Massachusetts Institute of Technology

6.375 Complex Digital System

2007 Spring

# Re-Order Buffer for Superscalar SMIPSv2 Processor

Final Project

Group IV

Wei-Yin Chen wychen@mit.edu

Myong Hyon Cho mhcho@mit.edu

# Abstract

In this project, we designed and implemented an out-of-ordering superscalar SMIPSv2 processor. The key of this project is designing a re-order buffer which controls a lot of information to tell whether each instruction is ready to be executed, committed, or discarded. Also, we used multiple number of operation units for the processor, including an ALU unit, a branch resolution unit, and an address calculation unit for memory instructions, to improve the performance even further. To deal with various situations, this processor has to be far more complex than in-order processors. Furthermore, jumps/branches and memory loads/stores were especially difficult and needed to be considered very carefully. Using Bluespec was another key point for this project. It enabled for us to develop the processor in a high-level point of view and guarantee correctness by construct, but at the same time, we needed to understand Bluespec well to make sure it produces a hardware design that we wanted to implement. Specifically, attaining high rule concurrency was not easy, and keeping the critical path short was also far from trivial. We faced a number of challenges, but we ended up with a working processor that can speculatively execute all ALU instructions and memory address calculation out-of-order with the optimal rule concurrency possible with a single write-port register file. The re-ordering superscalar machine has as high performance as SMIPSv2 processor in Lab3 which solved the data dependency problem in another way. After exploiting wide pipeline superscalar architecture, it will excel the processor in Lab3.

Index

Index of Figures:

# 1. Project Description

The main goal of our team is designing a re-order buffer for an out-of-ordering superscalar SMIPSv2 processor in Bluespec. To achieve this goal, we needed to deal with the behavior of out-of-ordering machine, which challenged our skills to carefully design complex logical operations of digital devices. A superscalar architecture is another powerful way to improve the CPU performance, and we aimed to do a superscalar processor as well because out-of-ordering execution greatly handles superscalar machines with different delay times of each execution unit.

Obviously, the purposes of the out-of-ordering machine are increasing the performance of the processor in terms of efficiency (IPS), and exploiting the parallelism of applications. The re-ordering unit will interact with various modules in the processor such as decoding unit, free list, renaming table, register file, ALU, branch unit, memory unit as well as the instruction cache, managing complex controls over data flow and data dependencies. We used the unified physical register file as used in MIPS R10K, Alpha 21264 and Pentium IV for ALU operations. However, we creatively merged a re-ordering buffer without unified physical registers for memory instructions to solve problems efficiently.

We needed to orchestrate a number of rules that defined the behavior of the re-ordering unit and also implement correct and efficient interface between logical units so that the processor would gain the optimal performance. Also, we will design a set of testing scenarios which will show what kind of design choices affect the processor performance most significantly and how much improvement we can expect from introducing out-of-ordering superscalar machine to SMIPSv2 processor we have examined through the lab assignments.

# 2. Introduction to Re-Order Buffer and Superscalar Processor

## 2.1. Out-of-Ordering Execution

Figure 1 illustrates why out-of-ordering machines can expect higher bandwidth in executing programs. With an in-order processor, if an instruction has a long delay and the execution of the next instruction is dependent of its result then the whole execution should wait until that instruction is finished. However, if we can execute other instructions that are not dependent of the instruction with long delay then we can accelerate program execution.

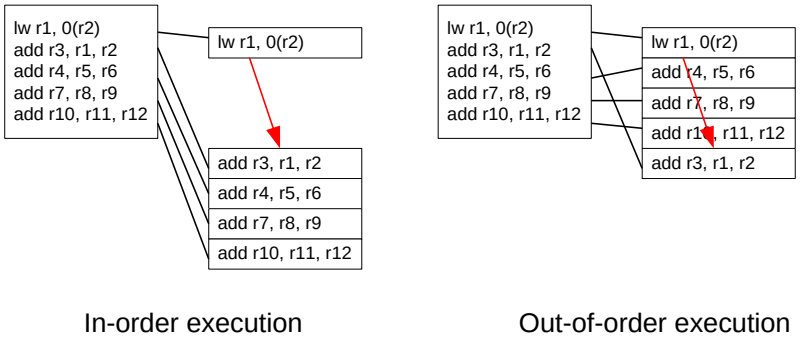


Figure 1 In-order execution vs. out-of-ordering execution

To implement this out-of-ordering execution, the processor must be able to 1) keep track of the status of multiple instructions, and 2) resolve dependencies between instructions so it could fire instructions which are ready to be executed. Moreover, it is important to handle branches and memory instructions because all speculative instructions need to be discarded if speculated wrong, even after they are already executed. Therefore, out-of-order execution should keep its in-order information to properly recover the previous states, which is illustrated in Figure 2. This brings much more complexity compared to conventional in-order processors.

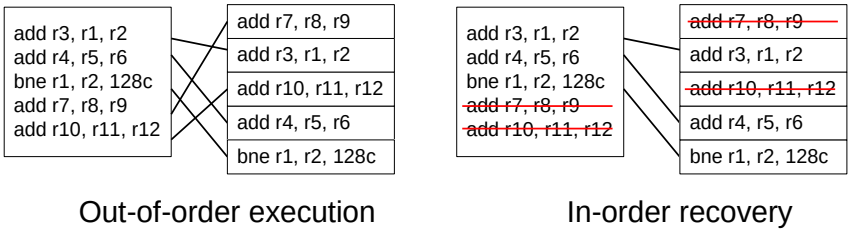


Figure 2 In-order recovery of out-of-ordering execution

## 2.2. Re-Order Buffer with Unified Physical Register File

Unified physical register is used in different commercial processors such as MIPS R10K, Alpha 21264, and Pentium IV to implement out-of-ordering execution. It keeps the information about dependencies between instructions by register renaming. Although architectural register names are continuously reused in original machine codes, all these names are renamed by processor into physical register names so dependencies are translated into the re-ordering buffer and the rename table, and solve the write-after-write hazard.

Out-of-ordering machines without unifies physical register file deals with dependencies by writing each calculated result onto re-ordering buffer itself. We choose to use unified physical register because we thought having a separate physical register file is the better way to utilize data storage than reserving storage size embedded in each re-order buffer entry.

# 3. High-level Processor Design

## 3.1. Life Cycles of Instructions in Re-order Buffer

### 3.1.1. ALU instructions

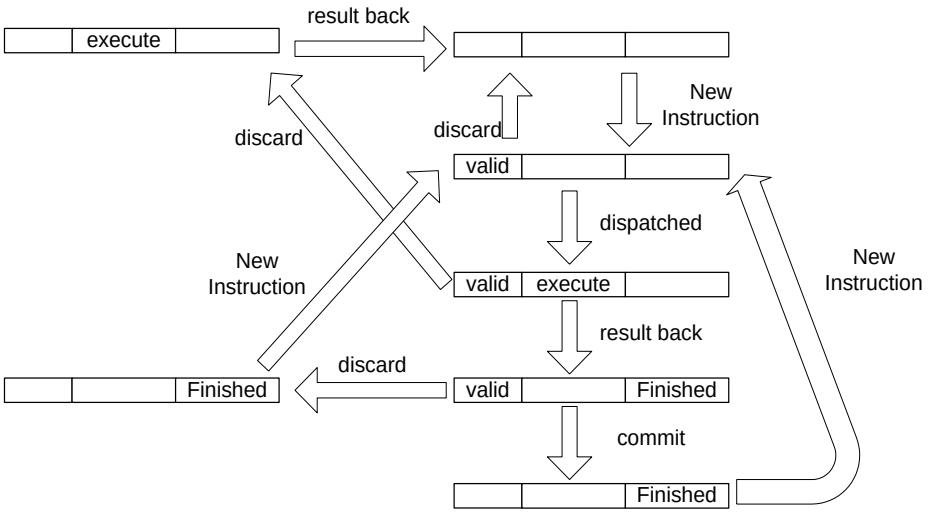


Figure 3 Life cycles of ALU instructions in re-order buffer

All instructions are fetched from the same fetching unit with in-order SMIPSv2 processor, but then inserted into the re-order buffer. ALU instructions only stay in the main re-order buffer. There is another special re-order buffer for memory instructions. We will refer to the main re-ordering buffer as ALU ROB, and the special re-ordering buffer for memory instructions as memory ROB. The newly inserted instruction enters ALU ROB with {valid} state, and it has bits showing whether the register containing source values for the operation is ready or not. If all sources are present, then it can be chosen by a decoder (out of order and oldest first), and its arguments are transferred to the ALU module, and its state becomes {valid, execution}. When ALU finishes its calculation and the result updates the buffer and the register file, then its state becomes {valid, finished}. When branch is resolved and this instruction is ready to be committed, it can be chosen by another decoder and committed, and then its state becomes {finished}.

