Superscalar RISCV processor

Thomas Bourgeat, Jonathan Terry - 6.375

May 9, 2016

Superscalar RISCV processor

RISCV is a free ISA with a growing ecosystem of software and hardware implementations. There is a port of Linux working with MMU, privilege modes, and all the system tools that makes the design of real hardware especially complex. Hence, RISCV is a good ISA for architectural expeditions with realistic evaluation of performance.

Superscalar processors exist since early 90s. Contrary to their cousins "scalar" that execute at most one instruction by cycle, the superscalar processors process simultaneously two (or more) successive instructions. They exploit a form of **instruction level parallelism**.

An other critical architectural idea in modern processor is register renaming. To consider architectural register as virtual registers that are implemented by physical registers with a mapping that dynamically change allows the designer to handle WAR/WAW hazards for free.

This project will start from an out-of-order processor that is missing a renaming table, and modify it to a partially superscalar out-of-order processor with register renaming.

The existing Out-of-Order processor.

The existing out-of-order processor is divided in three parts represented in figure.

- the front-end that takes care of fetching and decoding instructions. So it handles virtual memory, request to caches and TLB, and some branch prediction.
- the back-end that takes care of renaming the instructions, to execute them and to commit them.

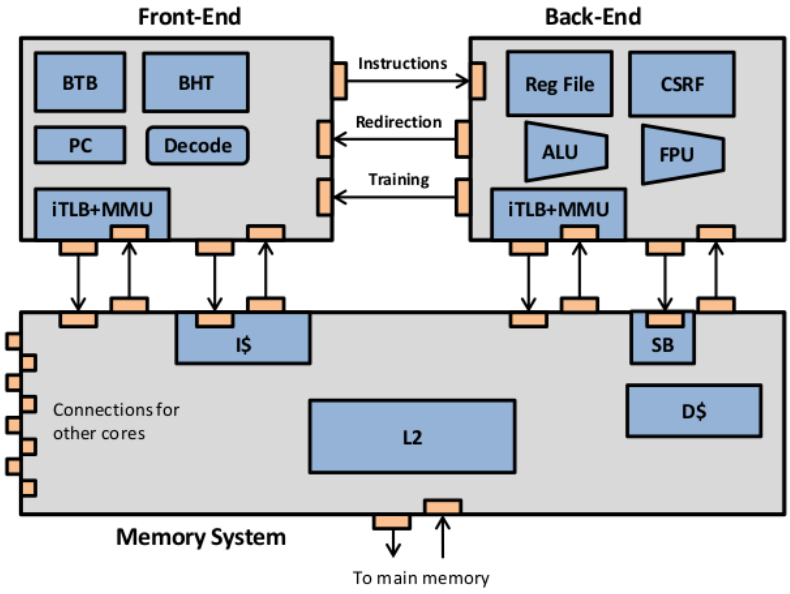

Figure 1: Figure 1

- the memory subsystem that performs the memory requests required by the front-end and the back-end.

We will mainly work on the front-end and on register renaming. Our changes will also require some changes in the memory subsystem. We discuss that in the present report.

Register renaming

Our goal was to explore a different way to design renaming tables where the free list is implicit.

Microarchitecture and interface.

First ArchRegs and PhyRegs represent the sets of architectural and physical registers involved in the currently handled instruction. Then we define the following interface.

```
interface RegRenamingTable;
    method PhyRegs get_renaming(ArchRegs r);
    method Action claim_renaming(ArchRegs r, SpecBits spec_bits);
    // The previous only uses r.dst.
    method Action commit;
    interface SpeculationUpdate specUpdate;
          method Action incorrectSpeculation(SpecTag tag);
          method Action correctSpeculation(SpecTag tag);
endinterface
```

We implement this interface using the following states :

```
// A commited table, that state the oldest ongoing bindings.
RegFile#(ArchRIndx, PhyRIndx) data <- mkRegFileFull();

// In-flight renaming stack
Vector#(TSub#(NumPhyReg,NumArchReg), Reg#(Bool)) valid <- replicateM(mkReg(False));
Vector#(TSub#(NumPhyReg,NumArchReg), Reg#(PhyRIndx))
    stackPhy <-replicateM(mkReg(unpack(0)));
Vector#(TSub#(NumPhyReg,NumArchReg), Reg#(ArchRIndx))
    stackArch <- replicateM(mkReg(unpack(0)));
Vector#(TSub#(NumPhyReg,NumArchReg), Reg#(SpecBits))
    stackSpecBits <- replicateM(mkReg(unpack(0)));
```

Informally we look first in the in-flight renaming before we look into the table of commited renamings. So *get_renaming* implements a look through a Content Adressable Memory that is the In-flight renaming stack.

