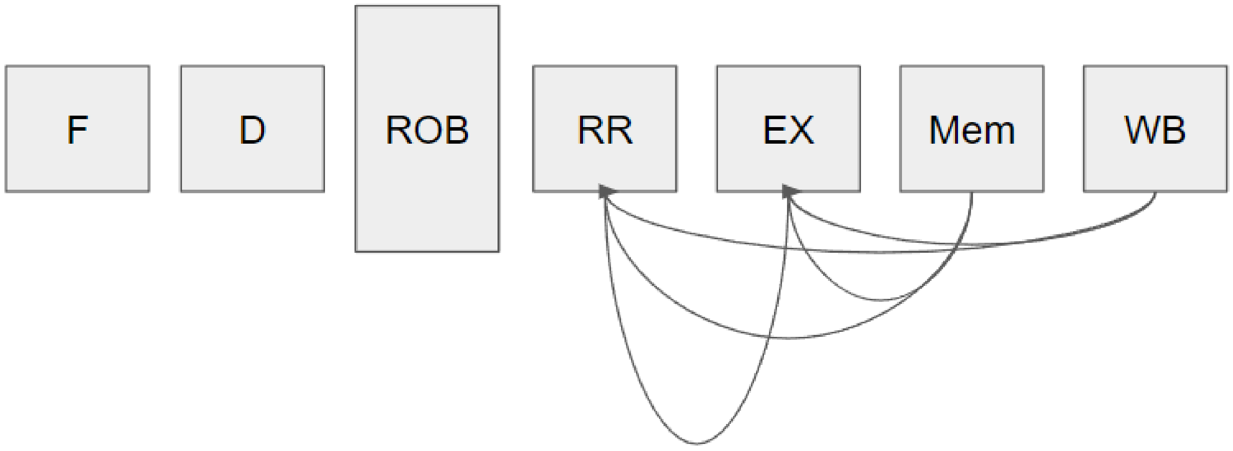
Nathan Ng
CE456
Prof. Nikos Hardavellas
Oct 1, 2022

# CE456 Project: Out-of-order Pinpoints

## Introduction

In a 6-stage RISC-V pipeline, a forwarding mechanism allows data to be forwarded from the Mem/WB/EX stage to the EX/RR (Register Read) stage. This removes the need for the IF/ID pipeline to stall while waiting for a register that has yet to be updated. Concurrently, instructions in the decode stage do not need to be read from register files, which incur a significant energy cost. This methodology can be further expanded into out-of-order processors; Out-of-order processors introduce a reorder buffer and an issue queue that allows processors to reorder instructions while providing an illusion to the programmer that his/her program executes in order. Implementing a reorder buffer allows the pipeline to take advantage of RAW hazards by moving dependent instructions closer. This allows the dependent instruction to take advantage of forwarding, which allows for data to be passed from the WB, Mem, and EX stages to the EX and RR stage.



Picture 1: Potential of Data Forwarding

This project will attempt to evaluate the significance of forwarding data to dependent instructions in a multi-stage RISC-V pipeline. This will be performed via Pin, a dynamic binary instrumentation tool for x86 ISA. Pin will be used to assess the frequency of RAW dependencies that do not use forwarding but could potentially be reordered to make use of it. If there is a significant number of potential forwarding in benchmark applications, the implementation of the pipeline can be performed.

## Similar Research

Past research, such as *The Energy Complexity of Register Files* (Zyuban, 1998), shows that each register read adds to the total energy cost of the system. While the paper suggests using a port priority selection technique to reduce energy costs, it targets the decode stage by modifying the read ports. Another research *Low-Complexity Reorder Buffer Architecture* (Kucuk, 2002) was introduced in 2002, which also aims to remove the need for read ports by introducing associatively-addressed retention latches to store data from previously executed instructions. Although the two previous research have the same motivation to reduce the energy cost of register reads, they do not make use of the reorder buffer to take advantage of forwarding by putting RAW hazards closer together.

**Front-end Implementation**

To allow for a large reorder buffer to detect and reorder dependent instructions, the front-end of the pipeline, which includes the fetch and decode stages needs to fill the reorder buffer to its capacity at a faster pace than the back-end, which includes the register read, execution, memory, and write-back stages.

