EHRs: Designing modules with concurrent methods

Arvind
Computer Science & Artificial Intelligence Lab. Massachusetts Institute of Technology

Elastic pipeline

\[
\begin{align*}
\text{rule stage1 if } & (\text{True}); \\
& \text{fifo1.enq(f1(inQ.first()));} \\
& \text{inQ.deq();} \\
\text{endrule} \\
\text{rule stage2 if } & (\text{True}); \\
& \text{fifo2.enq(f2(fifo1.first()));} \\
& \text{fifo1.deq();} \\
\text{endrule} \\
\text{rule stage3 if } & (\text{True}); \\
& \text{outQ.enq(f3(fifo2.first()));} \\
& \text{fifo2.deq();} \\
\text{endrule}
\end{align*}
\]

Whether these rules can fire concurrently depends crucially on the properties of fifo methods.
Elastic pipeline
Expressed as a single rule

rule elasticPipeline;
  if (inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f0(inQ.first); inQ.deq end;
  if (fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f1(fifo1.first); fifo1.deq end;
  if (fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f2(fifo2.first); fifo2.deq) end;
endrule

What is wrong?

Elastic pipeline
Expressed as a single rule using guard lifting

rule elasticPipeline;
  if (inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f0(inQ.first); inQ.deq end;
  if (fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f1(fifo1.first); fifo1.deq end;
  if (fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f2(fifo2.first); fifo2.deq) end;
endrule

All stages may be active simultaneously if an enq and deq on a FIFO can be performed simultaneously
Designing FIFOs

module mkCFFifo (Fifo#(1, t));
  Reg#(t)  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action
    enq(t x) if (!v);
    v <= True;   d <= x;
  endmethod
  method Action
    deq  if (v);
    v <= False;
  endmethod
  method t first  if (v);
    return (d);
  endmethod
endmodule

One-Element FIFO

module mkCFFifo (Fifo#(1, t));
  Reg#(t)  d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action
    enq(t x) if (!v);
    v <= True;   d <= x;
  endmethod
  method Action
    deq  if (v);
    v <= False;
  endmethod
  method t first  if (v);
    return (d);
  endmethod
endmodule

1. What if enq and deq were executed together?
2. Can enq and deq be ready simultaneously?
Two-Element FIFO

module mkCFFifo (Fifo#(2, t));
Reg#(t)   da  <- mkRegU();
Reg#(Bool) va <- mkReg(False);
Reg#(t)    db <- mkRegU();
Reg#(Bool) vb <- mkReg(False);
method Action enq(t x) if (!vb);
        if va then begin db <= x; vb <= True; end
        else begin da <= x; va <= True; end
endmethod
method Action deq if (va);
        if vb then begin da <= db; vb <= False; end
        else begin va <= False; end
endmethod
method t first if (va);
    return da;
endmethod
endmodule

Assume, if there is only one element in the FIFO it resides in da

Single-rule version

rule stage1;
    if(inQ.notEmpty && fifo1.notFull)
        begin fifo1.enq(f1(inQ.first)); inQ.deq end;
    if(fifo1.notEmpty && fifo2.notFull)
        begin fifo2.enq(f2(fifo1.first); fifo1.deq end;
    if(fifo2.notEmpty && outQ.notFull)
        begin outQ.enq(f3(fifo2.first); fifo2.deq) end;
endrule

This rule is illegal if concurrent operations on FIFOs are not permitted
Limitations of registers

Limitations of a language with only the register primitive

- No communication between rules or between methods or between rules and methods in the same atomic action i.e. clock cycle
- Can’t express a FIFO with concurrent enq and deq

EHR: Ephemeral History Register

A new primitive element to design modules with concurrent methods
Designing FIFOs using EHRs

- **Conflict-Free FIFO**: Both enq and deq are permitted concurrently as long as the FIFO is not-full and not-empty
  - The effect of enq is not visible to deq, and vise versa
- **Pipeline FIFO**: An enq into a full FIFO is permitted provided a deq from the FIFO is done simultaneously
- **Bypass FIFO**: A deq from an empty FIFO is permitted provided an enq into the FIFO is done simultaneously
One-Element Pipelined FIFO

module mkPipelineFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(False);
    method Action enq(t x)
        if (!v[1]);
            d <= x;
            v[1] <= True;
    endmethod
    method Action deq
        if (v[0]);
            v[0] <= False;
    endmethod
    method t first
        if (v[0]);
            return d;
    endmethod
endmodule

Desired behavior
- deq < enq
- first < deq
- first < enq

In any given cycle:
- If the FIFO is not empty then simultaneous enq and deq are permitted;
- Otherwise, only enq is permitted

One-Element Bypass FIFO

using EHRs

module mkBypassFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
    Ehr#(2, t) d <- mkEhr(??);
    Ehr#(2, Bool) v <- mkEhr(False);
    method Action enq(t x)
        if (!v[0]);
            d[0] <= x;
            v[0] <= True;
    endmethod
    method Action deq
        if (v[1]);
            v[1] <= False;
    endmethod
    method t first
        if (v[1]);
            return d[1];
    endmethod
endmodule

