Bluespec-3: A non-pipelined processor

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Outline
- First we will finish the last lecture
  - Synchronous pipeline
  - 802.11a results
- One-Element FIFO
- Non-pipelined processor
  - with magic memory
  - with decoupled, req-resp memory
Pattern-matching: A convenient way to extract datastructure components

```c
typedef union tagged {
    void Invalid;
    t Valid;
} Maybe#(type t);

case (m) matches
tagged Invalid : return 0;
tagged Valid .x : return x;
endcase

if (m matches (Valid .x) && (x > 10))
```

- The && is a conjunction, and allows pattern-variables to come into scope from left to right

Synchronous pipeline

```verilog
rule sync-pipeline (True);
if (inQ.notEmpty())
begin sReg1 <= Valid f1(inQ.first()); inQ.deq(); end
else sReg1 <= Invalid;
case (sReg1) matches
tagged Valid .sx1: sReg2 <= Valid f2(sx1);
tagged Invalid: sReg2 <= Invalid;
case (sReg2) matches
tagged Valid .sx2: outQ.enq(f3(sx2));
endrule
```
Folded pipeline

```
rule folded-pipeline (True);
    if (stage==1)
        begin sxIn= inQ.first(); inQ.deq(); end
    else sxIn= sReg;
    sxOut = f(stage,sxIn);
    if (stage==n) outQ.enq(sxOut);
    else sReg <= sxOut;
    stage <= (stage==n)? 1 : stage+1;
endrule
```

Need type declarations for sxIn and sxOut

---

802.11a Transmitter Synthesis results (Only the IFFT block is changing)

<table>
<thead>
<tr>
<th>IFFT Design</th>
<th>Area (mm²)</th>
<th>Throughput Latency (CLKs/sym)</th>
<th>Min. Freq Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelined</td>
<td>5.25</td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>(48 Bfly-4s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinational</td>
<td>4.91</td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>(48 Bfly-4s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Folded</td>
<td>3.97</td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>(16 Bfly-4s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Super-Folded</td>
<td>3.69</td>
<td>06</td>
<td>1.5 MHz</td>
</tr>
<tr>
<td>(8 Bfly-4s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF(4 Bfly-4s)</td>
<td>2.45</td>
<td>12</td>
<td>3.0 MHz</td>
</tr>
<tr>
<td>SF(2 Bfly-4s)</td>
<td>1.84</td>
<td>24</td>
<td>6.0 MHz</td>
</tr>
<tr>
<td>SF(1 Bfly4)</td>
<td>1.52</td>
<td>48</td>
<td>12 MHz</td>
</tr>
</tbody>
</table>

TSMC .18 micron; numbers reported are before place and route.
Why are the areas so similar

- Folding should have given a 3x improvement in IFFT area
- BUT a constant twiddle allows low-level optimization on a Bfly-4 block
  - a 2.5x area reduction!

Parameterization: An n-stage synchronous pipeline

```
Vector#(n, Reg#(t)) sReg <- replicateM(mkReg(Invalid));
rule sync-pipeline (True);
  if (inQ.notEmpty())
    begin (sReg[1]) <= Valid f(1,inQ.first()); inq.deq(); end
  else (sReg[1]) <= Invalid;
  for (Integer stage = 1; stage < n-1; stage = stage+1)
    case (sReg[stage]) matches
      tagged Valid .sx: (sReg[stage+1]) <= Valid f(stage+1,sx);
      tagged Invalid : (sReg[stage+1]) <= Invalid; endcase
  case (sReg[n-1]) matches
    tagged Valid .sx: outQ.enq(f(n,sx)); endcase
endrule
```

n and stage are static parameters
Syntax: Vector of Registers

Register
- suppose \( x \) and \( y \) are both of type Reg. Then
  \( x <= y \) means \( x._\text{write}(y._\text{read}) \)

Vector of (say) Int
- \( x[i] \) means \( \text{sel}(x, i) \)
- \( x[i] = y[j] \) means \( x = \text{update}(x, i, \text{sel}(y, j)) \)

Vector of Registers
- \( x[i] <= y[j] \) does not work. The parser thinks it means
  \( (\text{sel}(x, i)._\text{read}).\text{write}(\text{sel}(y, j)._\text{read}) \),
  which will not type check
- \( (x[i]) <= y[j] \) does work!