The process discussed above assumes that eventually this instruction will be committed, but it may need to be discarded due to branch misprediction. When misprediction is detected when its state is {valid} or {valid, finished}, then just the {valid} bit is cleared and it won't be dispatched to ALU or committed because decoders check whether valid bit is set. However, we need to take care of {valid, execution} state; we cannot just discard this entry because the ALU result will return and the result should be discarded as well, or it could overwrite future entries. This is why it has {execution} state, and in case that it has {valid, execution} on misprediction, then {valid} is cleared and instructions with

{execution} states will remain in the buffer until their results come back from ALU and get discarded.

### 3.1.2. Branch/Jump instructions

Branch/jump instructions stay only in ALU ROB, and have similar life cycles with ALU instructions. The difference is that they will dispatch to separate branch/jump units because we have superscalar processor. However, all branch/jump instructions are designed to be executed in-order, because if speculative branches or jumps are executed then it needs to keep enormous amount of information to recover from misprediction, which we concluded to be inefficient.

To implement different dispatching algorithm, all branch/jump instructions have one more state {in-order}, since they first come into the buffer until they are freed from the buffer. Actually, any instructions that need to be done in-order and not to be executed speculatively can be dealt properly only by setting this state on, which is already used with mtc0 instruction.

### 3.1.3. Memory instructions

Because memory instructions may cause difficult dependency problems, sending memory requests to data cache is planned to be executed in-order like branches and jumps. However, to execute memory instruction the processor first needs to calculate its source or destination address from base registers and offsets. Although we concluded that enabling fully out-of-ordering memory operations is inefficient to implement, we found that we could do the address calculation parts out of order and it can improve the performance effectively with much less costs on hardware. Also, this implies that the processor will have an independent address calculation unit other than ALU so the address calculation can be re-ordered freely to improve performance.

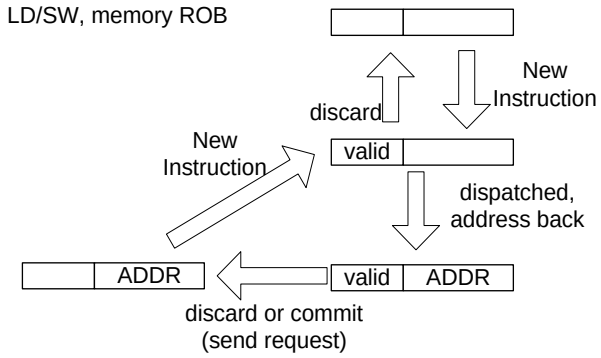


Figure 4 Life cycles of ALU instructions in re-order buffer

## 3.2. Branch Unit, Branch Resolution and Snapshots

This superscalar processor has an independent branch resolution unit so branch/jump instructions are dispatched separately. When there is no misprediction, it tells the re-order buffer so that instructions after this branch may be committed. When there are mispredictions, however, all speculative executions, including register renaming, free list, and present bits in the register file should be restored to the state before the branch.

To handle this problem, renaming table is snapshoot for each branch instruction, present bit is reset when the destination for a new instruction is renamed, and branch unit will inform a misprediction so the buffer can be recovered to the states before that branch.

Moreover, instructions in the buffers need to be discarded as well, and because they occupy physical registers for their results those registers are returned back to the free list when they are discarded. As we will see in the section Pipeline Stage3.4, the retire stage will pick an instruction in the buffers and check whether to commit or discard, and take corresponding actions.

## 3.3. Memory Address Unit

To implement out-of-ordering address calculations, we introduced another special re-ordering buffer for memory instructions, memory ROB, as stated in the section 3.1.1. Load instructions will stay both in ALU ROB and memory ROB, while Store instructions will stay only in memory ROB. This is because loads will write to a physical register: when a load instruction need to be committed, the physical register which the architectural register was previously renamed should be freed. And when the load instruction need to be discarded, the physical register reserved for the architectural register should be freed. However, this operation can be done without regarding to memory operation itself. Therefore, keeping the information in the main buffer and let its decoder logic handle this operation makes the whole design very simple and flexible. For store instructions, we don't need to care about this operation because they don't write to any physical registers. We will discuss about memory ROB more in detail in microarchitectural part, the section 4.3.5.

## 3.4. Pipeline Stages

As we have seen in the section 3.1, the superscalar units in this processor handle three kinds of instructions (ALU instructions, branch/jump instructions, and memory instructions) differently. Consequently, pipeline stages are separated according to the type of instructions. With the high-level point of view, each type of instructions goes through the following pipeline stages.

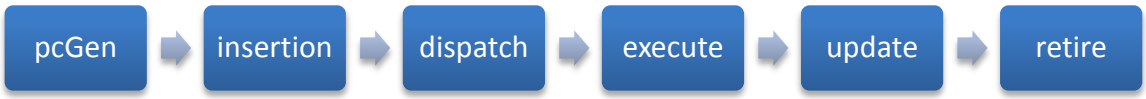


Figure 5 Pipeline stages for ALU instructions


Figure 6 Pipeline stages for branch/jump instructions


Figure 7 Pipeline stages for memory instructions

PC fetch stage is the same as in lab 3. In this stage, the next PC is generated from the branch predictor/BTB, and the request for the instruction of current PC is sent to I-cache. When the I-cache returns the instruction, it is decoded and inserted to the ROB at decode and insert stage. Depending on the type of the instruction, it will be inserted into the main buffer and/or the special memory buffer.

Now the pipeline is divided for different types of instructions. For memory instruction, it goes to address resolution. And then the calculated address is written back to the memory ROB, and at memory request stage the first one safe to commit generates real memory request. When the reply comes, the load update stage updates the register file and both ROBs. Note that we separate commitment of load instructions into two parts, and the commitment of load instruction in ALU ROB part will be taken care by ALU instructions' pipeline stage. Although this is not very clear here, it is much more efficient in implementing in hardware.

For ALU instructions, an instruction in ALU ROB goes into ALU dispatch and execution stage. Here the operand is read, and the instruction with its operands is dispatched to the execution module. After this stage, the instruction goes into execute stage. The operands are read in the dispatch stage, so the execution stage only handles the ALU related calculations without communicating with other modules. After an ALU operation is finished, the result is sent to result queue. ALU update is the next stage. In this stage, the result from execution stage is updated to the ROB and corresponding physical register. The present bits in the ROB are updated as well.

Branch instructions go through different pipeline stages as well, and the branch resolution handles all kinds of conditions resulting in discontinuous PC,

including branch instructions and J-type instructions. For these instructions, after the operand is read or decoded, the branch or jumping is resolved right away. And then, update stage will inform ALU ROB and memory ROB so they could commit or discard instructions after that branch/jump. Also, a correct PC target and a flag signal are sent back to the fetch stage and many modules need to discard the expired items in the following cycles.

For all kinds of instructions, the final stage is the retire stage. In this stage, an instruction is either committed or discarded. After all the results of previous branches are resolved the same as prediction, the result can be committed to the memory system and return the "last destination" register to the free list. If a branch is resolved wrong, then some part of the ROB will be discarded, and renaming table restore its previous snapshots. The free list would return to the state before the mispredicted branch when all the later instructions are discarded and all the destination registers are freed. However, store instructions take only memory related request at the retire stage, and loads instruction take both.

## 3.5. Data Dependency Loop

From the pipeline stages described in previous section, we can see the dependency caused by read-after-write is worsened than the in-order processor. In Lab 3, if we don't use the bypassing register file, and fire execution before the write back stage, then it has to stall for one cycle to resolve the immediate read-after-write dependency.

In our pipeline design, if all the instructions are dependent on the immediate previous instruction, the performance could be very low. For example, the first instruction is dispatched at the first cycle, executed at the second cycle, and write back to register file and update the ROB at the third cycle. Finally, the second instruction can be fired at the fourth cycle. So there are two bubble cycles in the pipeline, and the IPC can only be 1/3.

