Design Handout of a Cycle-by-Cycle Simulator of an Out-of-Order Processor

GAO Xufeng

April 10, 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; (3) copy to implement deep-copy of processor states; and (4) ctypes to provide C-like unsigned 64 integer type. The code is tested under the environment "Jupyter lab" (you can refer to EPFL Noto server: noto.epfl.ch).

In order to run this processor, an test.json file which contains input instructions should be provided under the same folder as the main.ipynb file, then an test_result.json file will be output to show the processor states in each cycle. The below plot gives the configuration:

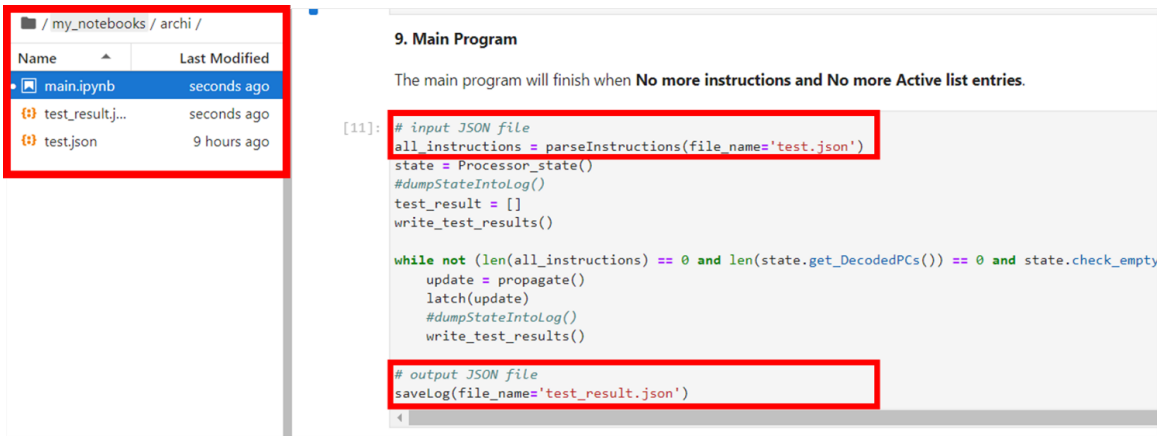

Figure 1: File configuration

2 Design principles

2.1 Date Structure

Three class objects are defined: (1) Processor_state, it is used to record the states of all data structure, e.g., Active List and Integer Queue; (2) Active_List_cell, as the name suggests, it is used to record all information stored in one entry of Active List, e.g., LogicalDest; and (3) Integer_Queue_cell, it works similar as the Active_List_cell but for Integer Queue. Thus, a complete Active List contains multiple Active_List_cell objects, while a complete Integer Queue consists of Integer_Queue_cell objects.

2.2 Pipeline Structure

The pipeline structure of the processor is shown as:

