A Cycle-by-Cycle Scheduler for a VLIW Processor

GAO Xufeng

May 15, 2022

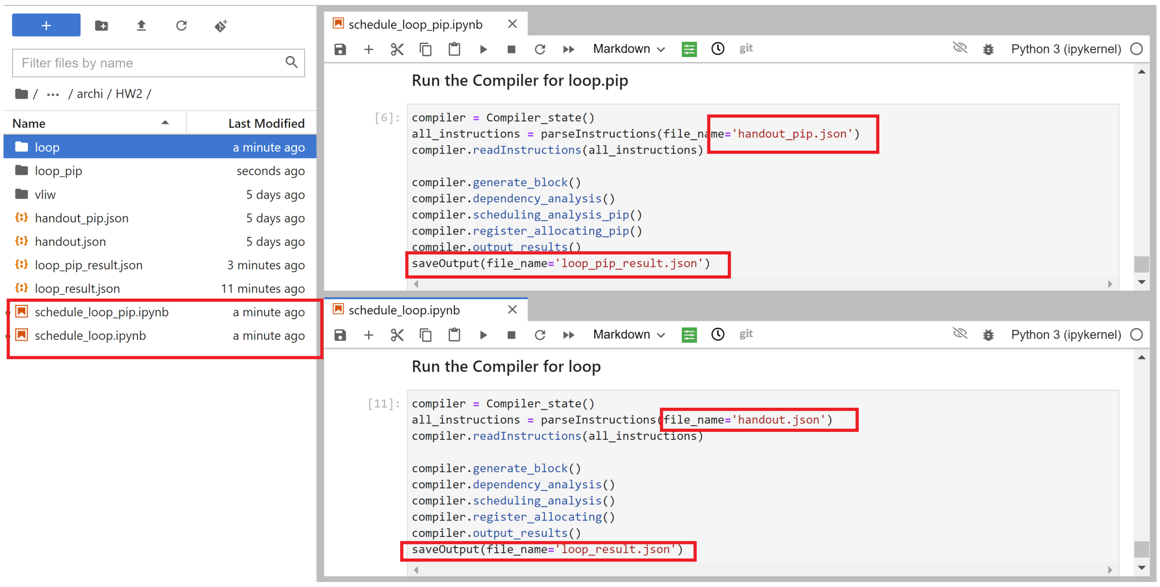
1 Coding Environment

I program the processor with Python. Four python packages are used, which are: (1) json to read/write json file; (2) numpy to manipulate calculations; and (3) copy to implement deep-copy of processor states; The code is tested under the environment "Jupyter lab" (you can refer to EPFL Noto server: noto.epfl.ch).

In order to run this scheduler, an handout.json file (change the filename shown in the code as you want) which contains input instructions should be provided under the same folder as the Notebook file, then an loop_result.json file will be output to show the schedule result.

There are two types of loop instructions, `loop` and `loop.pip`, their implementations are different in scheduling and register allocation, thus I separate them into 2 files, namely `schedule_loop_pip.ipynb` and `schedule_loop.ipynb`. It's possible to combine them into a single file, but the code structure can get ugly (thousands lines). As the name suggests, if your test cases contain `loop.pip`, please run the `schedule_loop_pip.ipynb` file, otherwise the `schedule_loop.ipynb` should be used.

The below plot gives the basic configuration:



Name	Last Modified
loop	a minute ago
loop_pip	seconds ago
vliw	5 days ago
handout_pip.json	5 days ago
handout.json	5 days ago
loop_pip_result.json	3 minutes ago
loop_result.json	11 minutes ago
schedule_loop_pip.ipynb	a minute ago
schedule_loop.ipynb	a minute ago

Run the Compiler for loop.pip

```
[6]: compiler = Compiler_state()
     all_instructions = parseInstructions(file_name='handout_pip.json')
     compiler.readInstructions(all_instructions)

     compiler.generate_block()
     compiler.dependency_analysis()
     compiler.scheduling_analysis_pip()
     compiler.register_allocating_pip()
     compiler.output_results()
     saveOutput(file_name='loop_pip_result.json')
```

Run the Compiler for loop

```
[11]: compiler = Compiler_state()
      all_instructions = parseInstructions(file_name='handout.json')
      compiler.readInstructions(all_instructions)

      compiler.generate_block()
      compiler.dependency_analysis()
      compiler.scheduling_analysis()
      compiler.register_allocating()
      compiler.output_results()
      saveOutput(file_name='loop_result.json')
```

Figure 1: File configuration

2 Date Structure

Two class objects are defined: (1) **schedule_cell**, it is used to indicate one bundle including 5 cells, e.g., ALU0, ALU1, Mult, MEM, and Branch; (2) Compiler_state, as the name suggests, it is used to finish all tasks of this project, e.g., parse instruction, dependency analysis, do scheduling and register allocation.

