constrained operations should follow is the topological order. This order should be enforced to limit the number of parallel operations starting in the same clock.

You can implement this scheduling algorithm by completing the following "TODOs" in `sdc-sdc/src/main_flow/resource.py`:

a) Complete `add_resource_constraints` to generate resource constraints.

Additional remarks

a) This should be solved per each BB individually.

b) The resources constraints are saved inside the dictionary `resource_dic` where the keys correspond to resource types and the value is the maximum allowed number of resources.

Question to Answer for the Report

- Vary the "add", "mul" and "zext" resource contraints from 1 to 3 (try all possible constraint combinations). Highlight interesting cases and compare area and latency with the results of Task 1. How resource constraints affect the scheduling?
- Which are the resources that influence the scheduling and with which maximum number of allowed operations?

Task 4: Pipelined Scheduling

Assume a loop with no internal control flow containing only a single BB. Minimize latency for the best possible II. Assume unlimited resources. Apply an iterative approach that increases the II in a stepwise manner until a valid solution is found.

You can implement this scheduling algorithm by completing the following "TODOs" in `sdc-sdc/src/main_flow/scheduler.py`:

a) Complete `set_opt_function` to generate the objective function for the pipelined scheduling.

b) Complete `set_II_constraints` to generate constraints to verify the feasability of a specific II value.

Additional remarks

a) The examples that should be analysed should contain loops (i.e., you only need to test your pipelined scheduler on *examples with loops*).

b) For examples with loops, you may assume that the BB with longest latency is the loop body (however, you still need to make sure that the *loop BB* identified in this way is consistent with the source code and LLVM IR).

c) You should only consider data loop-carried dependencies (i.e., the edge from *br* to *phi* is **NOT** a data dependency)

Question to Answer for the Report

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

Task 5: Pipelined Scheduling Under Resource Constraints

Starting from the scheduling generated from the previous task, implement the following heuristic to include resource constraints.

Task 5.1 : MRT Class

The proposed heuristic uses a *MRT* (Modulo Reservation Table). This table contains the execution clock of each operation across multiple overlapping iterations given a scheduling solution. Then, it evaluates the correctness of this scheduling with respect to a given resource constraint considering all overlapping interations.

You can implement MRT class by completing the following "TODOs" in `sdc-sdc/src/main_flow/resource.py`:

a) Complete `generate` in the `MRT` class to create the table containing the execution clock of each operation across multiple iterations.

b) Complete `is_legal` to check if the scheduling output respects the desired maximum number of instances (the resource constraint).

The `generate` function should iterate through the scheduling of the first iteration. For each clock, it should retrieve the operations taking place in the that cycle assuming an initiation interval $II = x$. The output of this function should be a dictionary where the keys correspond to the clocks and the values correspond to list of operations taking place in that clock. For instance, the output of the generate function for BB1 in Figure 5 with an $II = 3$ is $\{0 : \text{“}\_shr3\text{”}, 1 : \text{“}\_shr6\text{”}, 2 : \text{“}\_mul4\text{”}, 3 : \text{“}\_shr3\text{”}, 4 : \text{“}\_shr6\text{”}, 5 : \text{“}\_mul4\text{”}\}$. Meanwhile, for $II = 2$ the output is $\{0 : \text{“}\_shr3\text{”}, 1 : \text{“}\_shr6\text{”}, 2 : [\text{“}\_mul4\text{”}, \text{“}\_shr3\text{”}], 3 : [\text{“}\_shr6\text{”}], 4 : [\text{“}\_mul4\text{”}, \text{“}\_shr3\text{”}], 5 : [\text{“}\_sh6\text{”}]\}$.

The `is_legal` function returns $True$ if MRT respects all resource constraints in a specific cycle and $False$ otherwise.

Task 5.2 : Heuristic Algorithm

You can implement this scheduling algorithm by completing the following "TODOs" in `sdc-sdc/src/main_flow/resource.py`:

a) Complete `add_resource_constraints_pipelined` to add resource constraints if the solution is feasible. If not, it returns $Fail$

The order that the resource constrained operations should follow is the **topological order**. This order should be enforced to limit the number of parallel operations starting in the same clock. The heuristic algorithm uses the following variables:

- MRT : MRT object containing execution cycle of each operation according to ilp solution
- operations_solved : list of operations for which cycle has been decided
- constraint_list : list of new constraints added and to remove in case of failure
- schedQueue : initial queue containing all resource constrained operations
- budget : maximum allowed number of iterations
- ILP : ILP object which contains sdcSolution
- resource_constraint : resource constraint to respect

The pseudo-code of the heuristic algorithm is the following:

```
1: MRT ← MRT(ILP)
2: operations_solved ← [ ]
3: constraint_list ← [ ]
4: while schedQueue not empty and budget ≥ 0 do
5:     operation ← pop schedQueue                          ▷ Pop resource critical operation
6:     sv_operation ← ILP.get_operation_timing_solution(operation)   ▷ Retrieve the starting time of the operation
7:     if MRT.is_legal(operation, sv_operation, operations_solved, resource_constraint) then
       ▷ Check if the operation and its execution time are valid according to the resource constraint considering the list of solved operations
8:         constraint_id ← constraints.add(operation_time = sv_operation)   ▷ If the operation is legal, the operation is executed at a precise cycle
9:         constraint_list.append(constraint_id)
10:        operations_solved.append(operation)
11:    else
12:        constraint_id ← constraints.add(operation_time ≥ sv_operation + 1)   ▷ If the operation cannot be executed in this clock, it should be at least next one
13:        constraint_list.append(constraint_id)
14:        status ← ILP.solve(constraints)   ▷ Compute new solution due to new constraint
15:        if status is feasible then                      ▷ Check if the problem is feasible
16:            schedQueue ← push schedQueue operation
17:            MRT.generate(ILP)                ▷ Generate a new table for new sched. solution
18:        else
19:            ILP.remove_constraints(constraint_list)
20:            return Fail          ▷ If the problem is unfeasible, the heuristic returns "Fail"
21:        end if
22:    end if
23:    budget ← budget − 1                                 ▷ Each iteration decreases the budget
24: end while
25: if schedQueue not empty then
26:    return Fail
27: else
28:    return Success
29: end if
```

Additional remarks

The two **additional remarks** of **Task 4** also apply here. **Think twice** when you implement the *push* and *pop* functions.

Question to Answer for the Report

- Does this heuristic computes the optimal scheduling? Motivate your answer.
- Which feature of this heuristic has the most significant effect on the scheduling?
- How could this heuristic be improved?