However, this kind of extreme serial dependency is not normal for practical programs, and the software critical dependency path is normally shorter than 1/3 of the length of total code. So this longer dependency loop should not be a dominant factor of the performance.

## 3.6. Architectural Summary of High-level Design

Figure 8 shows relations between execution units according to the high-level behavior described in this section.

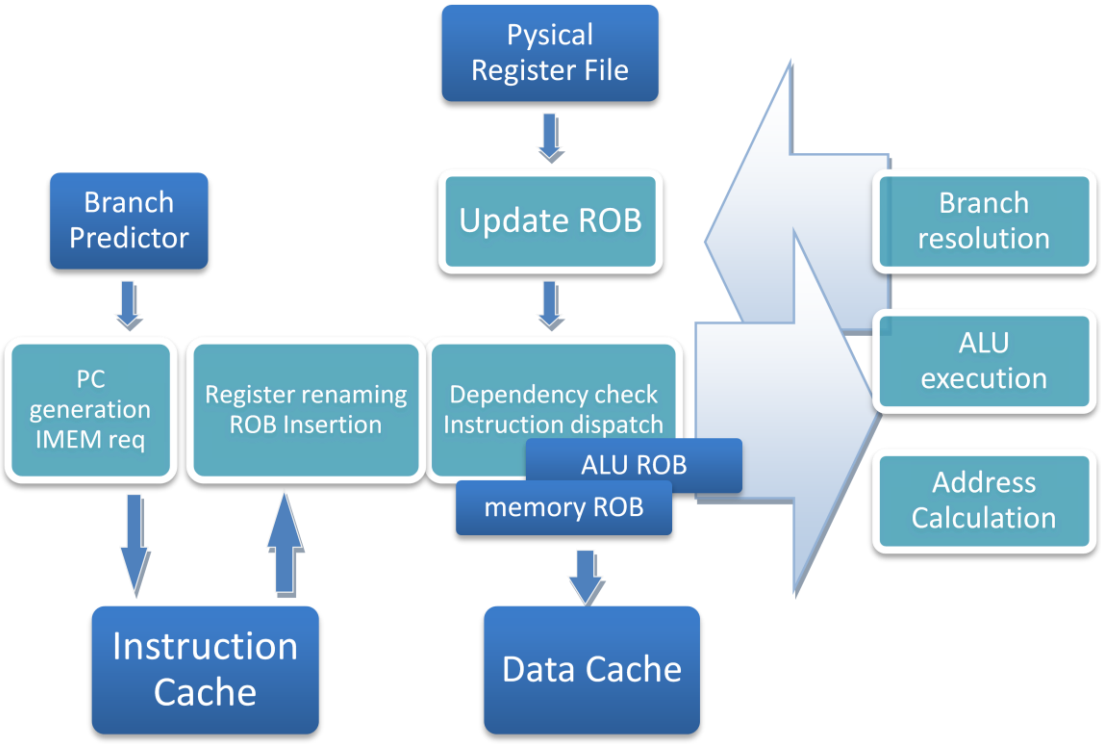


Figure 8 High-level description of out-of-ordering superscalar SMIPSv2 processor

# 4. Microarchitecture and Bluespec Implementation

## 4.1. Processor Module

Figure 9 in the next page shows detailed microarchitectural structure of the processor, regarding to Bluespec rules, modules and methods. The same high-level pipeline stage is often decomposed by several Bluespec rules to avoid those rules to be conflict by reading and writing into the same data structure at the same time. Some of these rules can fire at the same cycle and others cannot. We will provide more information about this in the section 5.

Here is the list of Bluespec rules in processor module and their descriptions.

`pcgen`

- Take predicted PC from branch predictor or increase the current PC by four, send requests to I-cache

`discardFetch`

- Dequeue instructions coming from I-cache in case of misprediction. Misprediction is determined by inspecting the tag from the I-cache and the current flag.

`decodeInsert`

- Decode instructions from I-cache and put into re-ordering buffers. Mutually exclusive with rule discardFetch

`dispatchALU`

- Decode ready ALU instructions, read the operands from the register file and dispatch to execution module

`branchResolve`

- Decode a ready branch or J-type instruction from ROB, read the operands and resolve the branch condition. Update the BTB and new PC

`branchStep2Link`

- For JAL instructions, write the current PC into register file and update ALU ROB.

`branchStep2`

- According to the branch resolution, tell ROBs whether the prediction was right or wrong

`aluUpdate`

- When the results of ALU instruction come from execution module, update the ROBs and register file.

`dispatchMem`

- Calculate address of ready memory instructions and update the calculated address in memory ROB

`memReq`

- Get the first ready memory operation in memory ROB. Send requests to D-cache

`memUpdate`

- When results of load instructions come from I-cache, update the ROBs and register file. If the register file only has one write-port, then the confliction with rule aluUpdate cannot be removed.

`memUpdateNOP`

- When results of store instructions come from I-cache, dequeue the response and do nothing else

`retireInst`

- Get retiring (commit or discard) instruction from ROBs, and put register index back to freeList

