18-640: Foundations of Computer Architecture

**Project 2: Modern Tomasulo’s Algorithm and Out-of-Order Execution**

Group NO.7 Luting Wang (lutingw) Wei Feng (wfeng1)

Part 1: Tomasulo’s Algorithm

In this part, we are trying to implement Tomasulo’s Algorithm. Tomasulo’s Algorithm is one of the most famous hardware algorithm implemented by Robert Tomasulo in 1967. The basic idea behind it is to make sequential instructions execute non-sequentially. The algorithm uses reservation stations, register tags and common data bus as auxiliary structure in order to reslove data, anti- and output dependencies.

1. Implementation

We implemented two reservation stations, one is for add operations and another is for multiplication operations. Each reservation station slot contains two tags and two ready fields for the operands, and one tag for the destination. The number of reservation station slots is defined in the configuration file, which also contains the adder latency, multiplier latency and maximum issue rate. Also, we implemented a physical register file structure with a slot of 64, to mimic the behavior of registers in hardware. It contains the data of the register and a tag. We also use a Map table to index into physical register file. Because data is no longer stored in the reservation station, so we need 32 entries in physical register to store those values, and map is used to index into these entries. And we use Freelist to pop a free register. Although we didn’t use ROB, the implementation is compatible with the functions given. The detailed implementation is discussed below:

**initTomasulo()**

In this function, we initialize all the data structures. It first reads parameters from the “config.default” file, which declares the number of adder and multiplier reservation station slots. Then we initialize the reservation station slots: set the tag field as -1 and Rdy field as -1 (ready). We also introduce another field “Empty” to indicate whether this slot is empty, and initialize it as 1(empty). The Dst is stored the number of reservation station. It is shown in the table1.

Table 1 Initialization of the reservation station

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Empty | Dst | Src1 | | | Src1 | | |
| Tag | Rdy | | Tag | Rdy | |
| 1 | 0 | -1 | | -1 | -1 | | -1 |
| 1 | 1 | -1 | | -1 | -1 | | -1 |

Then we initialize Map table. We set the r0 to r31 as 32 to 63. And for the physical register, all the Rdy filed is set as -1 and data filed as 0. And we set freelist as 0 to 31.

**int issue(instruction\_t \*theInstruction)**

In this function, we attempt to issue an instruction. First we determine the instruction type. And then we check the related reservation station to find an empty slot. Next we assign field values to this slot. The first operand could only be a register operand (eg. r13). We check the related register entry’s (13) Rdy filed to know whether this operand is ready. If it is ready, a new register ID is pop from the freelist (0), and though Map index, we find it is indexed to the 32th register, and we pass the already value to it in the physical register(32). And the tag in reservation station is set to 32, and Rdy field is set to -1. If it is not available, a new register ID is pop from the freelist (0), and though Map index, we find it is indexed to the 32th register, and because the data is not ready, so we pass the related register entry’s (13) Rdy filed (eg. 2) to the reservation station Rdy field, and the tag filed is set to 32. And for the second operand, if it is an immediate, a new register ID is pop from the freelist (1), and though Map index, we find it is indexed to the 33th register, and we put the value in it (33). And the tag in reservation station is set to 33, and Rdy field is set to -1 meaning it is ready.

**int execute(mathOp mathOpType, executeRequest\_t \*executeRequest)**

In this function, we find the already instruction in the reservation station and send it to functional unit. If there are several ready instructions, we pick the one who has waited the longest. To do this, we add a counter to each slot, and reset it to 0 if it is empty again. And in each cycle when calling the function, the counter will be incremented, which indicates the time each instruction in this slot has waited. Then we pass the Dst tag and operand value through indexing the Map table to the functional unit.

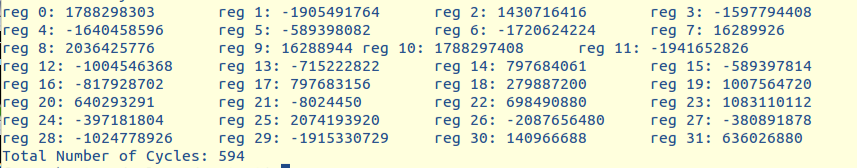
**void writeResult(writeResult\_t \*theResult)**

In this function, we modify the register file and reservation station. First we use the tag to update the register file. We search the Rdy field to find whether there is a register needing this value, and put the value into related data field and set the Rdy as -1. Then we go through the reservation station Rdy field to search a match, and use related Tag field to index the Map table, and put the value into the indexed physical register’s data field, and set the Rdy filed to -1. Once it has finished execution, we clear the slot and it can be inserted another instruction.