There is one subtile issue that is not specified in the interface : we require commit and incorrect/correctSpeculation to be able to fire simultaneously.

Because of that, and performance issues (fire more rules in the same cycle), several registers have been replaced by EHR in the final implementation.

Implementation

We first did a naive implementation using static elaboration with for loops. The compile time was huge, it was impossible to compile even for a stack of size 8. We then moved to a functional programming style because there were rumour that it would be more efficient to compile.

These rumours were true, it is a lot faster to replace

```
for (Integer i = 0; i< 42; i=i+1) begin
    foo[i]= f(bar[i]);
end
```

by

```
foo = map(f,bar);
```

After a discussion with Nirav Dave, a plausible hypothesis for why is the following : when bsc see a for loop, it does not know what kind of state is passed through every iteration of the loop, there are potentially arbitrarily complex connexions between each iterations. However when the bsc compiler see a map statement, it knows that all the fields of this map are independent, it changes a lot in term of elaboration, it can elaborate each element independently of the others.

So the designer should always use the most specific function when writting her code, indeed it could get a lot faster because the compiler can apply some optimization knowing that there are some specific structure in the problem solved.

The same reason apply why fold is faster than a for loop : the compiler exactly knows what is the datastructure that is accumulated on and so it can optimize for that, there are no surprise connexion that could happen at the last iteration of the loop.

We can remark that some static analysis and heuristic should allow the bluespec compiler to perform these optimizations automatically.That is one of the few cases where functional programming is naturally more efficient.

Subtility of scheduling in the Bluespec compiler.

At some point we were stuck on a scheduling problem that took lot of times to solve so we decided to report it here, it could be useful for future work.

We will give here a module that sounds perfectly correct, and is correct with the description of what the scheduler does in the model given in class. However it behaves in an unexpected way (in my opinion) when we add synthesize boundaries.

(This example is a minimalization of the one we were stuck on in our design, this code is written by Andy Wright)

```
module example()
    Reg#(Bool) foo <- mkReg(False);
    Reg#(Bool) bar <- mkReg(False);

    rule second
        foo <= !bar;
    endrule

    method first
        $display(foo);
        $display(bar);
    endrule

    method third
        bar <= !bar;
    endmethod
endmodule
```

This module seems to be perfectly fine, the names of the methods implicitly describe the order of logical firing that are exported by the compiler.

Without synthesize boundaries everything works fine, however when we add synthesize boundaries, then something unexpected happens.

Indeed, let's say that an external module calls in the same rule method first and third. Then this would be accepted with non modular compilation because the scheduler would consider this rule valid and this rule would never fire when rule "second" fire (the compiler would try to find such a logical order). However with modular compilation it would break everything because the rule would be accepted, and as the rule second is schedule in between the two methods, this top level rule and rule "second" would conflict. (With synthesize boundaries the scheduling within the module is frozen)

The way modular compilation works in Bluespec is that it exports a special Sequence Before Restricted schedule. Meaning that it accepts to be scheduled

before if and only if there is no request of firing the two methods in the same rule (in which case we could not prevent the rule "second" to fire...). So in the actual bluespec compiler there are at least C, CF, SB, SA, SBR and SAR.

Superscalar

Dependencies and misprediction

First let's explain what is making superscakar not trivial. When one wants to process simultaneously two instructions, 2 situations may happen :

- There are absolutely no dependencies between the two instructions and so the superscalar processor is roughly a two parallel pipeline of a scalar processor. That is the easy case.
- There are some dependencies and the processor should be aware for it. For example if we have the two following instructions :

  ```
  pc :  a = a + 2
  pc+4: b <- a
  ```

  The scoreboard - that keeps track of the RAW hazards, needs to inform the second instruction of the new write on register a.