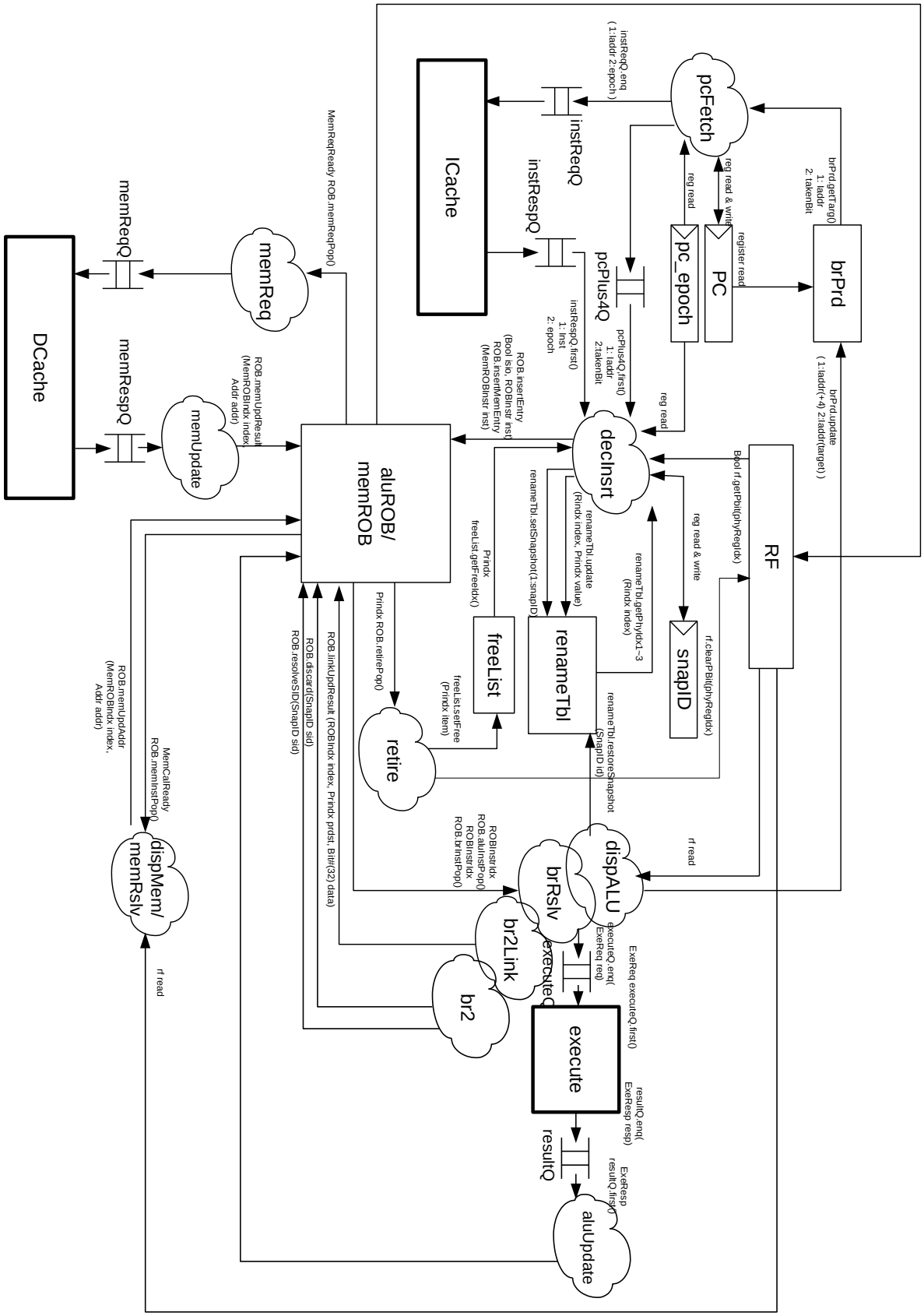


Figure 9 Microarchitectural structure of processor module

## 4.2. Re-ordering Module

The interface can be better described in Bluespec code. This is not the final version of the interface, just used to explain the concept, and different styles are mixed in the interface example, which will be described in the later parts.

```
interface ROB;
  method Action                    insertEntry     ( ROBInstr inst);
  method Action                    insertMemEntry  ( MemROBInstr inst );
  method ActionValue#(ROBInstrIdx) aluInstPop      ();
  method ActionValue#(ROBInstrIdx) brInstPop       ();
  method Bool                      getValidBit     ( ROBIndx index );
  method Action                    aluUpdResult    ( ROBIndx index, PRindx prdst, Bit#(32) data );
  method Action                    linkUpdResult   ( PRindx prdst, Bit#(32) data );
  method ActionValue#(ROBRetire)   retirePop       ();
  method ActionValue#(MemCalReady) memInstPop      ();
  method Action                    memUpdAddr      ( MemROBIndx index, Addr addr );
  method ActionValue#(MemReqReady) memReqPop       ();
  method Action                    discard         ( ROBIndx index );
  method Action                    resolve         ( ROBIndx index );
endinterface
```

This module contains two tables to store the reordered instructions. One is for memory related instructions, and another one is for all the other instructions and load instruction. They are described in section 4.3.4 and 4.3.5.

The functionality of these methods are described below:

`insertEntry/insertMemEntry`

- Insert an instruction to ALU/memory ROB.

`aluInstPop/brInstPop/memInstPop/memReqPop`

- Find out the first available ALU/branch/memory/memory instruction for execution/branch resolution/address calculation/memory request and label it as executing.

`getValidBit`

- Return the valid bit of an entry in ALU ROB. This is to check if the returned data from the execution stage can be updated to the register file.

`aluUpdResult`

- The present bits in ALU ROB are set if it matches the updated data and the entry that generate the updated data is valid. The base and source fields in memory ROB are updated as well.

`linkUpdResult`

- Similar with aluUpdResult, but the valid bit is not checked. It doesn't need to check the valid bit because this is only used by the linking instruction (JAL and JALR) and memory update, which are never speculative.

`retirePop`

- Return the instruction that can be retired, and label it as invalid. The head pointer is also incremented.

`memUpdAddr`

- Update the address field in the memory ROB

`discard`

- Label all the entries in ALU ROB and memory ROB as invalid if the snapid is "cyclically larger" than the snapid of mispredicted branch. The concept of cyclic comparison is to interpret the difference of the two values as signed value in the same bit width. For example, if the snapid is 3 bits wide, 4-7=-3, and 2-7=3. The possible number of snapids should be two times larger than the size of ALU ROB in order to guarantee the correctness of cyclic comparison.

`resolve`

- Label the corresponding branch instruction in the ALU ROB as finished.

Re-ordering module is supposed to have no rules, but there is only one rule to move the head pointers of memory ROB, because the discarding of memory ROB is not handled in the processor module.

`discardMem`

- When an instruction should be discarded from memory ROB, move pointers.

## 4.3. Leaf Modules

In this section, more detailed microarchitectural implementations of leaf modules are provided.

### 4.3.1. Free List

The interface can be better described in Bluespec code.

```
interface FreeList;
  method Action                       setFree      ( PRindx item );
  method ActionValue#(PRindx)         getFreeIdx   ();
endinterface
```

The state of free list can be represented in a 63-bit (assuming the size of physical register file is 64, and P0 is always 0) array. The query of a free index can be implemented by a priority decoder. The downside (or feature) of the priority decoder is the usage is not balanced. It can also be implemented by a shifted priority decoder (see section 4.3.6), and the shifting amount is updated by the chosen free index. As a result, the return value of the free list is evenly distributed and cyclic monotonic, and the newly returned item is not reused immediately. This property helps an optimization related to LW instruction.

As stated before, we don't have to use snapshots in the free list, as the discarded instructions returns the destination back to the list when they are discarded.

Furthermore, it is hard to use snapshots to restore the state because the branch can be resolved before the commitment of previous instructions. As a result, the snapshot is taken when the branch instruction is inserted, but we do want to keep the returned items by the previous instruction.

Here is a scenario that could cause this problem. An instruction is decoded and inserted. At the next cycle, a branch instruction is decoded and inserted, and the snapshot is taken. After that, the earlier instruction commits, and the last destination returns to the free list. Finally, the branch instruction is resolved and the result is mispredicted, so the snapshot is restored. However, the committed item of the previous instruction is gone.

### 4.3.2. Renaming Table

The interface can be better described in Bluespec code.

```
interface Rename;
  method Action update          ( Rindx index, PRindx value );
  method PRindx getPhyIndx1     ( Rindx index );
  method PRindx getPhyIndx2     ( Rindx index );
  method PRindx getPhyIndx3     ( Rindx index );
  method Action takeSnapshot    ( SnapID id );
  method Action restoreSnapshot ( SnapID id );
endinterface
```

The renaming table contains a register file with depth equal to the depth of architectural register file, and with width of $\log_2$(depth of physical register file). The set of snapshots duplicate the register file. The snapshot is accurate because renaming table is only modified in the decode-and-insert stage.

### 4.3.3. Physical Register File

The interface is defined as follows:

```
interface PRFile;
  method Action   wr( PRindx rindx, Bit#(32) data );
  method Bit#(32) rd1( PRindx rindx );
  method Bit#(32) rd2( PRindx rindx );
  method Bit#(32) rd3( PRindx rindx );
  method Bit#(32) rd4( PRindx rindx );
  method Bit#(32) rd5( PRindx rindx );
  method Bool      getPBit1( PRindx rindx );
  method Bool      getPBit2( PRindx rindx );
  method Action  clearPBit( PRindx rindx );
endinterface
```

The differences between this register file and that in lab3 are the size and present-bits. If not indicated otherwise, the size is 64 elements, with PR0 always stick to zero. The present bits are important for the decode-and-insert stage to look up whether the value of a register is present. The present-bit is set at the ALU-update stage along with the write to the register, and cleared at the commit stage. It doesn't need snapshots for the present bits because we can also clear it at decode-and-insert stage, so the wrongly set present-bit of destination would be erased whenever it is allocated again as the destination. As a result, speculative execution doesn't cause problems in the present-bits in register file.

### 4.3.4. ALU ROB

There are two pointers in the ALU ROB: head and tail. The entry pointed by the head pointer is the next to be committed or discarded, and the one pointed by the tail pointer is a place for the next insertion. There are also two Boolean variables: full and empty. These variables distinguish the state when the head pointer equals the tail pointer. They can also be used in the condition of methods, resulting in optimized critical path.

Each entry in the ROB contains several status bits: valid, execute, in-order, some present bits for operands, and finish. All the entries are initially invalid. After instruction insertion, it is valid. If an ALU or branch instruction pops, the execute bit is on. After the result of ALU returns or the branch is resolved, the execute bit is off and the finish bit is on. When handling the entry pointed by the head pointer, if the valid and finish bits are both on, then we can commit it. Or if it is invalid, then we can discard it. Otherwise just wait.

Regarding to atomicity, all the invalid entries are labeled at the same cycle when a branch is resolved. This guarantees the atomicity and the correctness. Even if there are some results on the fly, they will be discarded and not writing to the register file when coming back.

### 4.3.5. Memory ROB

Another important point of memory ROB is that it has actual value for base registers and source registers for store instructions, rather than physical register indexes and present bits. Because we have an independent address calculation unit, memory instructions do not have to act synchronously with ALU ROB, only except the fact that it should check whether the data for memory instructions will be still valid in physical register file when their executions begin. ALU ROB doesn't care about memory ROB, so the physical register that contains data for memory instructions may be freed before the memory instructions read it. To solve this problem, memory ROB has data field rather than physical register, so when the source value or base address is produced, memory ROB will take those values inside it so ALU ROB may not consider memory ROB in freeing physical registers.