3 Scheduling Pipeline

3.1 Parse Instruction

The compiler can parse the provided instruction set (.json file) by invoking **readInstruction** function. In my implementation, each instruction and corresponding PC (start from 0) will be stored as one key-value pair (the dictionary in python). Moreover, to define the loop structure as earily as possible, i.e., to distinguish blocks **BB0**, **BB1** and **BB2**, the PC of loop instruction and the immediate loopStart are memorized by the compiler during the parsing process.

3.2 Dependency Analysis

In my implementation, the dependency analysis has 2 phases. In phase 1, the compiler will go through the parsed instructions, record their PCs, opcodes, destination registers (e.g., producer registers) and consumer registers, then assign them IDs (0,1,2...). In phase 2, four functions, namely, `find_Local_dep()`, `find_PostLoop_dep()`, `find_LoopInvar_dep()` and `find_InterLoop_dep()`, are used to find the dependencies. An attribute called Dependency_tab is used to store all these results, which is as follow:

```
self.Dependency_tab = {'Instr_Addr':[], 'ID':[], 'opcode':[], 'Des_reg':[], 'Local_dep':[],\
                       'InterLoop_dep':[], 'LoopInvar_dep':[], 'PostLoop_dep':[],\
                       'Consume_reg':[]}
```

Figure 2: Dependency analysis

- **find_Local_dep():** The compiler goes through each instruction block separately (e.g., **BB0**, **BB1** and **BB2**), it compares the consumer registers of current instruction with the producer registers of previous instructions located in current block. If there are multiple producers, only the final one will be considered.

- **find_PostLoop_dep():** The compiler goes through **BB2** block, it compares the consumer registers of current instruction with the producer registers of instructions located in **BB1** block. If there are multiple producers, only the final one will be considered.

- **find_LoopInvar_dep():** For instructions in **BB1**, the compiler compares their consumer registers with the producer registers of instructions located in **BB0** and **BB1**. Only consumers meet the below condition are included in this dependency:

  ```
  if Consumer in Producer_BB0 and Consumer not in Producer_BB1
  ```

  Loop invariant dependency is also possible in **BB2**, thus the compiler then goes through the instructions in **BB2**, and compare their consumer registers with the producer registers of instructions located in **BB0**, **BB1** and **BB2** (to avoid collision of local dependency). It needs to meet below condition:

  ```
  if Consumer in Producer_BB0 and Consumer not in Producer_BB1 and Consumer not in Producer_BB2
  ```

Same as before, if there are multiple producers, only the final one is included.

- **find_InterLoop_dep():** This dependency is only possible in **BB1**, which is the only one can have two producers. In my implementation, the compiler goes through the instructions in **BB1** and compares their consumer registers with the producers located in **BB1**. Here, I divide the producers in **BB1** into 2 parts, Producer_before_current (check to avoid local dependency) and Producer_withANDafter_current. Then only consumers meet below condition are included in this dependency.

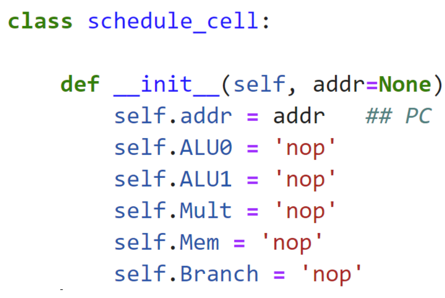
  ```
  if Consumer not in Producer_before_current and Consumer in Producer_withANDafter_current
  ```

  If the above condition is satisfied, meaning one inter-loop dependency producer is found in **BB1**, then the compiler will compare this consumer with the producers in **BB0**, to check if another producer appears.

3.3 Scheduling Analysis

3.3.1 For loop instruction

In my implementation, the scheduling analysis is divided into 4 steps. An empty bundle (based on class **schedule_cell**) is created first, which follows the below format. All these scheduled bundles are appended into a ordered list, to construct a schedule table.