  In this case the superscalar processor cannot be a mere duplication of the pipeline where the two pipelines would barely communicate with each other. There is an implicit ordering between those two pipeline. This kind of dependencies also exists in the front end. For example in decoding there are redirection going on, we need to make sure that when a redirection happen, the next instruction is considered as wrong path instruction. (So that we flip the epoch and that the next instruction that could be fetch in the same cycle see the flip in the epoch).

Superscalar fifo

The superscalar fifo is the core tool required to do the superscalar processor in a bluespec style. Indeed that will be the glu that is going on between every stage of the pipeline. The interface is the following :

```
interface SupFifo#(numeric type k, numeric type n, type t);
    method Bool notFull;
    interface Vector#(k,function Action enq(t x)) enqS;
    method Bool notEmpty;
    interface Vector#(k,function Action deq()) deqS;
```

```
    interface Vector#(k,function t first) firstS;
    method Action clear;
endinterface
```

Internally I satisfy this interface using the following state structure :

```
Vector#(k, FIFOF#(t)) internalFifos <- replicateM(mkSizedFIFOF(valueOf(n)));
```

And then we have a datastructure that keeps track of the fifo in which we should enqueue/dequeue updated this way:

```
Ehr#(TAdd#(1,k), Bit#(TLog#(k))) enqueueFifo <- mkEhr(unpack(0));
Ehr#(TAdd#(1,k), Bit#(TLog#(k))) dequeueFifo <- mkEhr(unpack(0));

function Action enq(Integer i, t x);
    return (action
        enqueueFifo[i]<= enqueueFifo[i]+1;
        internalFifos[enqueueFifo[i]].enq(x);
        endaction);
endfunction
```

The problem is that the compiler is not able to figure out that there is no conflict between the different enq functions. Indeed it is not obvious that all the port of enqueueFifo refers to different numbers. This holds because we have a number of enq function less or equal than k. (Indeed we always add 1, so we will wrap around at function k).

One could think that I could just add attributes (like (* conflict-free "foo,bar" *)). It is not possible to add such attributes on methods that are inside a vector with the current bluespec compiler. More generally, attributes are not (to my knowledge) first order objects, they can't be manipulated in the language like rules can be using addRules, so when we have a parametric number of methods we can't use attributes on this methods.).

I used a workaround - that I believe is quite general - to impose whatever scheduling constraint I wanted on my Action methods : the methods don't directly touch the state of the module, they perform requests modifying EHRs, where we organize the ports of the EHRs to achieve the scheduling we want between the rule. At the end of the cycle we have a canonicalize rule that takes all the request - so all indirectly all the calls to action methods - that the user wants to perform this cycle, the rule performs them on the state and reset the status of the EHR to prepare them for the next cycle with new requests. We need to manually lift the guard up because the guard entirely disappeared with those "proxy EHRs". Here is a snippet :

```
Vector#(k, FIFOF#(t)) internalFifos <- replicateM(mkSizedFIFOF(valueOf(n)));
Ehr#(TAdd#(1,k), Bit#(TLog#(k))) enqueueFifo <- mkEhr(unpack(0));
Ehr#(TAdd#(1,k), Bit#(TLog#(k))) dequeueFifo <- mkEhr(unpack(0));
Vector#(k, Ehr#(2,Maybe#(t))) enqueueElement <- replicateM(mkEhr(tagged Invalid));
Vector#(k, Ehr#(2, Bool)) willDequeue <- replicateM(mkEhr(False));

// Those are the future methods
// (as there is a parametric number of them,
// there are still functions at this point)
// we will map them in the vector later

function Action enq(Integer i, t x); // Needs k to be a power of 2
    return (action
        when(internalFifos[enqueueFifo[0]+fromInteger(i)].notFull(),noAction);
        enqueueElement[i][0] <= tagged Valid x;
        endaction);
endfunction

function Action deq(Integer i);
    return (action
        when(internalFifos[dequeueFifo[0]+fromInteger(i)].notEmpty,noAction);
        willDequeue[i][0] <= True;
        endaction);
endfunction
function t first(Integer i);
       return (internalFifos[dequeueFifo[0]+fromInteger(i)].first());
endfunction

rule canonicalize;
for (Integer i = 0; i < valueOf(k); i = i+1) begin
    case (enqueueElement[i][1]) matches
    tagged Invalid : noAction;
    tagged Valid .el:
        begin
        enqueueFifo[i] <= enqueueFifo[0]+fromInteger(i)+1;
        internalFifos[enqueueFifo[0]+fromInteger(i)].enq(el);
        enqueueElement[i][1] <= tagged Invalid;
        end
    endcase
    if (willDequeue[i][1]) begin
        dequeueFifo[i] <= dequeueFifo[0] + fromInteger(i) + 1;
        internalFifos[dequeueFifo[0]+fromInteger(i)].deq;
        willDequeue[i][1] <= False;
    end
end
endrule
```