**Reorder Buffer**

In order to facilitate potential forwarding, the reorder buffer needs to allow new instructions that are entering the buffer to detect if its operand instructions are ahead of it. Moreover, the distance between the new instruction and its operand instructions has to have a relatively large distance, where it does not already make use of forwarding. The main goal is to have an instruction's operand instructions at the Execution, Memory, or Writeback stage where it can directly forward data to the instruction in the Register Read or Execution stage. This can be achieved either by moving the new instruction to an index where it is one to three instructions away from its operand. In the case of three operand instructions ahead in the queue, all four instructions must be consecutively placed, whereby the operand instructions are in the Execution, Memory, and Writeback stage and the new instruction is in the Register Read stage.

The challenge in reordering instructions in the buffer lies in finding an effective algorithm that can detect a chain of dependent instructions within the reorder buffer. The buffer must ensure that no instructions between the new instruction and its operand instructions are dependent on the aforementioned operand instructions. Moreover, when operand instructions are available in the buffer and not located consecutively, dependency has to also be checked between them before the operand instructions can be fitted consecutively.

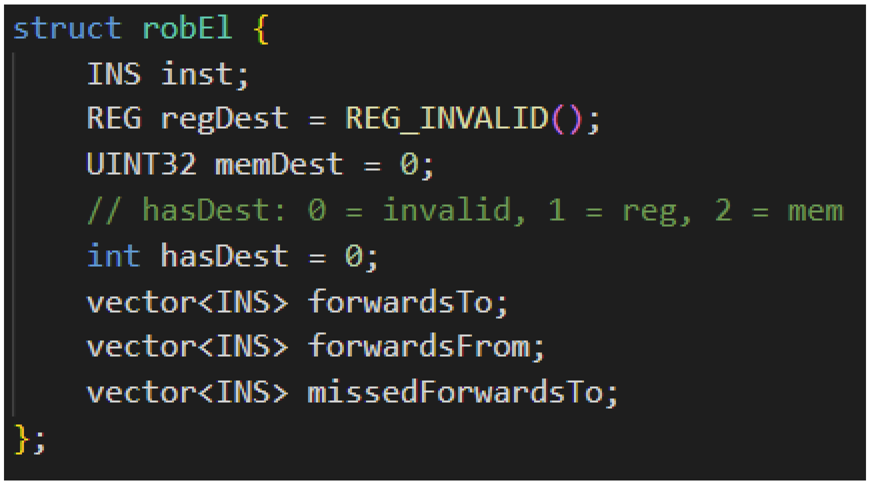
There are two methods of moving dependent instructions closer together; either by moving the new instruction forward or by moving the operand instructions downwards. For the current implementation, the former approach is predominantly used. The latter is only used if moving new instruction forward is not possible.

**Pin Implementation**

**Reorder Buffer**

In order to detect potential forwarding, a simulated reorder buffer is used in Pin. Pin tool, designed by Intel, allows developers to instrument applications using a C++ file.

The simulated reorder buffer (ROB) that was implemented consists of ROB elements. Each ROB element is a structure that contains the instruction that is added to the pipeline and some extra information.

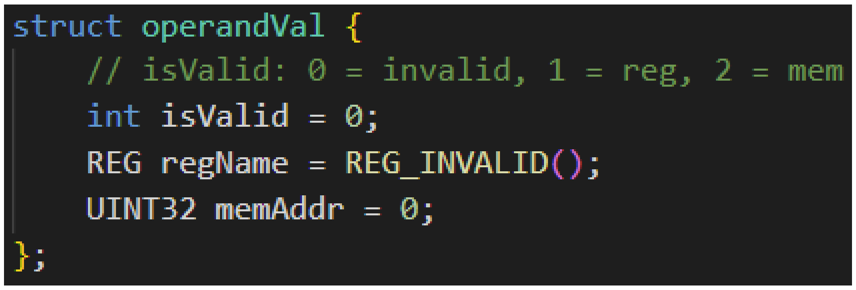


```
struct robEl {
    INS inst;
    REG regDest = REG_INVALID();
    UINT32 memDest = 0;
    // hasDest: 0 = invalid, 1 = reg, 2 = mem
    int hasDest = 0;
    vector<INS> forwardsTo;
    vector<INS> forwardsFrom;
    vector<INS> missedForwardsTo;
};
```

Picture 2: Structure of a ROB element

A ROB element consists of the instruction itself, and an integer: hasDest, which indicates if the instruction has forwarding potential and whether the data is derived from a register or memory. The structure also contains 2 vector of instructions that contain all of the operand instructions that have data being forwarded to and from it respectively. This allows us to monitor if the instruction can be ignored when moving new instructions around. Another vector of instruction is added to monitor the missed potential forwardings that could have happen.