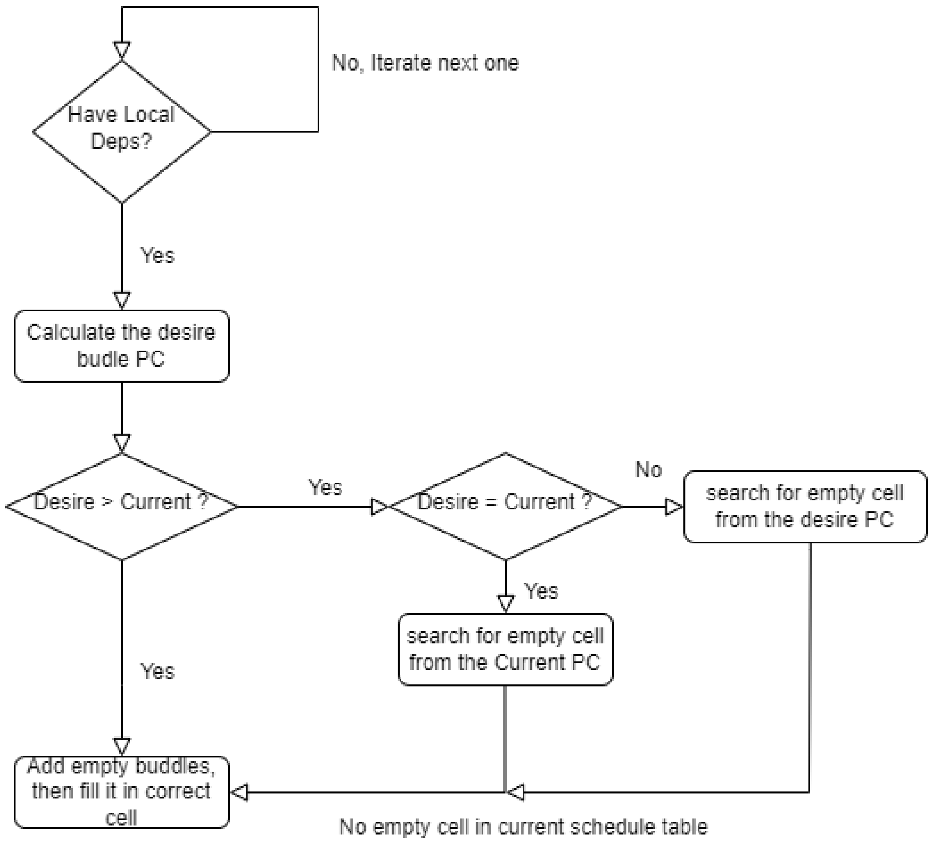
```
class schedule_cell:

    def __init__(self, addr=None):
        self.addr = addr    ## PC
        self.ALU0 = 'nop'
        self.ALU1 = 'nop'
        self.Mult = 'nop'
        self.Mem = 'nop'
        self.Branch = 'nop'
```

Figure 3: Format of schedule_cell

- **Step 1:** The compiler starts to go through the instructions in block **BB0** to solve the only possible **Local Dependency**. The figure below illustrates how this step conducts.

  As the figure shows, after checking the instruction if has local dependency, a function called `get_max_desire_addr` is used to find the desire scheduling bundle PC for the current instruction (the consumer). For example, if the schedule PC of producer (latency = 3 cycles) is 2, then the desire schedule PC for consumer is 4. After that, the desire PC will be compared with the Current PC (or the PC of last bundle in current schedule table). The comparison gives three possible cases, which will be solved appropriately.

- **Step 2:** In this step, the compiler schedules the instructions in **BB1** (corresponding to the phase-1 mentioned in Homework instruction file). There are 3 potential dependencies in this part (abbr. Local, interloop, loopInv), and my compiler will try to calculate the desire bundle position for each of them (similar to step 1). If the desire PC for Local is the largest one, the other two dependencies will be no issues and the process shown in Fig. 4 is conducted. Otherwise, empty bubbles will be added after the end position of BB0 scheduling to avoid phase-1 interloop and loopInv. Note: the distance between current shceduling instruction and the local producer should be always guaranteed.



Have Local Deps?
No, Iterate next one
Yes
Calculate the desire budle PC
Desire > Current ?
Yes
Desire = Current ?
No
search for empty cell from the desire PC
Yes
search for empty cell from the Current PC
Yes
Add empty buddles, then fill it in correct cell
No empty cell in current schedule table

Figure 4: Scheduling BB0 instructions

- **Step 3:** In this step, the compiler schedules the instructions in **BB2**. There are dependencies such as Local and postloop. The scheduling is similar to steps 1 and 2, and will not be repeated here.
- **Step 4:** In this step, the compiler tries to fix the phase-2 interloop. The scheduling follows the provided instruction handout, and will not be repeated here.