As a result, only ALU ROB uses the unified physical register architecture. Memory ROB just put everything in the table. This also enables the possibility to do sophisticated dependency checking within the memory ROB.

### 4.3.6. Shifted Priority Decoder

A priority decoder outputs the first met item and returns the index of that item. The priority is static and can be monotonically increase or decrease with the index.

Shifted priority decoder is a variance of the priority decoder. The priority has a shifting amount, which is a parameter changeable in the run-time. This shifted priority decoder search for the first met item from the shifting amount on, and return the index. The priority is circular monotonically increasing or decreasing from the shifting amount.

For example, if we have an array v[n], and the shifting amount is k, then if v[k] is met, the decoder returns k, otherwise, it searches v[k+1], v[k+2], and so on, and after searching beyond v[n-1], it returns to search v[0], and ends on v[k-1].

The hardware is combining a vector rotation module, a normal priority decoder, and an adder with rounding control. First module rotates the input vector by the shifting amount, so the item with highest priority is rotated to the first element. The output of the priority decoder is added with the shifting amount, but if the sum is greater than the size of the input vector, the answer is subtracted by the size. The last step is equivalent to modulation, but this is cheaper in hardware.

This module is used to find the first available instruction in the reorder buffer, and the first free index in the free list.

# 5. Obtaining High Rule Concurrency

## 5.1. Calling Relationship

The top level module is the processor module, and all the actions are described as rules, which are described in 4.1. These rules call many leaf modules, and ROB is the largest and most complex leaf module, and actually causes most of the conflictions. The calling relationship is shown below. As a result, solving the method confliction in the ROB becomes the first task.

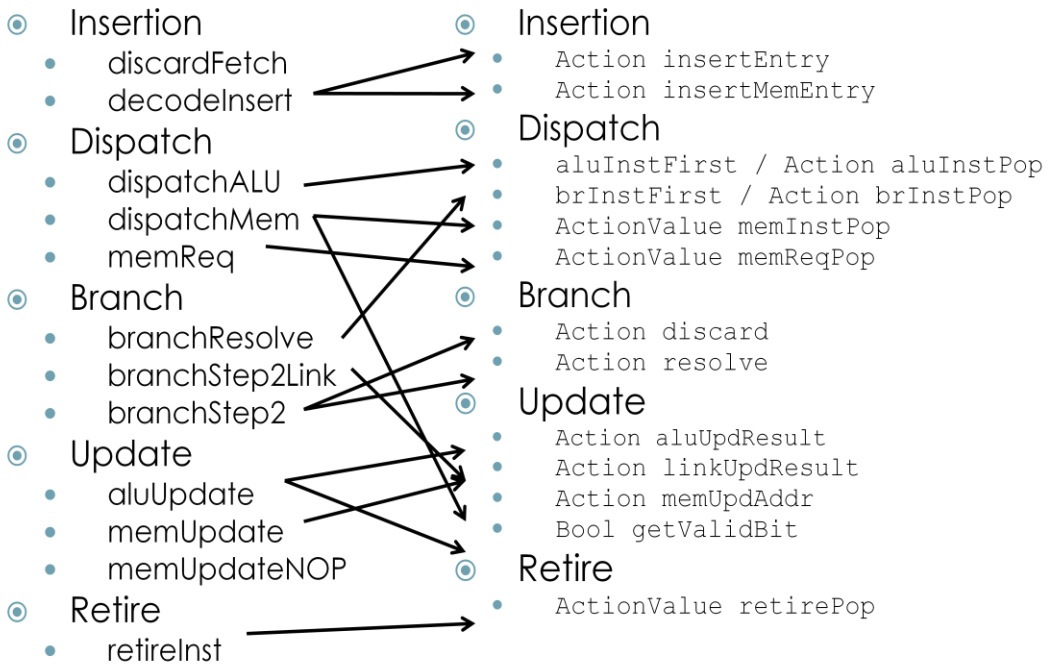


Figure 10 Calling relationship between processor and ROB modules

## 5.2. Read-Write Pattern

The reason of having rule confliction is the incompatible read-write patterns in different rules. If the read set and write set are analyzed as incompatible, the compiler would not fire these two rules at the same time even if they are both able to fire. As a result, the rule concurrency is low and the performance is bad.

To have high rule concurrency, the first step is to manually analyze the read-write pattern of all the conflicting rules and methods, and see what we can do to improve the confliction.

### 5.2.1. Read-Write Pattern in ROB

Most of the rules in the processor call the methods in the ROB, and the method confliction in ROB is the main cause of rule confliction in the top level.

In the original design of the ROB module, the ROB buffer and memory ROB buffer are contained in two huge registers. The type of these registers are registers of vector of structure, `Reg#(Vector#(ROBSize, ROBEntry))` if expressed in Bluespec syntax. Since the confliction analysis is done on the basis of register, any write to each portion of the buffer would be considered as a write to the register, hence results in the confliction. This property makes the confliction analysis worse because even if two rules write to different fields in the table, the compiler would regard them as conflicting.

Furthermore, writing in this style also makes compiling extremely slow. With ROB size 4, the compilation time is around half an hour; with ROB size 7, it takes 3 hours, and ROB size 8 can never finish, which is very unreasonable.

### 5.2.2. Read-Write Pattern in the Physical Register File

In the register file, all the reading methods happen before all the writing methods, and they are sequentially composable. However, we have two writing methods: one is `wr()`, and the other is `clearPBits()`, which are conflicting.

As the register file is separately synthesized, the number of ports are fixed in the bit level, so the all the rules that call `wr()` would also conflict with each other. Even if we don't separately synthesize the register file, we also have to use some other methods to explicitly make some sequentially composable writing ports for different rules to call.

### 5.2.3. Multiple Write Problem

One of the common problems in the confliction analysis is multiple write. Since write/write and update (read then write)/update pairs are not sequentially composable but widely used in the design, we have to use some structures that support multiple writes, or we cannot get high concurrency.

## 5.3. Methodologies

### 5.3.1. Structural Coding Style vs. EHR

The traditional method to solve the confliction is to write structural code. Structural code is a coding style that explicitly takes care of lower level details in the hardware structure.

One possibility is to use `RWires` to get all the written data, and use a rule that is fired in every cycle to handle all the possible combinations of the written data. Therefore, the methods become confliction free, and the register is only updated in one rule. This is a trick to lie to the compiler that these methods are confliction free. The drawback is the difficulty to guarantee the correctness and worse readability of the code.

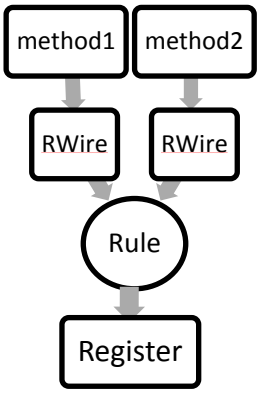


Figure 11 Using RWires to solve confliction

The other possibility is write a rule for each storage element, thus the register is updated only by one rule. However, this method is sometimes even harder to implement because the original rule splitting is based on concept grouping instead of writing pattern.

Since both of the solutions involve large change in the code or even the organization of rules and methods, we decided to extensively use `EHRs` and use the methodology similar to what is covered by the lecture to solve our problems and make few changes to the method/rule structures.

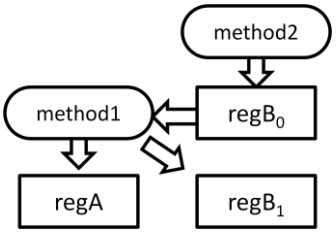


Figure 12 Using EHR to solve confliction

However, the methods/rules become sequentially composable instead of being confliction free, and there is an ordering constraint on them. Therefore, if two rules call different methods that have inconsistent ordering, these rules are still conflicting.

In order to avoid the inconsistent ordering, we need to assign a global ordering of all the top level rules, and propagate the ordering to all the leaf modules, and change their EHR index accordingly. Notice that the EHR index within a rule/method should be kept consistent, even in the condition, or the atomicity is broken.