**Checking for operands**



```
struct operandVal {
    // isValid: 0 = invalid, 1 = reg, 2 = mem
    int isValid = 0;
    REG regName = REG_INVALID();
    UINT32 memAddr = 0;
};
```

Picture 3: Structure of a operandVal

When an instruction with operands enters the tail of the reorder buffer, we can check if the operand is a register or memory address and store these data as a structure operandVal, which allows for the code further down to evaluate the operand's type and location. The operandVal structures are then stored in a vector, operandVals. This allows us to compare the operands with the destination value of instructions in the reorder buffer; allowing us to detect dependencies. With the new instruction's vector of operands, we can iterate through the reorder buffer and find all instructions that match the operands within the vector.

**Checking for dependencies**

After checking the operands and storing their type and register or memory address, we iterate through the reorder buffer and find potential instructions that can forward its data to the current instruction. When a reorder buffer instruction's destination matches the operand's register or memory address, we add it to a potentialForwardLocs vector. A prevPotentialForwardLocs vector also exist to mark where the potential forwarding instruction's previous destination is at in the reorder buffer. This is essential when the case of two operands have potential to get forwarded data. If the later forwarding instruction has a dependency on a previous instruction that occurs after the earlier forwarding instruction, then we can ignore the earlier forwarding instruction due to a RAW hazard when the later forwarding instruction is moved up.

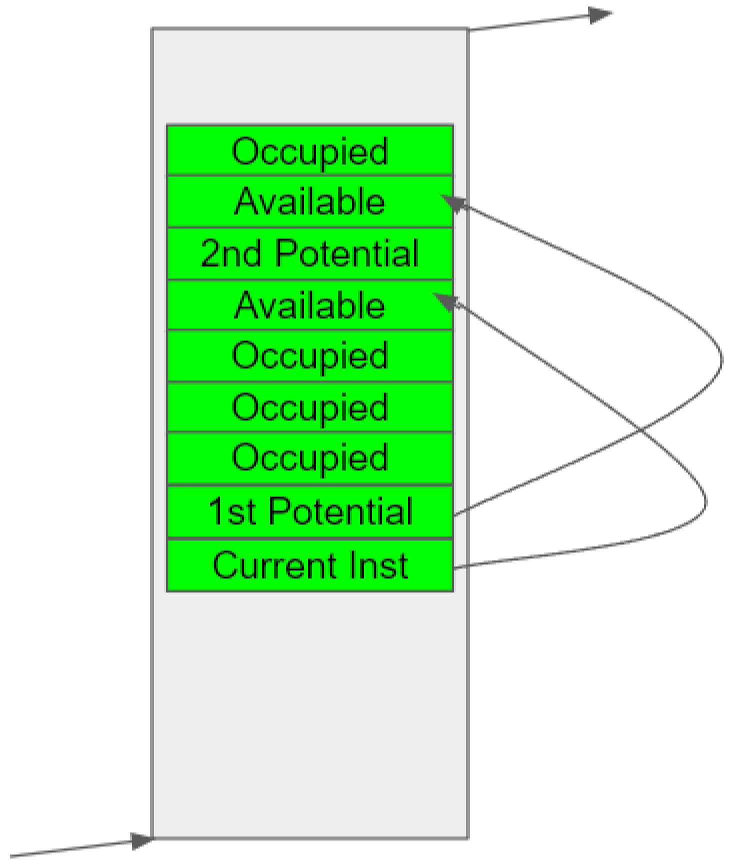
**Moving instructions**

Since the EX, MEM, and WB stages of the pipeline has the ability to forward data to both RR and EX stage, this means that an instruction can forward data one to three instructions away from itself.

In order to maximize the forwarding potential of the reorder buffer, we first sort our potentialForwardLoc vector in descending order, from tail to head of reorder buffer. This allows us to first attempt the move of the current instruction to its nearest dependent instruction, which maintains the RAW policy.

