Register renaming and superscalar RISCV processor

Thomas Bourgeat - 6.375

May 11, 2016
RISCV out-of-order superscalar implementation
Overview

Figure 1:
Descriptive overview

- 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.
- the memory subsystem that performs the memory requests required by the front-end and the back-end.
Renaming
interface RegRenamingTable;
    method PhyRegs get_renaming(ArchRegs r);
    method Action claim_renaming(ArchRegs r,
                               SpecBits spec_bits);
    method Action commit;
    interface SpeculationUpdate specUpdate;
        method Action incorrectSpeculation(SpecTag tag);
        method Action correctSpeculation(SpecTag tag);
endinterface
Naive implementation

<table>
<thead>
<tr>
<th>Arch</th>
<th>Phy</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>P4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>R32</td>
<td>P57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arch</th>
<th>Phy</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>P2</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>P1</td>
<td>0001</td>
</tr>
</tbody>
</table>

Figure 2: Figure 1
State used

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)));

so slow

- It cannot elaborate a normal size circuit.
Would there be some magic tricks? Functional Programming!

let listzip = zip4(
    readVEhr(0, valid),
    readVReg(stackArch),
    readVReg(stackPhy),
    indexes);

let leftToRightR1 = map_maybe(
    tpl_3,
    find(vectorSearchL(archR1, enqP),
        reverse(listzip)
    )
);
Why is it faster?

- When I write \( \text{map}(f,v) \) the compiler knows that all the iterations can be elaborated independently.
- When I write a fold, the compiler knows exactly what is the state, or the “dependentness” that is passing around just looking at the type of the accumulator.
- When I write a for, the compiler cannot assume anything about the relation between the different iterations of the loop without doing a complex static analysis.
- Functional programming is intrinsically faster! (I knew it!)
Superscalar
A new fifo (proposal):

```plaintext
interface Fifo#(a);
    method Action enq1(a);
    method Action enq2(a);
    method a first();
    method a second();
    method Action deq1(a);
    method Action deq2(a);
```
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
Architectural realization

Figure 3: Figure 1
States

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));

function Action enq(Integer i, t x);
  return (action
    enqueueFifo[i]<= enqueueFifo[i]+1;
    internalFifos[enqueueFifo[i]].enq(x);
  endaction);
endfunction
BSC does not like this code

- BSC cannot realize that all the port of the ehr enqueueFifo will have different value.

  ```java
  enqueueFifo[i] <= enqueueFifo[i] + 1;
  internalFifos[enqueueFifo[i]].enq(x);
  ```

- Indeed: it is a hard problem, it is expected.
- No way to overwrite what bsc does for this scheduling.
So how do we do?

Vector#(k, Ehr#(2, Maybe#(t))) willEnqueue
  <- replicateM(mkEhr(tagged Invalid));
Vector#(k, Ehr#(2, Bool)) willDequeue
  <- replicateM(mkEhr(False));

function Action enq(Integer i, t x);
  return (action
    when(
      when(
        internalFifos[enqueueFifo[0]+fromInteger(i)].notFull(),noAction
      );
      willEnqueue[i][0] <= tagged Valid x;
    endaction);
endfunction
And we canonicalize

rule canonicalize;
for (Integer i = 0; i < valueOf(k); i = i+1) begin
  case (willEnqueue[i][1]) matches
default: noAction;
  tagged Valid .el:
    begin
      enqueueFifo[i] <= enqueueFifo[0]+fromInteger(i)+1;
      internalFifos[enqueueFifo[0]+fromInteger(i)] .enq(el);
      willEnqueue[i][1] <= tagged Invalid;
    end
  endcase
end
endrule
Superscalarization of the front-end
From pc we compute 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).
- If we have a superscalarity degree of n, we will fetch 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 stop if we arrive at the end of a cacheline or that the btb indicate a jump.

This is required to compute ppc
We just forward the request to memory at this point we still have one instruction!
The cache is modified to answer a vector of size $k$ of $\text{Maybe}(\text{Instructions})$ that consists of $i < k$ instructions that are before the end of the cache line and in a strike.
Fetch3

- We receive several instructions from memory!
- We enqueue them in order, using our superscalar fifo.
Decode

- We have as many decode rules as the degree of superscalarity (using addRules)
- this rules touch an EHR pc, and epoch in order.
- and then enqueue in the next superscalar fifo in order too.
Correctness

- :) : We pass all the benchmark tests and all the assembly tests
- :( : We fail on linux (apparently spike throw a page fault when we don’t), we have several hypothesis why but it will take time to debug. It takes time because every modification takes roughly 3 hours to test...
Performance

- :( : We were first slower!
- :) : After killing instructions in place we have the same performance (because the backend is not superscalarized yet).
Thank you

Thank you!