# Modeling Out-of-Order Execution Core

The goal of this assignment is to help you understand the basics of out-of-order execution. Your tasks for this assignment are 1) to model an out-of-order execution core based on reservation station renaming, and 2) to analyse a set of traces to find how various ILP parameters (issue width, reservation station size, and ROB size) affect processor performance and how much ILPs are available for the given traces.

## Instruction Traces

### A. Format

A set of dynamic instruction traces collected from SPEC2006 benchmarks will be provided. The traces have simplified information about instructions executed, including instruction types, source operands, destinations, and memory addresses. The format of the traces is as follows.

```
<inst type> <destination> <src1> <src2> <addr>
```

1. `<inst type>`: Specify one of the following four types of instructions
    - IntAlu: integer ALU instruction
    - MemRead: load instruction
    - MemWrite: store instruction

2. `<destination>`: Destination register.
    - If 0, no output register (e.g. stores)
    - else, result is written to R# (#=`<destination>`)

3. `<src1> <src2>`: Source register.
    - If 0, no source operand
    - else, source operand1 or operand2 is R# (#=`<src>`)

4. `<addr>`: If <inst type> is MemRead, <addr> is the load address. If MemWrite, <addr> is the store address. Else, this field must be ignored.

5. There are 16 architectural registers (R1-R16) in the RSA.

### B. Examples

- `IntAlu 9 7 0 0`: An IntAlu type instruction with destination register R9, and one source operand R7.
- `MemWrite 0 6 7 2000`: A MemWrite type instruction with two source operands, but no output register. The actual address for the store is 0x2000.

# Task 1: Modeling Out-of-Order Execution Core

A core model for this assignment processes an input instruction trace, and simulates the backend of out-of-order execution core. We assume the front-end parts (BTB, branch prediction, icache, etc) are perfect, so the front-end can always feed the backend with N instructions (on a correct execution path) every cycle, as long as the back-end can process them.

The core model must use a centralised reservation station as a renaming and scheduling mechanism. For this assignment, you do not need to model a load-store-queue.

## A. Pipeline

- IntAlu: Fetch → Decode → Issue → Ex → Commit
- MemRead: Fetch → Decode → Issue → Ex1 → Ex2 → Ex3 → Commit
- MemWrite: Fetch → Decode → Issue → Ex → Commit

*N* is the issue width in the following description. (*N*-way superscalar)

- **Fetch**: Read upto *N* instructions from a trace. If the fetch queue is full, instruction fetch is stalled. Assume the fetch queue size is equal to 2*N*.
- **Decode**: Decode upto *N* instructions from the fetch queue. Rename source and destination registers and put instructions into the reservation station and the ROB. Once an instruction is moved to the reservation station (and the ROB), remove it from the fetch queue. If any of the ROB or reservation station is not available, instructions must stay in the fetch queue and no more instructions are decoded. (Assume register/ROB reads occur at this stage.)
- **Issue**: Select up to *N* ready instructions from the reservation station, and send them to execution units. When multiple instructions become ready at a cycle, ***you should pick the oldest instruction first***. A reservation station entry is not deallocated until the execution is completed.
- **Execute**: Execute instructions, but in this trace-baced model, nothing really happens. Instructions just consume the execution resources. When the execution of an instruction is completed, make the destination available by updating the reservation station and ROB. ***If there are any pending instructions in the reservation station, the instruction will become ready to be issued at the next cycle***.
- **Commit**: Commit completed upto *N* instructions starting from the head of ROB. Update the register rename map.