### 5.3.2. Field Splitting

The main cause of method confliction in ROB module is the misuse of the register. We split the structure according to the read-write analysis of different fields. For those fields who needs to be read in every entries or written for multiple times, they are put in the `EHR` with type `EHR#(n, Vector#(ROBSize, ROBBookkeep)`, where `ROBBookkeep` is a structure containing all the bookkeeping status bits. For those fields who is only read in limited number of entries and written in only one rule, it can be fit in the built-in structure `RegFile`.

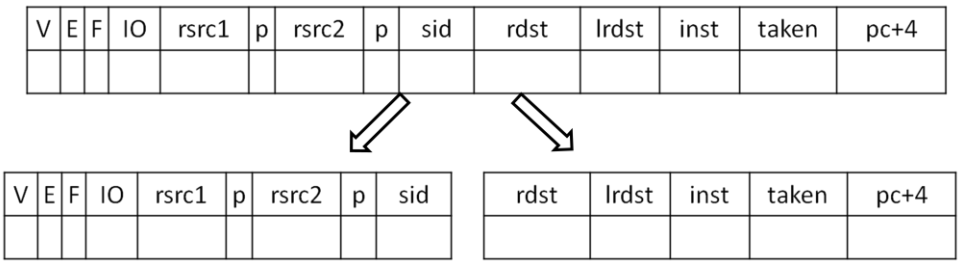


Figure 13 Field splitting in ROB module

After doing this, the compile time also improved greatly, and ROB size 8 can be compiled in half an hour, and size16 is done within 2 hours.

### 5.3.3. Critical Path Consideration

Using EHR to solve the confliction is convenient, but it usually results in larger area and longer critical path.

The additional area comes from two places. One is the multiplexers in the EHR itself, and another is that the customized combinational logic may be duplicated because the EHR index of the input might be different. The longer critical path comes from the combinational path introduced by the EHR and more gate delays caused by the multiplexers in front of the writing port.

In order to get high concurrency and similar critical path at the same time, we utilize the unsafe EHR hacks and use some domain knowledge to guarantee the correctness.

### 5.3.4. Safe Non-Coherent EHR Index

The safe type of non-coherent EHR index usages is to read values early in the condition and write late in the body. Moreover, the data written by the other rules cannot make the condition of this rule come from false to true, thus it doesn't affect the atomicity in this rule. This is relatively simple to verify if these conditions only contains Boolean types and the written data are constants.

If these given conditions are met, the correctness will also be guaranteed, and it only influences the performance. The given condition should be able to be

verified or analyzed by the compiler, but the current version of bsc doesn't do this.

After doing this optimization, the combinational path is largely decreased.

### 5.3.5. Unsafe Non-Coherent EHR Index

On the other hand, we also use some non-coherent EHR index that is generally unsafe. The correctness is guaranteed by the domain knowledge of the protocol we used in the design, and this kind of operation is too hard for the compiler to analyze and optimize.

For example, in the renaming table, the snapshot can be taken and restored at the same cycle, and the restoration happens after the taking action in the method ordering. However, from the domain knowledge, we know that the copies taken and restored cannot be of the same epoch, so we don't have to search for the updated snapshots in the restoration method, thus decrease the combinational path. This kind of optimization is error prone, and huge effort is needed to analyze the interaction between methods and rules.

## 5.4. Results

### 5.4.1. Remained Confliction

The only remained conflictions in the processor are rule aluUpdate, memUpdate, and branchStep2Link, as all of them write to the register file, and the register file only has one writing port. Consequently, the method being called by these three rules, methods aluUpdResult, and linkUpdResult in ROB module can be conflicting without affecting the overall performance. Keeping the confliction of these two methods also decreases the length of the combinational path.

This results in the optimal concurrency; given the system only has one-writing-port register file.

### 5.4.2. Critical Path

The physical numbers are compared with the low concurrency version of the design.

After synthesis, the critical path only increases by 2%, and the area only increases by 8%. So we can say that we achieved high concurrency while keep the physical overhead low.

# 6. Design Exploration and Evaluation

## 6.1. Exploration Dimension

### 6.1.1. The Size of ROB

The size of ROB determines how far the processor can re-order instructions. If a program has a lot of parallel computations and hence high ILP (instruction level parallelism), then it may have instructions independent of previous instructions. In order to get better performance, the size of ROB should be larger than the length of the pipeline, or at least larger than the length of back-to-back dependency loop. However, if ROB is too large then the penalty of misprediction also grows because more instructions need to be discarded if the branch instruction is resolved late.

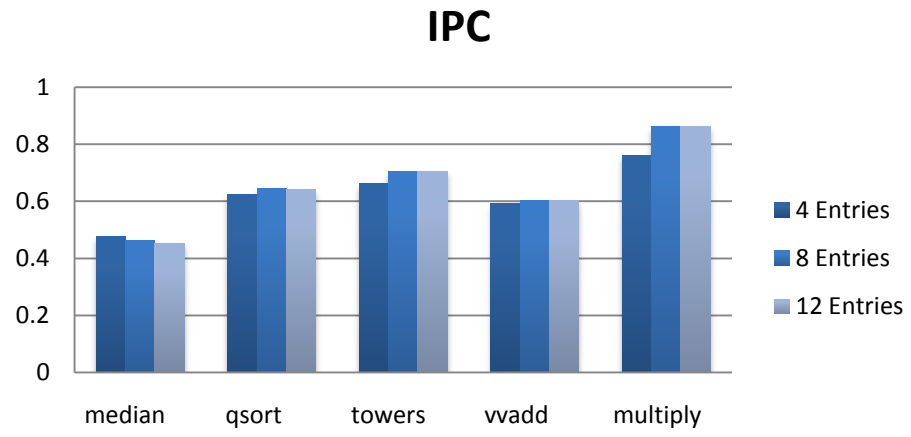


Figure 14 IPC of different ROB sizes

We took ROBs with the size of 4, 8 and 12. As the result shows, we cannot expect any benefits by increasing ROB size from 8 to 12 unless we have a sophisticated BTB module and bulk retirement architecture. In our final design, the size of ROB was chosen to be 8.

### 6.1.2. Adjusting Pipeline Stages

After finishing the first working version with high concurrency, we tried to revise the microarchitecture of our processor for possible improvements. Merging execute stage with update stage was successful, because it shortened end-to-end dependencies from 3 cycles to 2 cycles. The trade-off expected was a longer critical path, but post-synthesis results showed that the critical path is only 1% longer than the previous version, while IPC is increased by 5.8%, hence

the IPS is improved. The implementation of this change was as simple as just changing a FIFO to be a Bypass FIFO because we already decoupled the execution stage by FIFOs.

## 6.2. Evaluation Results

### 6.2.1. Hardware cost

A. Critical Path

| | |
|---|---|
| Post-synthesis | 4.44ns |
| Post-P&R | 9.40ns |

The critical path is on the branch resolution stage. In terms of post-synthesis timing, at 1.88ns, a branch instruction is dispatched from ROB so we can see that dispatching logic (shifted priority decoder) takes almost half the clock cycle. At 2.86ns, data is read from the register file and finally at 4.24ns the PC register is updated to be a new value.

The post-P&R critical path is almost twice as long as in Lab 3, as the critical path of Lab 3 is between 5 to 6ns. The delay of the shifted priority decode, hence the delay of ROB is inevitable for out-of-order architecture. In order to get similar critical path, the pipeline stage should be more fine-grained than in-order processors like in Lab 3.

B. Area Analysis

The total area is reported to be 1.42 $mm^2$ after place and route. Each hardware component contributes to the area as the following chart.

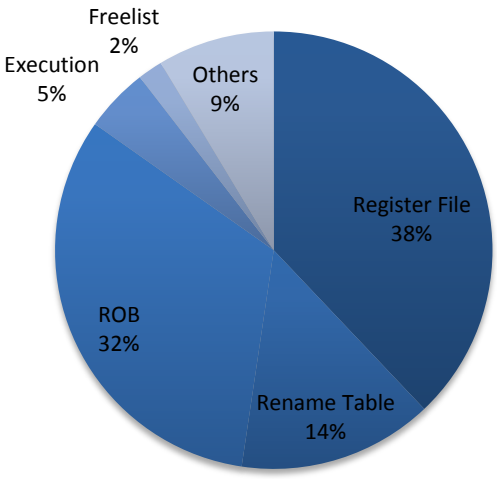


Figure 15 Module level area break down after place-and-route

Register file takes similar portion of the whole area with the one in Lab 3 processor while ROB takes the second large portion. ROB is mostly consists with data paths instead of flip-flops. This result is from the unified physical register file system, and if we used ROB having data field instead, then ROB size would have been dominated and the total area could be even larger. The renaming table is huge because it contains several copies of snapshots, so the flip-flops take a huge area.

