Sorting

Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

But first… some lab stuff

- Problems see why two rules conflict
- Bluespec flag:
  `–show-rule-rel <rule1> <rule2>`

```
RL_m_appEnv_wr_app_core_proc_exec
RL_m_appEnv_wr_app_core_proc_writeback
```

- Once you have configured, edit `SConstruct` in `build/default/<modelname>/pm`
- Add new flag to BSC_FLAGS string

Task:Sorting

- Given an:
  - array in memory (base address)
  - it's length N
- Sort the array in place

Algorithm: Bubble Sort

```
Reg#(int) i  <- mkReg(0);
Reg#(int) j  <- mkReg(0);
Reg#(int) n  <- mkReg(0);
Reg#(int) t1 <- mkReg(0);
Reg#(int) t2 <- mkReg(0);

Stmt doSort = seq
 for(i<=0; i<N; i<=i+1)
  for(j<=i; j<N; j<=j+1)
   seq mem.req(Rd j);
       action let x <- mem.resp(); t1 <= x; endaction
       mem.req(Rd (j+1));
       action let x <- mem.resp(); t2 <= x; endaction
       if (t2 > t1)
         seq mem.req(Wr{j   ,t2});
             mem.req(Wr{j+1,t1}); endseq;
endseq;

FSM sortFn <- mkFSM(doSort);
```

$N^2/2$

4 cycles if in-order, 6 if out-of-order

Making the design better

- Algorithmic Improvements
  - Switch to O(N*log(n)) sort
    - Quick sort
    - Merge sort
    - Shell sort
- Which should we pick?

Improving Bandwidth Usage

- Large memories are cannot fit on FPGA
  - Separate chip
  - Talk to it via Bus
- Bus transactions take a few cycles to setup
  - Handshake and reservation takes a few cycles
  - Longer bursts are more efficient

Faster Memory Interface

```
typedef struct{
   Rd struct{numWord:  int,
             baseAddr: Addr};
   Wr struct{numWord:  int,
             baseAddr: Addr};
} MemCmd deriving(Eq, Bits);

Interface LocalMem
  method Action      startCmd(MemCmd x);
  method Action         dataIn(Value x);
  method ActionValue#(Value) dataOut();
```

Improving Mem Bursts:
A Simple Cache Implementation

```
LocalMem                    mem <- mkLocalMem;

Reg#(Maybe#(Addr)) baseAddr  <- mkReg(Invalid);
RegFile#(Addr,Val) cacheline <- mkRegFile(0, N-1);

Reg#(Maybe#(Addr))    memPtr <- mkReg(Invalid);
FIFO#(Op)             reqQ   <- mkFIFO;
FIFO#(Value)          respQ  <- mkFIFO;
Reg#(Bool)           doingWB <- mkReg(False);
```

Do base read/writes

```
method Action req(Op x);
  reqQ.enq(x);
endrule

method ActionValue#(Value) resp();
  respQ.deq(); return respQ.first();
endmethod

rule doRead(reqQ.first matches tagged ORd .a &&&
            inLine(baseAddr, a));
  respQ.enq(cacheLine.sub(lineAddr(a)));
  reqQ.deq();
endrule

rule doWrite(reqQ.first matches tagged OWr {.a,.v} &&&
             inLine(baseAddr, a));
 cacheLine.upd(lineAddr(a),v));
endrule
```

Writeback Cache Values

```
rule startWB(memPtr matches Valid .a && !doingWB &&
             !inLine(baseAddr, reqAddr(reqQ.first)));
  mem.cmd(Wr{reqAddr});
  memPtr  <= Valid baseAddr;
  doingWB <= True;
endrule

rule doWriteback(memPtr matches Valid .a && doingWB);
 mem.dataIn(mem.sub(a));
 memPtr <= isFinalAddr(a) ? Invalid: Valid(a+1);
 if (isFinalAddr(a))
  begin
   baseAddr <= Invalid;
   doingWB <= False;
  end
endrule
```

Load Cache Lines

```
rule startCacheLoad(memPtr matches Invalid &&
                    baseAddr matches Invalid);
  let ba = baseAddr(reqAddr(reqQ.first));
  mem.cmd(Rd ba);
  memPtr <= ba;
endrule


rule doStore(memPtr matches Valid .a &&
             baseAddr matches Invalid);
  let v <- mem.dataOut();
  mem.upd(a,v);
  memPtr <= isFinalAddr(a) ? Invalid: Valid(a+1);
endrule
```

Choosing an algorithm

- Which algorithm should you pick?
  - Should be efficient algorithmically
  - ***Predicatable*** linear memory accesses
    - Amenable to memory bursts
- MEMOCode design contest
  - MIT chose merge sort

Making a small Merger Unit

- Enter in two ordered streams of data
- Output the unified stream in order

```
interface Merger;
  method in1(Value x);
  method in2(Value x);
  method ActionValue#(Value) out();
endinterface
```