3.3.2 For loop.pip instruction

The scheduling of code including loop.pip keeps steps 1 and 3 mentioned above to solve dependencies in **BB0** and **BB2**. For instructions in **BB1**, I use a while-loop to keeping scheduling until all interloop can meet equation $S(P)+\lambda(P) \leq S(C)+II$. Moreover, a new structures called Reserve_tab is used to remember which slot is reserved, which consists of II bundles. Each bundle has its own cycle name, from 0 to $II-1$. And all loop-stages in my scheduling table are copied from the current state of Reserve_tab, so that the reserved information are always attached. A print format of Reserve_tab is shown below. With these newly provided features, the instructions are scheduled

```
--------------
Reserve_tab:
addr: None, ALU0: Reserved, ALU1: nop, Mult: nop, Mem: Reserved, Branch: nop, cycle: 0
addr: None, ALU0: nop, ALU1: nop, Mult: Reserved, Mem: Reserved, Branch: nop, cycle: 1
addr: None, ALU0: nop, ALU1: nop, Mult: Reserved, Mem: nop, Branch: Reserved, cycle: 2
--------------
```

Figure 5: Format of reserve table

in the similar way to the case including loop.

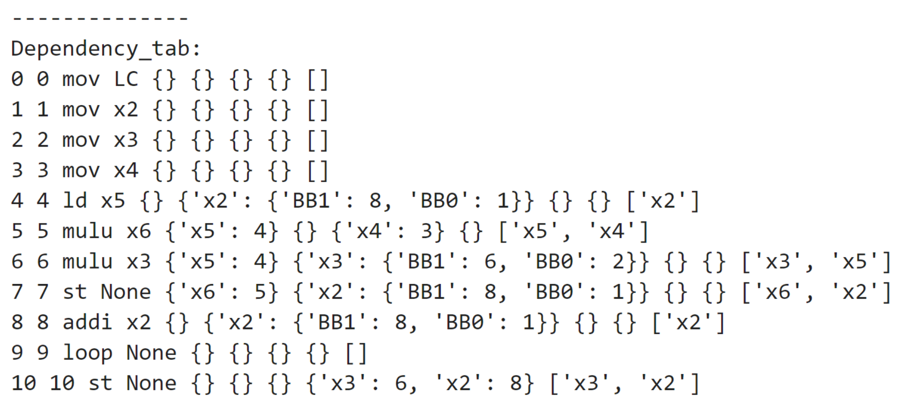
3.4 Register Allocation and Renaming

The methods applied here are highly correlated with the provided instruction, so I will not repeat too much here.
For loop case:

- The compiler starts to check each cell in scheduling table (see Fig). When it sees the instructing ID, it will jump to the corresponding column of Dependency table and rename the destination register. This renaming link will be recorded in Register_rename_tab.
- Before going to the next cell, the compiler will check if the instruction scheduled into the current cell has consumers (By looking up the Consume_reg column of dependency table). If the answer is no, meaning that the current instruction has no dependencies, the corresponding instruction operands are fetched by compiler and concatenated with the renamed destination register in correct format. If the answer is yes, the compiler replace the consumer operands with the renamed ones by looking up the Register_rename_tab (if the producer has been already reamed), then do the concatenation. The revised instructions will be written back to the corresponding cell of scheduling table.
- This step is used to rename the consumers of instructions that cannot be revised in above step.
- Finally, all instructions are revised and can be output in json format. Note: The loopStart is changed according to the scheduling table.

For loop.pip case: The strategy has changed slightly according to the homework instructions, but most of the data structures mentioned in the loop case are still used to facilitate register allocation and renaming. The below gives a clear illustration of the data structures I used.