**int checkDone(int registerImage[NUM\_REGISTERS])**

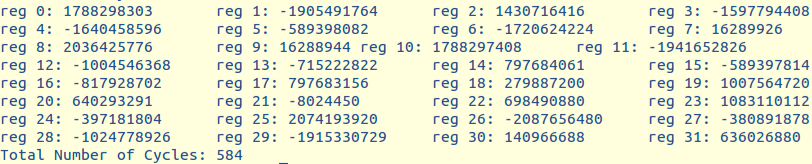
In this function we try to know whether the simulation has finished. It is done by checking the register Rdy fields (0-31). If all the register’s Rdy fields are -1, which means they are all ready, we can say that the simulation has ended and return 1. So there is no need to implement ROB to test the end of simulation. And the values of these 32 registers is the final value of them.

The following image shows the results on a medium size trace with the default configuration (3, 5, 2, 2, 1).



Figure

And the following image shows the result on the same trace but increasing the size of reservation stations and maximum issued instructions per cycle (4, 6, 3, 3, 2).



Figure

As you can see, the number of cycle has decreased, which means an increase of throughput performance. It will be explained in the Question Part.

1. Questions
2. **Explain how Tomasulo's algorithm avoids WAW, RAW, WAR hazards. If we made the Instruction issue out-of-order, do any of these hazards now exist?**

Tomasulo avoids WAW, RAW and WAR hazards by renaming registers through adding tag filed in it, and broadcasting tags through CDB (common data bus) after instructions’ execution. Every time we want to write the register we will rename of that register to a register tag, which is taking from the ID of the reservation station who contains that instruction. And the CDB will transfer data from the execution result to those reservation slots and registers who needs the result. Once there is a match, they will pull in the result, and this can resolve WAW and WAR dependency, thus to reduce pipeline stall. And for RAW hazards, it is resolved in execution stage. Instructions can’t be issued until its operands are available, which is indicated by a tag. It keeps waiting in the reservation station and monitors CDB. When it detects a match, it will latch the operand and the operand becomes ready. So RAW hazards can be resolved.

Therefore, if we made the instruction issue out-of-order, all of the hazards would be resolved. To sum up, WAW hazard is resolved by changing the tag in registers, WAR is resolved by broadcasting tag and result in CDB, and RAW is resolved by matching a tag in the reservation slot whose operand is not available and wait to read data from the CDB.

1. **How did you generate tags in your implementation? Is this how you would do it in hardware?**

In our implementation, tags have multiple usages. Most of time it contains the number of reservation slots. Supposed we have N adder reservation slots and M multiplier reservation slot, 0 to N-1 is the number of adder reservation slots, and N to N+M-1 is the number of multiplier reservation slots. And for the Dst tag, it also indicates whether this slot is empty for a new instruction to be inserted. When tag is -1, it means this reservation is empty. When it finishes execution, the tag would be set to -1 to indicate this slot is empty now and can be inserted into a new instruction. Initially, we set all the tags in Dst as -1 (all reservation slot is free).

In hardware, we can add tag fields and assign values to them, just the same way as we do in implementation. The difference is: in implementation, we need to send tags to the execute() function, and return the value and tag in writeResult() function to find whether there is a match. But in hardware, we don’t need to send the tag. All tag fields are connected to the CDB and keeping monitor it until they find a match and latch the data from CDB.

1. **Try increasing the number of slots in your reservation stations and the maximum issue rate. Why does performance improve even though only 1 instruction can begin execution per cycle for a single reservation station? When is it better to increase the maximum instructions issued per cycle? When is it better to increase the number of reservations station slots?**

Performance is improved even though only one instruction can begin execution per cycle. It is because we have many slots in a single reservation station which contains idling instruction to be issued until it is ready. So the number of idling cycles is reduced due to increased dispatched instructions, therefore improves performance.

As for when we should increase the maximum instructions issued per cycle or increase the number of reservation station slots, I think it depends on the instruction dependency. If instruction dependency is few, it is better to increase maximum instructions issued per cycle. Because of few dependency, most of the time instruction operands are available and it can be executed immediately. If instruction dependency is frequent, it is better to increase the number of reservation station slots. Because instructions tend to idle in the reservation station waiting for operands to be ready, and if we have more slots, more instruction can be executed if there is a match in CDB.

1. **Although your implementation only had a single reservation station per functional unit, it is possible to have multiple reservation stations with same functionality (e.g.,4 separate reservation stations, each with a multiplier functional unit) to increase overall throughput. How does one choose the right number of reservation stations?**

In most cases, more reservation stations we have, higher overall throughput we would get. The reason is obvious: we can have more instructions idling in many reservation stations to reduce idled cycles. But it would increase the hardware cost. Since reservation station is not cheap in hardware, we should balance cost and throughput. The number of reservation stations should be based on several factors: hardware cost, throughput performance and instruction dependency, etc.

Part 2: Out-of-Order Execution

In this part, we are working with Out-Of-Order execution by using FS simulation mode for Alpha architecture. And PARSEC benchmarks will be used for testing.