Simple Merger

```
module mkMerger(Merger);
 FIFO#(Value) i1  <- mkFIFO;
 FIFO#(Value) i2  <- mkFIFO;
 FIFO#(Value) out <- mkFIFO;
 rule merge1 (i1.first <= i2.first);
  out.enq(in1.first); in1.deq;
 endrule
 rule merge2 (i1.first > i2.first);
  out.enq(in2.first); in2.deq;
 endrule

 method in1(x) = i1.enq(x);
 method in2(x) = i2.enq(x);
 method ActionValue#(Value) out();
  out.deq(); return out.first();
 endmethod
endmodule
```

Problem! This can never be empty

Adding lengths to Merger

- Record the length of streams
  - Input N values, output 2N values

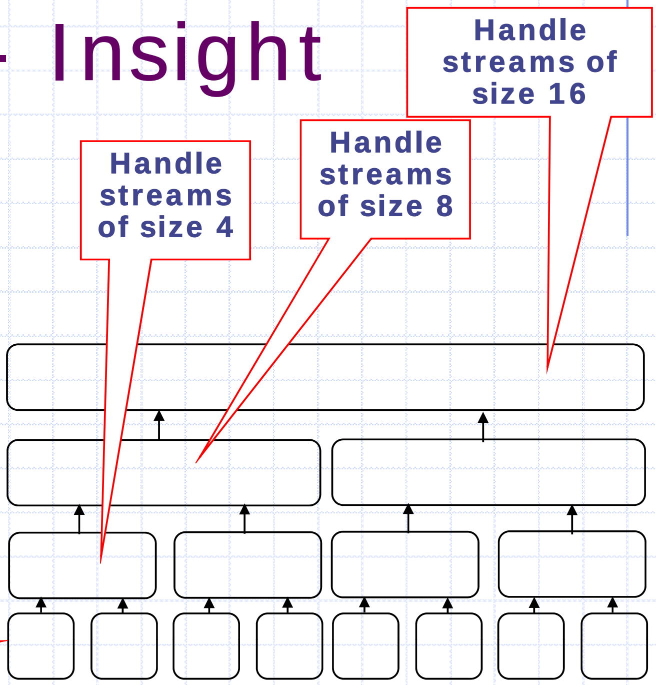
```
interface Merger
  method streamLength(int x);
  method in1(Value x);
  method in2(Value x);
  method ActionValue#(Value) out();
endinterface
```

Extend Merger

```
module mkMerger(Merger);
 Reg#(int)   cnt1 <- mkRegU; Reg#(int)  cnt2 <- mkRegU;
 FIFO#(Value)  i1 <- mkFIFO; FIFO#(Value) i2 <- mkFIFO;
 FIFO#(Value) out <- mkFIFO;
 rule merge1 (i1.first <= i2.first);
  out.enq(in1.first); in1.deq; cnt1<=cnt1-1; endrule
 rule merge2 (i1.first > i2.first);
  out.enq(in2.first); in2.deq; cnt2<=cnt2-1; endrule
 rule finish2(cnt1 == 0 && cnt2 > 0);
  cnt2<=cnt2–1; out.enq(in2.first); in2.deq; endrule
 rule finish1(cnt2 == 0 && cnt1 > 0);
  cnt1<=cnt1–1; out.enq(in1.first); in1.deq; endrule
 method Action streamLength(n) if(cnt1==0 && cnt2==0);
   cnt1 <= n; cnt2 <= n; endmethod
 method Action in1(x) if (cnt1>=0); i1.enq(x); endmethod
 method Action in2(x) if (cnt2>=0); i2.enq(x); endmethod
 method ActionValue#(Value) out();
  out.deq(); return out.first(); endmethod
endmodule
```

MIT solution - Insight

- Merge can be done in a massive tree structure



Handle streams of size 16

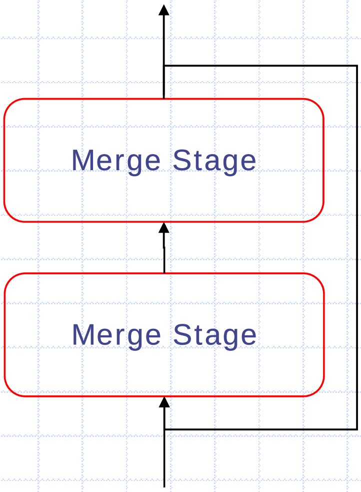
Handle streams of size 8

Handle streams of size 4

Handle streams of size 2

MIT solution High-level Idea

- Each level of merger handles the same amount of data
- We'd like to reuse the logic for each level BUT FIFO sizes and the number of FIFOs are different
- Use a single unified cache to implement all FIFOs in a stage
  - Can change the sizes the FIFOs dynamically



More details in Paper

- http://people.csail.mit.edu/mdk/papers/memocode_2008_cryptosorter.pdf