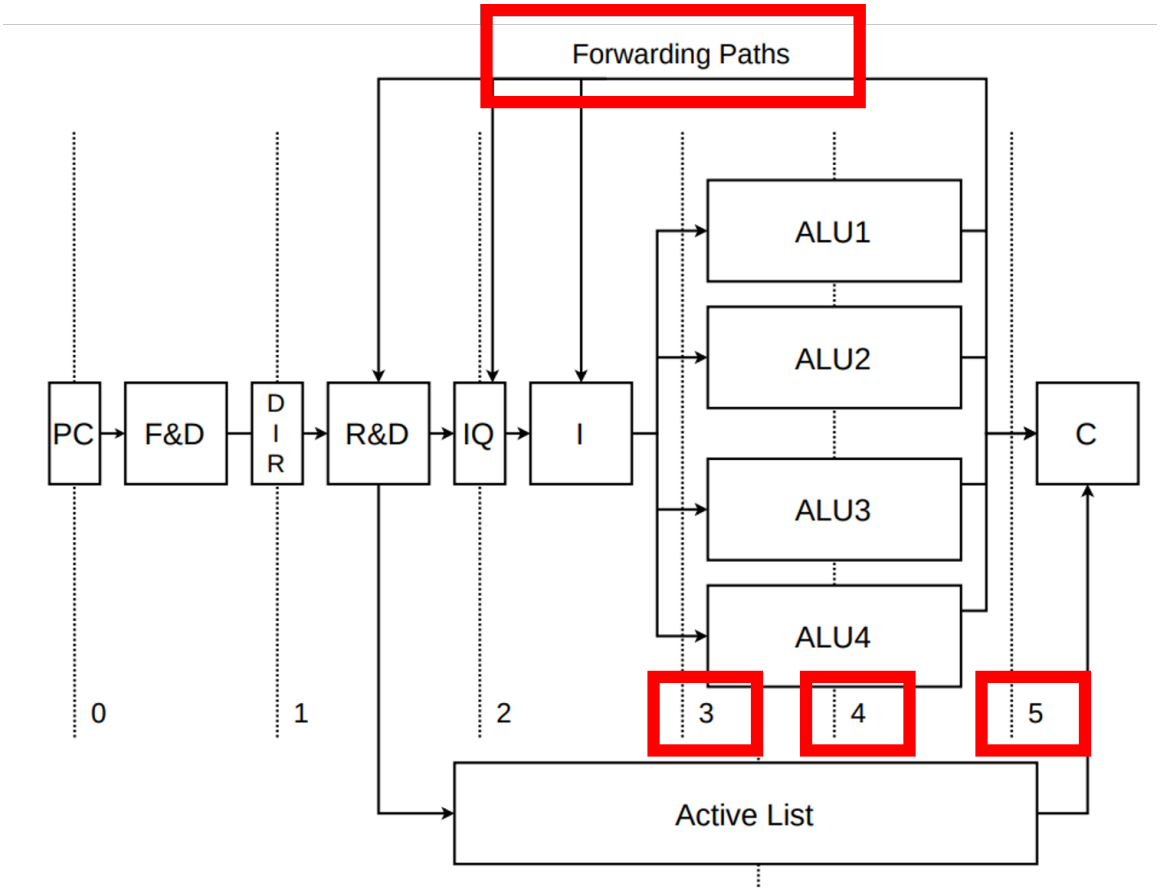

Figure 2: The pipeline structure of the processor

In addition to the structures shown in above plot such as Free List and DIR, I introduced some auxiliary pipeline structures to make my design more flexible. IssueQueue works as a pipeline register between Issue stage and ALU (say register 3 in Fig.2), which contains the most 4 instructions picked by Issue stage. In the 1st ALU cycle the instructions are performed and the corresponding results (most 4) are stored in a pipeline register called ALU_registers (say register 4 in Fig.2). The IssueQueue and ALU_registers are two normal registers to complete the pipeline. In the 2nd ALU cycle, the ALU results are supposed to be broadcast to forwarding paths. From Fig.2, (a): Three paths are connected to R&D, IQ and Issue Stage directly, so I think these 3 units can see and use the ALU results in the same cycle that the ALU results are broadcast; (b) In 3.4 of the provided handouts, it mentions that the Commit unit is capable of marking instructions done or exception on receiving results from the forwarding paths. According to Fig.2 above, the forwarding path between ALUs and the Commit Unit is latched by an intermediate pipeline register 5. Thus, there is no combinational connection from the forwarding path to Commit Unit, and I think this register 5 will cause one cycle delay for Commit Unit to see the ALU results broadcast in current cycle, because the register is only latched on the rising edge of clock. Thus, for example, if ALU results are broadcast in cycle 5, the R&D, IQ and Issue Stage can see and use them in cycle 5, while the Commit Unit can only see and use them to mark entries of Active List with Done/Exception flags in cycle 6.

However, by following the provided test_result.json file, when ALU results are released to forwarding paths in cycle 5, we can see that both operations, the “mark Done/Exception” and “update

Physical Register File & Integer Queue", are performed and finished in same cycle. It is obvious that this result cannot follow the pipeline structure shown in Fig.2. After discussing with TA (Shanqing LIN), we think it's indeed a bug. The only way I can make to follow the test_result.json it to mark Done/Exception the entries of Active List (without using Commit Unit) as soon as ALU results are broadcast.

Thus, to meet the requirements of broadcasting ALU results in time, I introduced 2 separate forwarding paths. The first one is called forwarding_path (say Forwarding paths in Fig.2), which is a combinational wire going from ALU to R&D, IQ, and I. The second one is called Done_or_Exception (say register 5 in Fig.2), which is mainly used to mark the entries of Active List with Done/Exception flags.

2.3 Fetch & Decode Stage

- This stage is designed to keep fetching & decoding instructions until these is an Exception, Backpressure or No more instructions case.
- During the Exception Mode, this stage will stall, and the PC will be set to 0x10000.
- The handout didn't mention how the processor works after exception mode. Thus, I assumed the whole pipeline will keep stall, although there are left instructions in the JSON file (like in memory).

2.4 Rename & Dispatch Stage

This stage keeps Renaming & Dispatching instructions until there is an Exception, No more empty spaces in Active ListInteger Queue, No more free registers in Free List, No more DIR instructions case.

2.5 Issue Stage

This stage will issue 4 ready instructions (at most) until there is an Exception or No more entries in Integer Queue case. The pipeline register IssueQueue is used to store these issued instructions.

2.6 Execution Stage

This stage takes 2 cycles:

- In the 1st cycle, most 4 issued instructions are executed by available ALUs. The calculated results will be stored in the pipeline register ALU_registers.
- In the 2nd cycle, these results will be broadcast to forwarding paths.
- This stage will be reset if there is an exception.
- Because this is a cycle-accurate simulator, the logic in 2nd ALU cycle is executed before the ones in 1st cycle to update each data structure correctly.

2.7 Commit Stage

This stage is considered to work with 2 modes:

- **Normal mode (without exception):** (1) The active list is scanned regularly, and at most 4 instructions (marked with Done) will be retired in one cycle. (2) If the head instruction in active list is not completed yet, the commit stage will stall and only those instructions already picked will be retired. (3) When an instruction triggers an exception, the commit stage will stall. Because the **Exception Mode** works as a register in our system, the commit stage cannot change its content in current cycle. Thus, a special bus called **bus_to_exception** going from commit stage to exception register is introduced, which causes the contents of exception register to be updated in the next cycle. To make things easier, I assumed this bus contains 2 fields: one for exception trigger signal and the other for recording exception PC. Therefore, at the start of the next cycle, the **Exception_register** will set the exception flag and store the exception PC, and all other stages can read from it (i.e., can enter exception mode).

- **Exception mode:** All other stages will see the exception flag and enter the exception mode. For commit stage, it will recover the Register Map Table, Free List and Busy Bit Table accordingly. Finally, when Active list become empty, the **bus_to_exception** will be reset in the same cycle. Thus, at the start of next cycle, all stages will know exception is handled, and they will quit the exception mode. In this last cycle, the PC is 0x10000, the Exception PC is the PC pointing to the instruction triggering exception, and the Exception flag is false.