1. Results

We record the buffer usage in Instruction Queue, Load Queue, Store Queue and ROB by adding counters to related files in src/cpu/o3/( inst\_queue.hh, inst\_queue\_impl.hh, lsq.hh, lsq\_impl.hh, rob.hh, rob\_impl.hh, iew\_impl.hh, iew.hh, cpu.cc).

For instruction queue, we increment the related counters in updateInstQueueCount()

*template<class Impl>*

*void*

*InstructionQueue<Impl>::**updateInstQueueCount(){*

*int numberOfUsedBuffer = countInsts();*

*float utilization = numberOfUsedBuffer\*1.0/64;*

*if(utilization < 0.25) InstQueueB1++;*

*else if(utilization < 0.50 ) InstQueueB2++;*

*else if(utilization < 0.75 ) InstQueueB3++;*

*else InstQueueB4++;*

*}*

For Load Queue, we increment the related counters in updateLoadQueueCounter()

*template<class Impl>*

*void*

*LSQ<Impl>::**updateLoadQueueCounter(){*

*int numberOfUsedBuffer = numLoads();*

*float utilization = numberOfUsedBuffer\*1.0/32;*

*if(utilization < 0.25) LoadQueueB1++;*

*else if(utilization < 0.50 ) LoadQueueB2++;*

*else if(utilization < 0.75 ) LoadQueueB3++;*

*else LoadQueueB4++;*

*}*

For Store Queue, we increment the related counters in updateStoreQueueCounter()

*template<class Impl>*

*void*

*LSQ<Impl>::**updateStoreQueueCounter(){*

*int numberOfUsedBuffer = numStores();*

*float utilization = numberOfUsedBuffer\*1.0/32;*

*if(utilization < 0.25) StoreQueueB1++;*

*else if(utilization < 0.50 ) StoreQueueB2++;*

*else if(utilization < 0.75 ) StoreQueueB3++;*

*else StoreQueueB4++;*

*}*

For ROB, we increment the related counters in updateBufferCounter()

*template <class Impl>*

*void*

*ROB<Impl>::updateBufferCounter(){*

*int numberOfUsedBuffer = numInstsInROB;*

*float utilization = numberOfUsedBuffer\*1.0/192;*

*if(utilization < 0.25) ROBufferB1++;*

*else if(utilization < 0.50 ) ROBufferB2++;*

*else if(utilization < 0.75 ) ROBufferB3++;*

*else ROBufferB4++;*

*}*

And we call these functions in iew\_impl.hh and CPU.cc in the tick() function.

Then we run runscript\_1.rcS and runscript\_2.rcS to find utilization for blackscholes and x264 benchmark, and get four separate sets of statistics for each counters at the end of simulation. To evaluate the utilization of these benchmarks, we normalize these values by dividing the total number of cycles and draw a graph as below.

Table 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Blackschole*s* | | | | X264 | | | |
|  | B1 | B2 | B3 | B4 | B1 | B2 | B3 | B4 |
| IQ | 0.7720 | *0.0954* | *0.0352* | *0.0974* | 0.7742 | *0.0946* | *0.0350* | *0.0962* |
| LQ | *0.8099* | *0.0947* | *0.0872* | *0.0082* | *0.8117* | *0.0941* | *0.0861* | *0.0081* |
| SQ | *0.9105* | *0.0394* | *0.0091* | *0.0410* | *0.9111* | *0.0391* | *0.0010* | *0.0408* |
| ROB | *0.7002* | *0.1565* | *0.1220* | *0.0213* | *0.7034* | *0.1550* | *0.1205* | *0.0211* |

1. Questions
2. **Analyze the results you collected. Did you find anything interesting?**

By looking at the plot, an interesting phenomenon can be discovered that: for most of time, the utilization of instruction queue, load/store queue and ROB is less than 25%. The same trend can be seen in different benchmark.

The low usage of instruction queue may because of the control dependency within the program. If one of the instructions is a branch and the subsequent instructions would be discarded if the branch is taken.

The low usage of load/store queue may because of there are only a small amount of load/store instructions in the program. So the load/store queue is always occupied a small size in the buffer.

And for the ROB, the low usage may due to the sequential program flow. Because ROB is used to reorder the out-of-order execution results, and to some degree it represents how deep the CPU is executing instructions out of order. So low usage of ROB means the benchmark is somehow sequential. There may be lots of dependencies in the benchmark program, which leads to the sequential execution of the CPU.

And the difference between these two benchmarks is very slight. The second one’s buffer utilization is a bit smaller than the first one. It may because there is more dependencies and more load/store instructions in the first benchmark.

1. **If you were asked to report number of cycles an Instruction spends at various stages of pipeline, how would you go about it? What result do you expect?**

If we need to report the number of cycles an instruction spends at various stages of pipeline, we will set several counters to record the number of cycles for all the instructions going through each stage. And then divided them by the total number of instructions. Then we can get the average time an instruction spent on each stage, i.e. CPIj (j is the stage number). The result can be expected that the total CPI will be larger than CPIj in any stage of pipeline.