Action Value methods

- Value method: Only reads the state; does not affect it
  - e.g. \( \text{fifo.first()} \)

- Action method: Affects the state but does not return a value
  - e.g. \( \text{fifo.deq}(), \text{fifo.enq}(x), \text{fifo.clear}() \)

- Action Value method: Returns a value but also affects the state
  - e.g. \( \text{fifo.pop}() \)
  - syntax: \( x \leftarrow \text{fifo.pop}() \)

This use of \( \leftarrow \) is not to be confused with module instantiation \( \text{reg} \leftarrow \text{mkRegU}() \)
One-Element FIFO

```verilog
module mkFIFO1 (FIFO#(t));
    Reg#(t)    data  <- mkRegU();
    Reg#(Bool) full  <- mkReg(False);
    method Action enq(t x) if (!full);
        full <= True;   data <= x;
    endmethod
    method Action deq() if (full);
        full <= False;
    endmethod
    method t first() if (full);
        return (data);
    endmethod
    method Action clear();
        full <= False;
    endmethod
endmodule
```

```verilog
method ActionValue#(t) pop() if (full);
    full <= False; return (data);
endmethod
```

A simple non-pipelled processor

Another example to illustrate simple rules and tagged unions (also to help you with Lab 2)
Instruction set

typedef enum {R0;R1;R2;...;R31} RName;

typedef union tagged {
    struct {RName dst; RName src1; RName src2;} Add;
    struct {RName cond; RName addr;}            Bz;
    struct {RName dst; RName addr;}             Load;
    struct {RName src; RName addr;}             Store;
} {
    Instr deriving (Bits, Eq);
}

typedef Bit#(32) Iaddress;

typedef Bit#(32) Daddress;

typedef Bit#(32) Value;

An instruction set can be implemented using many different microarchitectures

Tagged Unions: *Bit Representation*

typedef union tagged {
    struct {RName dst; RName src1; RName src2;} Add;
    struct {RName cond; RName addr;}            Bz;
    struct {RName dst; RName addr;}             Load;
    struct {RName src; RName addr;}             Store;
} {
    Instr deriving (Bits, Eq);
}

00 dst src1 src2
01          cond addr
10           dst addr
11           src addr

Automatically derived representation; can be customized by the user written pack and unpack functions
Non-pipelined Processor

module mkCPU#(Mem iMem, Mem dMem)();
  Reg#(Iaddress) pc <- mkReg(0);
  RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
  Instr instr = iMem.read(pc);
  Iaddress predIa = pc + 1;
  rule fetch_Execute...
endmodule

Non-pipelined processor rule

rule fetch_Execute (True);
case (instr) matches
  tagged Add {dst:.rd,src1:.ra,src2:.rb}:
  begin
    rf.upd(rf[ra]+rf[rb]);
    pc <= predIa
  end
  tagged Bz {cond:.rc,addr:.ra}:
  begin
    pc <= (rf[rc]==0) ? rf[ra] : predIa;
  end
  tagged Load {dest:.rd,addr:.ra}:
  begin
    rf.upd(rf[ra]);
    pc <= predIa;
  end
  tagged Store {value:.rv,addr:.ra}:
  begin
    dMem.write(rf[ra],rf[rv]);
    pc <= predIa;
  endcase
endrule
Syntax: RegFile vs Vectors

- A RegFile (register file) has a different type than a Vector of Registers
- A RegFile is a library module and has one write and multiple read methods
  - rf.sub(i) returns the value of the ith register
  - rf.upd(i, v) updates the ith register
- It is created by mkRegFile(lowerIndex, upperIndex);
  the type of the contents is inferred from the LHS declarations.