Desired behavior
- enq < deq
- first < deq
- enq < first

In any given cycle:
- If the FIFO is not full then simultaneous enq and deq are permitted;
- Otherwise, only deq is permitted
Two-Element Conflict-free FIFO

module mkCFFifo(Fifo#(2, t)) provisos(Bits#(t, tSz));

Ehr#(2, t) da <- mkEhr(?);
Ehr#(2, Bool) va <- mkEhr(False);
Ehr#(2, t) db <- mkEhr(?);
Ehr#(2, Bool) vb <- mkEhr(False);

rule canonicalize if (vb[1] && !va[1]);
vb[1] <= False; endrule

method Action enq(t x) if (!vb[0]);
    db[0] <= x; vb[0] <= True; endmethod

method Action deq if (va[0]);
    va[0] <= False; endmethod

method t first if (va[0]);
    return da[0]; endmethod
endmodule

Assume, if there is only one element in the FIFO it resides in da

Desired behavior
eng CF deg
first < deq
first CF enq

In any given cycle:
- Simultaneous enq and deq are permitted only if the FIFO is not full and not empty

Register File:

normal and bypass

- Normal rf: the effect of a register update can only be seen a cycle later, consequently, reads and writes are conflict-free
  - \{rd1, rd2\} < wr
- Bypass rf: in case of concurrent reads and write, check if rd1==wr or rd2==wr then pass the new value as the result and update the register file, otherwise the old value in the rf is read
  - wr < \{rd1, rd2\}
Normal Register File

module mkRFile(RFile);
  Vector#(32,Reg#(Data)) rfile <- replicateM(mkReg(0));
  method Action wr(Rindx rindx, Data d);
    rfile[rindx] <= d;
  endmethod
  method Data rd1(Rindx rindx) = rfile[rindx];
  method Data rd2(Rindx rindx) = rfile[rindx];
endmodule

Bypass Register File using EHR

module mkBypassRFile(RFile);
  Vector#(32,EHR#(2, Data)) rfile <- replicateM(mkEHR(0));
  method Action wr(Rindx rindx, Data d);
    rfile[rindex][0] <= d;
  endmethod
  method Data rd1(Rindx rindx) = rfile[rindx][1];
  method Data rd2(Rindx rindx) = rfile[rindx][1];
endmodule
Bypass Register File
with external bypassing

module mkBypassRFile(BypassRFile);
RFFile rf <- mkRFile;
Fifo#(1, Tuple2#(RIndx, Data))
    bypass <- mkBypassSFifo;
rule move;
    begin
        rf.wr(bypass.first);
        bypass.deq
    endrule
method Action wr(RIndx rindx, Data d);
    bypass.enq(tuple2(rindx, d));
endmethod
method Data rd1(RIndx rindx) =
    return (!bypass.search1(rindx)) ? rf.rd1(rindx) :
        bypass.read1(rindx);
method Data rd2(RIndx rindx) =
    return (!bypass.search2(rindx)) ? rf.rd2(rindx) :
        bypass.read2(rindx);
endmodule
N-element Conflict-free FIFO

Checking for emptiness and fullness
- Empty: \( \text{enqP} = \text{deqP} \)
- Full: \( \text{enqP} = \text{deqP} + \text{FIFO\_size} \)

To deal with the wrap around problem, assume \( \text{enqP} \) and \( \text{deqP} \) can contain indices for up to twice the size of the FIFO. Pointers are incremented modulo \( 2^n \) while buffer is accessed by pointer modulo \( n \) values.

N-element Conflict-free FIFO

- An enq updates \( \text{enqP} \) and puts the data into the array. It also sets \( \text{enqEn} \) to false to prevent further enqueues.
- A deq updates \( \text{deqP} \) and sets \( \text{deqEn} \) to false to prevent further dequeues.
- Canonicalize rule calculates the new count and sets the \( \text{enqEn} \) and \( \text{deqEn} \) bits appropriately.
N-element Conflict-free FIFO methods

```verilog
method Action enq(t x); if(enqEn[0]);
enqP[0] <= (enqP[0] + 1) % n2; // n2 is 2*nb
enqEn[0] <= False;
d[enqP[0] % nb] <= x;
endmethod

method Action deq if(deqEn[0]);
deqP[0] <= (deqP[0] + 1) % n2;
deqEn[0] <= False;
endmethod

method t first if(deqEn[0]);
return d[deqP[0] % nb];
endmethod
endmodule
```

N-element Conflict-free FIFO: The canonicalization rule

```verilog
rule canonicalize;
let cnt = enqP[1] >= deqP[1]?
enqP[1] - deqP[1]:
(enqP[1] % nb + nb) - deqP[1] % nb;
endrule
```