**Example**

```
IntAlu 1 2 3 0
IntAlu 4 5 1 0
```

- The second instruction is dependent on the first instruction.
- For a single-issue processor:

| | Instruction 1 | Instruction 2 |
|---|---|---|
| Cycle 1: | Fetch | |
| Cycle 2: | Decode | Fetch |
| Cycle 3: | Issue | Decode |
| Cycle 4: | Ex | Stall |
| Cycle 5: | Commit | Issue |
| Cycle 6: | | Ex |
| Cycle 7: | | Commit |

## B. Structures to Model

- **Fetch Queue**: Allocated at the fetch statge. Deallocated at the decode stage. The number of entries is 2*N*. It holds the fetched instructions until they are moved to the ROB and the reservation station.
- **Reservation Statione**: Allocated at the decode stage. Deallocated at the last execution stage. The size is configurable.
- **Rename Map**: points to the most recent value for registers (either architectural register or ROB).
- **Execution Units**: *N* ALUs to process all IntAlu type and the address calculation of MemRead/MemWrite. (1 cycle latency)
- **Reorder Buffer**: allocated at the decode stage. Deallocated at the commit stage.

Any structures deallocated at cycle *N* must be reusable at cycle *N+1*. (but not at cycle *N*)

## C. Memory Operations

For this assignment, you do not need to model properly the execution of memory instructions. The address field of instruction traces is not used for this assignment.

- Load and store instructions also consume a reservation station entry and an ALU

execution unit to calculate the load/store address.

- Ignore memory dependency. A load can be issued as soon as the source operands are ready.
- A store is just committed, when it becomes the head of ROB and the execution is completed. (Assume we have infinite write buffers.)
- This assignment does not require modeling caches. A load takes a fixed latency of 2 cycles, assuming they are all cache hits. Three execution stages are needed: Ex1(address calculation)-Ex2-Ex3.

## D. Output

The program is required to get a dump option from the config file. If a dump option is turned on, print the content of Reservation Station and ROB every cycle. Print out the final stats regardless of the dump option.

- If `dump` is 0, do not print out any dump states.
- If `dump` is 1, print out ROB states only.
- If `dump` is 2, print out RS and ROB states.

**Output format (10 RS entries, and 20 ROB entries)**

```
= Cycle 1
RS1   : ROB1 <src1> <src2>
...
RS10  : ROB2 <src1> <src2>

ROB1  : <status>
...
ROB20 : <status>
```

- Reservation station entries must be printed out from RS1 to RSn (fixed).
- ROB entries must be printed out from the oldest one to the youngest one. (From ROB1 to ROB20 in the example)
- RS: <src> can be V for valid, or ROBid if pending
- ROB: <status> can be C (completed) or P (pending).

**Final Stats**

```
Cycles            89
IPC               1.12
Total Insts       100
IntAlu            71
MemRead           20
MemWrite          9
```

You sould keep the order and format as above. Do not consider how many spaces there are if you give more than one spaces between each name and number.

# Task 2: ILP Characterization

Vary the following parameters to see the available ILP from the given traces.

- Issue with: 1 - 8
- Reservation station size: ROB/2, ROB
- ROB size: 16 – 512

## Simulator Options

Your program should require config file name and trace file name as options for TAs to check the functionality of your simulator, i.d. your program should run using the following command on the terminal.

```
core_simulator config_file_name trace_file_name
```

**Config file**

The config file will have the following lines.

```
Dump
Width (fetch = issue = commit width)
ROB size
Reservation station size
```

Below is an example of a config file.
(Dump=0, width=2, ROB=128, ReservationStation=64)

```
0
2
128
64
```

You can add your own options after the required options. However, your simulator must work even if only the required options are given.

# Report

You have to submit a report which contains the following.

- Graphs for measured IPC varying parameters and workloads.
- Analysis of the graphs.
- The compilation method and environment.
- The execution method and environment.

# Submission

Submit all your files, (source code, config files, report, and so on) in the single compressed file, "your_student_number_hw2.zip". ex) 201800000_hw2.zip

Due date is ***November 27th 11:59 PM***.

# Notes

- Your source code must be compilable or your score will be 0.
- Please do not contain the workloads in the submission file.
- If you are not following the given instructions, you can be penalized.

# Late Penalty

- One day: Your score will be 70%.
- Two days: Your score will be 40%.
- Three days: Your score will be 10%.
- More days: Your score will be 0.