Memory Interface

- “magic memory” responds to a read request in the same cycle and updates the memory at the end of the cycle for a write request
- In a realistic memory, a read request typically takes many cycles
  - Synchronous memory responds in a fixed number of cycles
  - A pipelined memory holds up to n requests and processes requests in a FIFO manner (n is the raw latency of accessing the memory)
  - Request/Response type of memory interface “decouples” the user from the memory
RAMs: Synchronous vs Asynchronous view

- Basic memory components are "synchronous":
  - Present a read-address \( A_J \) on clock \( J \)
  - Data \( D_J \) arrives on clock \( J+N \)
  - If you don't "catch" \( D_J \) on clock \( J+N \), it may be lost, i.e., data \( D_{J+1} \) may arrive on clock \( J+1+N \)

- This kind of synchronicity can pervade the design and cause complications

Request-Response Interface for RAMs

```verilog
interface Mem#(type addr_T, type data_T);    
  method Action req(MemReq#(addr_T, data_T) x);  
  method ActionValue#(MemResp#(data_T)) resp();  
endinterface

typedef union tagged 
  {addrT Read; 
   Tuple2#(addrT, dataT) Write; 
  }MemReq#(type addrT, type dataT);
```
Non-pipelined Processor with decoupled memory

An instruction will take two or three cycles: Fetch-Execute, Fetch-Execute-WB

```verilog
module mkCPU#(Mem iMem, Mem dMem);
    Reg#(Iaddress) pc <- mkReg(0);
    RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
    Reg#(Stage) s <- mkReg(Fetch); Reg#(RName) d <- mkRegU();
    Iaddress predIa = pc + 1;
    rule fetch (s==Fetch);
        iMem.req(Read pc); s <= Execute endrule
    rule execute (s==Execute);
        Instr instr <- mem.resp();
        case (instr)
            matches
                tagged Add {dst:.rd, src1:.ra, src2:.rb}:
                    begin
                        rf.upd(rd, rf[ra]+rf[rb]);
                        pc <= predIa; s <= Fetch
                    end
                tagged Bz {cond:.rc, addr:.ra}:
                    begin
                        pc <= (rf[rc]==0) ? rf[ra] : predIa;
                        s <= Fetch
                    end
                tagged Load {dest:.rd, addr:.ra}:
                    begin
                        dMem.req(Read rf[ra]);
                        pc <= predIa; s <= Writeback; d <= rd;
                    end
                tagged Store {value:.rv, addr:.ra}:
                    begin
                        dMem.req(Write tuple2(rf[ra],rf[rv]));
                        pc <= predIa; s <= Writeback
                    end
            endcase
        endrule
endmodule
```

Decoupled processor-memory

Execute rule

```verilog
rule execute (s==Execute);
    Instr instr <- mem.resp();
    case (instr)
        matches
            tagged Add {dst:.rd, src1:.ra, src2:.rb}:
                begin
                    rf.upd(rd, rf[ra]+rf[rb]);
                    pc <= predIa; s <= Fetch
                end
            tagged Bz {cond:.rc, addr:.ra}:
                begin
                    pc <= (rf[rc]==0) ? rf[ra] : predIa;
                    s <= Fetch
                end
            tagged Load {dest:.rd, addr:.ra}:
                begin
                    dMem.req(Read rf[ra]);
                    pc <= predIa; s <= Writeback; d <= rd;
                end
            tagged Store {value:.rv, addr:.ra}:
                begin
                    dMem.req(Write tuple2(rf[ra],rf[rv]));
                    pc <= predIa; s <= Writeback
                end
        endcase
endrule
```
Load/Store Writeback rule

rule write-back (s==Writeback);
    DmemResp resp <- dMem.resp();
    case (resp) matches
        tagged LoadResp .v: rf.upd(d, v);
    endcase
    s <= Fetch;
endrule

Next time – microarchitectural exploration via IP lookup