### 6.2.1. Application Performance

The IPC results are compared between two variances of Lab 3 processor and three variances of out-of-ordering processor.

| | |
|---|---|
| Case 1 | LAB3 – low profile version<br>wbQ size 2, no bypassing register file |
| Case 2 | LAB3 – high profile version<br>wbQ size 8, bypassing register file, decoupled wbQ and memory |
| Case 3 | OoO Superscalar – non-concurrent version |
| Case 4 | OoO Superscalar – initial version<br>ROB size of 8, execution and update are separate |
| Case 5 | OoO Superscalar – merged pipeline stages<br>ROB size of 8, execution and update are merged |

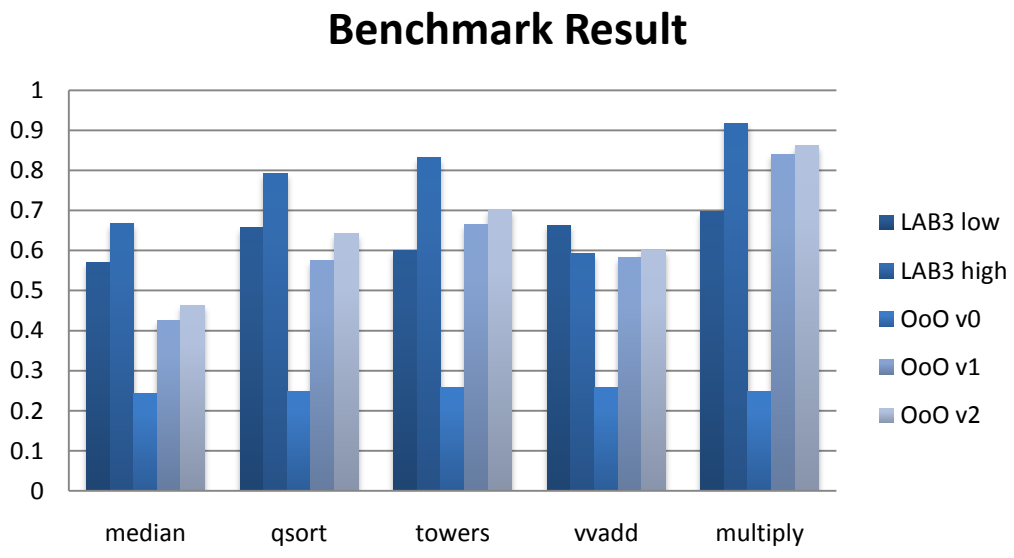


Figure 16 IPC results of different architectures

The version without high concurrency is expected to be slower than the same 'no concurrency' version of Lab 3 processors because we have deeper pipeline than Lab 3. For towers and multiply, the OoO processors were faster than low

profile version of LAB3 processors, but overall Lab 3 processors are faster. The biggest reason is that, in fact, the Lab 3 processor does not suffer much from data dependency. Because SMIPSv2 has a very simple ALU so all instructions other than memory instructions have no significant delays. And Lab 3 processor decouples writeback and execution by using writeback Queue so it can 'fill' the bubbles introduced by memory instructions. Furthermore, using bypassing register file erases the bubble cycles caused by back-to-back dependency completely. If the program can be perfectly predicted by using BTB, and there are few memory operations, the IPC of lab 3 can be very close to 1, like the multiply benchmark running on high profile lab 3 processor.

The real bottleneck of Lab 3 processor is the width of pipeline. In order to have high performance on a wide pipeline superscalar processor, out-of-order execution is even more important.

This out-of-ordering superscalar processor can achieve IPC larger than 1 if it can fetch and retire multiple instructions at one cycle, which will fully exploit the superscalar architecture. In particular, when the processor has complex execution units such as multipliers and floating point units, then out-of-ordering execution will become far more important to utilize multiple execution units with different delays. Moreover, this architecture can overcome any size of memory latency with larger ROBs. Therefore, we consider the current result is good enough to show the potential of out-of-ordering machines.

## 6.3. Final Physical Optimization

After we had a high concurrency version and gave the presentation, we had some feedback and we tried to make the critical path as short as an in-order processor. This is another direction of optimization. In this scope, the IPC is less important than having a shorter critical path.

### 6.3.1. Methodology

As this is the physical optimization, and some of the characteristics of the physical device cannot be estimated accurately without actually synthesizing and routing, the methodology is to have some intuition about how slow each action is by synthesizing various versions of the processor with different task segmentation. After that, we can heuristically determine the optimal task segmentation by manual analysis.

There is an optimization loop because we also need to optimize slightly lower than microarchitecture level when some single task is the critical path. In order to speed up, we only rely on the synthesized result as the physical feedback.

To sum up, the first step is to cut the tasks into more stages by using FIFOs and try to find the slowest tasks and optimize them in circuit level. After

optimizing all the targets, we put some stages back together if the total delay of the stages is not longer than the critical path.

### 6.3.2. The Journey of Physical Optimization

In the beginning, our critical path is at the branch resolution, and it is 4.44ns.

After cutting all the tasks into tiny stages, instruction read out from the ROB, reading the register file, execution/branch resolution/address calculation, and update are all separate stages.

Then the critical path becomes around 4ns and it is from robbook to robbook, which is the EHR register containing the state bits. After optimizing that, the critical path is around 3.5ns and it is through the shifted priority decoder in the free list. After optimizing that, the critical path is lowered to 3.2ns, from memrob to memrob, which contains the ROB table of memory instructions, and then we have to focus on the report writing.

The detail of how we optimized the EHR structure and shifted priority decoder is described below.

### 6.3.3. EHR Simplification Fallacy

The reading port of an EHR is dependent on the enable signal of all the "previous" writing port. If the EHR is not separately synthesized, bsc can optimize the unused reading and writing port away, resulting in fewer multiplexers in front of the final real writing port.

Here is a simplified model of how we used the EHR. Assuming we have an EHR register called `v`, which contains a structure of field `A` and `B`, and `v[i]._read()/._write()` is the $i^{th}$ reading or writing port. We write to `v[0].A` in some rule, and read `v[1].B` in another. This generates an ordering, but since we already have a global ordering, this is acceptable. According to the read-write pattern, the value `v[1].B` is dependent on the enable signal of `v[1]`, although `v[0].B` is never updated.

The fallacy here is that we assumed the synthesizer can use some gate level optimization techniques to erase the constant writing values from `v[0].B` to `v[0].B`. In fact, it cannot be removed. As a result, we have fake dependency circuit in after synthesizing.

In order to optimize this, we put each fields to different EHRs. So the fake dependency is removed, and as bsc can erase the unused EHR index, we can keep using the old EHR index without worrying about if the hardware is larger.

### 6.3.4. Optimizing Shifted Priority Decoder

The original design of shifted priority decoder is the combination of a vector rotator, a normal priority decoder, and an adder with overflow control. The critical path can still be improved.

We use a new priority decoder that receive an answer vector, and return the element in the vector with the same index as selected item. The timing and area should be the same as the old priority decoder if the answer vector is a constant.

Now we make an answer vector containing the index, and use the vector rotator to rotate both the Boolean input vector and this answer vector, and feed them to the new priority decoder. The output of the new priority decoder would be just the correct answer.

After this optimization, the last stage of the adder with overflow control is eliminated, and the critical path is shortened. However, the area is larger because we use more rotators.

Compared with a Boolean vector rotator, we cannot just transform the answer array to a bit array and use a normal shift operation with shifting amount multiplied by the width because the width of an entry in the vector can be larger than one, resulting in wider shifting circuit and a multiplier.