We look at the first potential forwarding instruction to check if there is still space to accommodate a forwarding. This is done by checking its forwardTo vector size and making sure it is less than or equal to 3. Otherwise, it has maximize its forwarding. If there is availability, we check if forwarding is already occurring by default. If not, then we need to check the next 3 instructions in the reorder buffer following this potential forwarding instruction; the check consists of checking if these instructions are getting forwarded data, while monitoring if they are forwarding data to any instructions. If an instruction have is not receiving forwarded data, we can mark the instruction's location as a potential spot. If an instruction is forwarding data to an instruction, we ignore any spot situated after it. For example, if we mark index 51 as a potential spot, and index 50 forwards data to 53, we cannot place the current instruction at 51. This is because doing so will make the instruction at index 53 move to index 54, loosing its forwarding. Once these checks are completed and a potential spot is found, we can move the current instruction to the potential spot's index. A boolean canStillForward is also used to indicate that the current instruction and first potential forwarding instruction have no other forwarding dependencies.

Next, we look at the second potential forwarding instruction if it exist. If forwarding is already being done without moving instructions, we can skip the move attempt. Otherwise, if the first forwarding is successful, we can attempt to do the second forwarding. The second forwarding does similar checks to the first forwarding, with the primary difference being that two indexes must be available to accommodate the current instruction and the first forwarding instruction. If two spots after the second forwarding instruction is available, we can move both the current instruction and first forwarding instruction to the spots. However, if only one spot is available, we can check if the instruction above the second forwarding instruction is forwarding to any other instructions. If it is not forwarding data, we can place the first forwarding instruction before the second forwarding instruction. An example is the picture below.



Picture 4: Moving close to 2nd potential instruction, edge case

If there exist a third potential forwarding instruction and the current instruction, the first potential instruction, and the second potential instruction do not forward to and from any other instructions, we can attempt to move the three instructions to the third potential forwarding instruction. This attempt can only be successful if the next three instructions following the third potential forwarding instruction do not get data forwarded to them.

**Build tool**

To build the c++ source file into a pin tool, the following steps should be performed.

1. Download pin3.13 and extract the pin folder.
2. set INTEL_JIT_PROFILER /{PIN_PATH}/intel64/lib/libpinjitprofiling.so
3. cd /{PIN_PATH}/source/tools
4. mkdir RobScan and copy contents into folder
5. make RobScan.test which will create an .so file in the obj-intel64 folder
6. ../../../pin -t obj-intel64/RobScan.so – ./matmul256
7. Returns a RobScan.out in current directory

If on Zythos, place all contents within a folder and run condor_submit robScanPinNormal.sub. This will return a RobScan.out in current directory.

## Evaluation

A test has been conducted on the Zythos cluster using Pin 3.13. The pin tool is executed with a matrix multiplication of size 256x256, and a reorder buffer size of 256. Since the implementation counts all forwarding regardless if it occurs by default, a baseline implementation is also executed to count the number of naturally occurring forwards. A baseline constant is added to the source file that can be triggered if baseline is desired.
The result of this execution is a follow:

| | Normal | Baseline | Actual |
|---|---|---|---|
| Forwarding Count | 6152428 | 5304128 | 848300 |
| Total Instruction Count | 22435060 | 22435022 | 22435060 |
| Potential Forwarding % | 27.4233 | 23.6422 | 3.7811 |

The pin tool has not been tested on benchmark applications. A larger buffer size should have a much larger impact on the forwarding capabilities but would require a larger memory capacity to hold the reorder buffer vector.

The tool should be attempted at a larger scale through the use of Zythos and SPEC benchmarks to better determine if the Pin Tool has issues that need to be solved.

## Unexplored implementations

The current implementation attempts to reorder the instructions throughout the reorder buffer. However, an alternative implementation exists where we focus on the issue queue and the immediate instructions that are being executed. By choosing the in-order instructions from the reorder buffer to be issued, we can potentially minimize complexity while still allowing for significant potential forwarding.

## References

Kucuk, G., Ponomarev, D., & Ghose, K. (2002). Low-complexity reorder buffer architecture. *Proceedings of the 16th International Conference on Supercomputing - ICS '02*. https://doi.org/10.1145/514191.514202

Zyuban, V., & Kogge, P. (1998). The Energy Complexity of Register Files. *Proceedings of the 1998 International Symposium on Low Power Electronics and Design - ISLPED '98*. https://doi.org/10.1145/280756.280943

University of Washington. (n.d.). *Forwarding - courses.cs.washington.edu*. Retrieved June 10, 2022, from https://courses.cs.washington.edu/courses/cse378/07au/lectures/L12-Forwarding.pdf