Superscalarization of the front end

Here we describe how we superscalarized the fetch1 (tlb request)/fetch2 (tlb response/mem request)/ fetch3 (memory response)/decode pipeline of the processor.

We will only superscalarize when the pc does not refer to the last instruction of a cacheline. More generally when the degree is more than 2, we superscalarize as much as we can in a cacheline.

The decode stage is really straightforward as we can just duplicate the combinatorial circuit and we handle the concurrent access on the epoch using EHRs.

- Fetch 1 computes how many instructions it can superscalarize (For example we don't want to enqueue pc+4 if pc+4 is not predicted by the branch predictor). We send the same request to TLB, knowing that we will be able to superscalarize only if the instructions are in the same cacheline. So the translation for pc+4 is pc_phy + 4 for example. (page granularity is lower than cache line granularity). We take care of ppc that become the next instruction after the last instruction that will be superscalarized.
- Fetch 2 send the same memory instruction as we just want strikes in the same cacheline as previously.
- The cache is modified to answer a vector of size k of Maybe#(Instructions) that consists of i<k instructions that are before the end of the cache line.
- Fetch 3 get back the vector of instructions from the cache, and enqueue them into the superscalar fifo.
- k decode rules that can fire together (they are ordered) will get from the superscalar fifo fromFetch3, decode the instruction, modify the state of the epoch and PC acccordingly, and enqueue into the next superscalar fifo that goes to register renaming.

Difficulties

There are a couple of difficulties we had to handle :

- The memory are not addressed at the same granularity at different place of the processor. For example in the cache there are cache lines formed of double words (64 bits), but the instructions are 32 bits wide. This kind of thing is a very good source of off-by-one error.
- At the beginning of the frontend, the instructions are somewhat "virtual" in the sense that one instruction will make several instructions appears in stage 3. (Basically every instructions between pc and ppc will be silently

fetched, but does not exist in the first stages). For this reason there are some combinatorial computations going on to predict what we will superscalarize. The logic is the following : if we have a superscalarity degree of n, we will superscalarize the biggest strike of instruction less than n after pc such that all the instructions are in the same cache line, and the btb indicates pc+4 for all the instructions of this strike.

- We had to modify several interface, between the ICache and the fetch stage, and between the fetch stage and the processor. It sounded scary at first but it actually went well. We also had to modify the cache slighlty, as I did not know anything about the way it was written, I was afraid.

Test : correctness and performance.

Set of tests

The test suite consists in the following :

- assembly tests, assembly_fp tests (for floating points)
- Small C benchmarks written in different styles (median, matrix multiplication, mandelbrot, nqueens, pascal, various sorts, thuemorse, towers, vvadd ...).
- the linux kernel

Correctness

During the project we ran our processor on all the assembly tests and C benchmarks. We made sure that they passed after every small step we did in the project.

At the end of the project we tried linux. (The reason why we did not try before is that the synthesis for FPGA takes forever). At first it did not work because of a minor bug in register file for floating points numbers. We fixed this bug, know it works!

Performance

At first we benchmarked and evaluated the performance and realized that we were losing performance! The reason for that is that when we were mispredicting something we were enqueuing more garbage in the pipeline that took time to evacuate, as we were just poisoning the instructions. (Our pipeline being slower at the end than at the beginning)

So I changed the redirect rule to kill as much things as possible in place, to avoid having this kind of problem. I also added an epoch filter earlier in the front end, to drop wrong path instructions earlier. (I cannot kill in place all the wrong path instructions because there are requests flying in the TLB and in the I-cache).

After this change, I see no difference in performance between the partially superscalar out-of-order and the starting out-of-order. Apparently we won't see any benefit in performance before we superscalarize the commit.