Instead, we transpose the answer array, and chop each bit in the answer to a separate bit array. So the number of shifters is the same as the width of the element. We are so proud to write such a smart Bluespec hack so we include the code in the report. The provisos problem is discuss in section 8.5.

```
function Vector#(size, Bit#(1)) rotateN_1( Bit#(TLog#(size)) n, Vector#(size, Bit#(1)) v);
  let shifted = pack(append(v,v)) >> n;
  Bit#(size) truncated = shifted[valueof(TSub#(size,1)):0];
  return unpack(truncated);
endfunction

function Vector#(size, Bit#(width)) rotateN_wide( Bit#(TLog#(size)) n, Vector#(size, Bit#(width)) v)
    provisos (
      Bits#(Vector#(size, Vector#(width, Bit#(1))), TMul#(width,size))
      ,Bits#(Vector#(size, Bit#(width)), TMul#(width,size))
    );
  Vector#(size, Vector#(width, Bit#(1))) vv = unpack(pack(v));
  Vector#(width, Vector#(size, Bit#(1))) trans = transpose(vv);
  Vector#(width, Vector#(size, Bit#(1))) trans_shifted = zipWith(rotateN_1, replicate(n), trans);
  Vector#(size, Vector#(width, Bit#(1))) ansv = transpose(trans_shifted);
  return unpack(pack(ansv));
endfunction
```

### 6.3.5. Result of Physical Optimization

Although this physical optimization was started after the presentation, and we have only a few days to try it out, the result is promising.

As stated before, the critical path after synthesize decreased from 4.44ns to 3.2ns, a 28% reduction, and the average IPC is lowered by 5%. The total gain of IPS is positive. We have to point out that the post-synthesize timing is similar to the in-order processor in lab 3 now, despite of all the ROB overheads.

# 7. Conclusion and Future Work

During this project, we have solved a number of difficult problems in designing complex digital logic. It was so challenging to carefully design and verify the logic of the complex processor and find a good way to implement it in Bluespec. Above all, we dealt with speculative execution of instructions after branch and out-of-ordering memory calculation of memory instructions which were not easy but helped to improve processor performance.

| Out-of-order execution |
| --- |
| • ALU instructions and address calculations are speculative and out-of-order.<br>• Branch resolutions and memory requests are in order. |
| **Superscalar architecture** |
| • ALU execution, branch resolution, address calculation and memory request can be dealt simultaneously. |
| **Optimal concurrency** |
| • The highest possible rule concurrency with single write-port register file and renaming table.<br>• ROB compensates memory latency. |

The first possible follow-ups is implementing multiple instruction fetch and retire in order to exploit the superscalar architecture, which will results in better performance than Lab 3 processor. Furthermore, introducing more execution units, especially complex ones such as multipliers and floating point units will reveal the true benefits of out-of-ordering machine. We are also interested in implementing more features of modern processors including accurate exception handling, because it must be a good experience to design and implement more complex processors which are more similar to commercial processors. Some deeper design explorations of out-of-ordering superscalar SMIPSv2 will be interesting because we believe that this relatively simple processor compared to other commercial processor will reveal how microarchitectural changes can affect to the processor performance more clearly.

# 8. Appendix – Work on Bluespec / Bluespec Compiler

During this project, we have faced different problems working with Bluespec compiler and took some time to solve those problems. We think that summarizing those problems will be helpful for those who study Bluespec and especially those who do similar projects in the future.

## 8.1. No automatic multiplexing when the called module is synthesized separately

### 8.1.1. Symptom

This could be a known bsc bug, because we faced it since lab2. When a module is synthesized separately with compiler directive (* synthesize *), the interface is fixed in bit-level, thus is the number of allowed concurrent methods. However, even if we manage to fix the number of calling methods, the compiler would not generate a multiplexer in front of the input to that particular method. Instead, it complains about multiple usage of the method, although these multiple usages are mutually exclusive and could be multiplexed into only one output.

### 8.1.2. Solution

One way to work around is to manually write a MUX for the method, but this is generally tedious. Another way is not to synthesize the called module separately, but we cannot analyze the area of the module, and the modular boundary is invisible in the lower level. So this is not actually solved.

## 8.2. Unusable constant in static elaboration stage

### 8.2.1. Symptom

Some constants available in the static elaboration stage can be used in the declaration but cannot be used in the loop-test expression in for-loops.

### 8.2.2. Solution

We use compiler directive `define to declare a macro with the same name, and use the macro wherever the typedef version cannot be used. We also use some provisos to make sure the `define version and the typedef version are equal. After understanding the difference between numerical type expression and ordinary static value, we eliminated all the redundant `define macros and use pseudo-function valueof() to convert the constants.

## 8.3. Run-time system uses up huge memory

### 8.3.1. Symptom

Sometimes when some of the parameters are adjusted to a larger value, the compiler would report huge memory usage of the Haskell run-time system, and just stops.

### 8.3.2. Solution

Increasing the limitation of the run-time system memory usage by adding "+RTS -K320M" to the command line would solve this problem. Fortunately the physical memory of the remote vlsifarm workstations is barely enough.

## 8.4. Needing explicit hardware decomposition

### 8.4.1. Symptom

When we implement the priority decoder, in some cases the compiler generates an extremely long error message in the elaboration stage and stops; in the other cases, the generated hardware is far less efficient than expected.

An example code can better illustrate the situation. The following code segment is a function generating the first 0-bit in bit array "used[]".

```
function Maybe#(PRindx) firstFree();
  Maybe#(PRindx) ans = Invalid;
  for(Integer i=PRSize-1; i>=0;i=i-1) begin
    if( used[i] == 1'b0 )
      ans = Valid (fromInteger(i));
  end
  return ans;
endfunction
```

The generated Verilog code is not exactly a simple priority decoder with inverted input, although it should be.

### 8.4.2. Solution

If we decompose the logic to the Boolean function part and the priority decoder part, then the compiler works fine.

Take the example again. If we transform the condition "used[i] == 1'b0" to a Boolean function, map the bit array "used[]" to a Vector of Bool by using that function, and use a for-loop to describe the priority decoder, then the generated

Verilog code is what we expected. As a result, we always define a Boolean function to decompose the condition part out of the for-loop for priority decoders.

## 8.5. Parameterization and Provisos

### 8.5.1. Symptom

It is generally simple to write functions of modules with specific parameter. However, when we try to generalize and parameters, some problems about the provisos would appear, and some of them are subtle to solve.

For example, a simple vector rotating module is implemented as follows:

```
function Vector#(size, Bool) rotateN( Bit#(TLog#(size)) n, Vector#(size, Bool) v);
  return unpack(truncate(pack(append(v, v)) >> n));
endfunction
```

This seems correct in concept, but bsc requires a provisos `Add#(size, size, TAdd#(size, size))`. That requirement is always true, but bsc fails to conclude that. Manually adding the provisos wouldn't solve the problem at once, because the requirement is propagated upstream the hierarchy.

### 8.5.2. Solution

One possible solution is to add the tautological provisos in all the upstream hierarchy, but that is ugly. Another one is to explicitly write the lower level assignment like this:

```
function Vector#(size, Bool) rotateN( Bit#(TLog#(size)) n, Vector#(size, Bool) v);
  let shifted = pack(append(v,v)) >> n;
  Bit#(size) truncated = shifted[valueof(TSub#(size,1)):0];
  return unpack(truncated);
endfunction
```

The tricky part is that if we replace `let shifted...` by `Bit#(TAdd#(size,size)) shifted...`, then the original provisos is still needed. However, if `Bit#(size2) shifted...` is used instead, then we also need the provisos `Add#(size, size, size2)`, but this provisos won't propagate upstream.

## 8.6. Assignment on Vector

### 8.6.1. Symptom

If we have a variable of type Register of Vector, and we try to update the value in some of the elements in the vector, it might be some problems. If we write the code like this:

```
Reg#(Vector#(size, int)) arr <- mkReg(?);
for(Integer i=0;i<valueof(size);i=i+1) begin
  if( some_condition(...) )
    arr[i] <= val;
end
```

The compiler can handle it if the size is extremely small (<5). However, if the size is slightly larger, the static elaboration step would take a long time and generate a huge error message saying it finds some conflicting assignment which doesn't exist.

### 8.6.2. Solution

The code can be rewritten like this:

```
Reg#(Vector#(size, int)) arr <- mkReg(?);
Vector#(size, int) arr_w = arr;
for(Integer i=0;i<valueof(size);i=i+1) begin
  if( some_condition(...) )
    arr_w[i] = val;
end
arr <= arr_w;
```

By using a temporary variable to receive all the updated value, and assign it back to the original register of vector, the compiler doesn't have to check if we have conflicting assignment as these assignments are similar to the blocking assignment in Verilog.

The static elaboration time is also reduced.