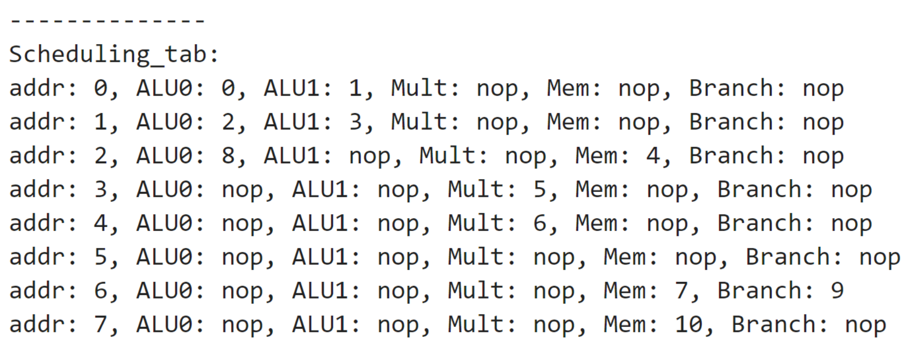
Results of dependency analysis for loop and loop.pip:



```
--------------
Dependency_tab:
0 0 mov LC {} {} {} {} []
1 1 mov x2 {} {} {} {} []
2 2 mov x3 {} {} {} {} []
3 3 mov x4 {} {} {} {} []
4 4 ld x5 {} {'x2': {'BB1': 8, 'BB0': 1}} {} {} ['x2']
5 5 mulu x6 {'x5': 4} {} {'x4': 3} {} ['x5', 'x4']
6 6 mulu x3 {'x5': 4} {'x3': {'BB1': 6, 'BB0': 2}} {} {} ['x3', 'x5']
7 7 st None {'x6': 5} {'x2': {'BB1': 8, 'BB0': 1}} {} {} ['x6', 'x2']
8 8 addi x2 {} {'x2': {'BB1': 8, 'BB0': 1}} {} {} ['x2']
9 9 loop None {} {} {} {} []
10 10 st None {} {} {} {'x3': 6, 'x2': 8} ['x3', 'x2']
```

Figure 6: Results of dependency analysis

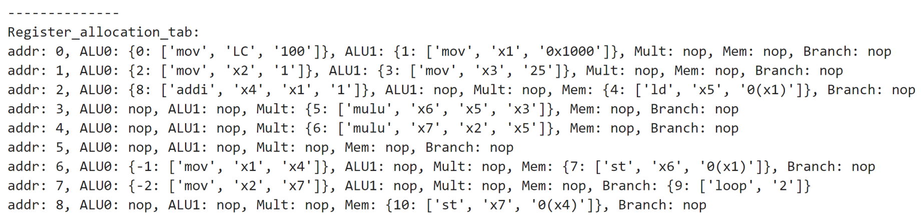
Results of scheduling analysis for loop:



```
--------------
Scheduling_tab:
addr: 0, ALU0: 0, ALU1: 1, Mult: nop, Mem: nop, Branch: nop
addr: 1, ALU0: 2, ALU1: 3, Mult: nop, Mem: nop, Branch: nop
addr: 2, ALU0: 8, ALU1: nop, Mult: nop, Mem: 4, Branch: nop
addr: 3, ALU0: nop, ALU1: nop, Mult: 5, Mem: nop, Branch: nop
addr: 4, ALU0: nop, ALU1: nop, Mult: 6, Mem: nop, Branch: nop
addr: 5, ALU0: nop, ALU1: nop, Mult: nop, Mem: nop, Branch: nop
addr: 6, ALU0: nop, ALU1: nop, Mult: nop, Mem: 7, Branch: 9
addr: 7, ALU0: nop, ALU1: nop, Mult: nop, Mem: 10, Branch: nop
```

Figure 7: Results of scheduling analysis

Results of register allocation and renaming for loop:



```
--------------
Register_allocation_tab:
addr: 0, ALU0: {0: ['mov', 'LC', '100']}, ALU1: {1: ['mov', 'x1', '0x1000']}, Mult: nop, Mem: nop, Branch: nop
addr: 1, ALU0: {2: ['mov', 'x2', '1']}, ALU1: {3: ['mov', 'x3', '25']}, Mult: nop, Mem: nop, Branch: nop
addr: 2, ALU0: {8: ['addi', 'x4', 'x1', '1']}, ALU1: nop, Mult: nop, Mem: {4: ['ld', 'x5', '0(x1)']}, Branch: nop
addr: 3, ALU0: nop, ALU1: nop, Mult: {5: ['mulu', 'x6', 'x5', 'x3']}, Mem: nop, Branch: nop
addr: 4, ALU0: nop, ALU1: nop, Mult: {6: ['mulu', 'x7', 'x2', 'x5']}, Mem: nop, Branch: nop
addr: 5, ALU0: nop, ALU1: nop, Mult: nop, Mem: nop, Branch: nop
addr: 6, ALU0: {-1: ['mov', 'x1', 'x4']}, ALU1: nop, Mult: nop, Mem: {7: ['st', 'x6', '0(x1)']}, Branch: nop
addr: 7, ALU0: {-2: ['mov', 'x2', 'x7']}, ALU1: nop, Mult: nop, Mem: nop, Branch: {9: ['loop', '2']}
addr: 8, ALU0: nop, ALU1: nop, Mult: nop, Mem: {10: ['st', 'x7', '0(x4)']}, Branch: nop
```

Figure 8: Results of register allocation and renaming