Simple Inelastic and Folded Pipelines

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 14, 2011

Pipelining a block

Clock: \( C < P \neq FP \)  
Area: \( FP < C < P \)  
Throughput: \( FP < C < P \)
Inelastic Pipeline

```
rule sync-pipeline (True);
    if (inQ.notEmpty())
    begin sReg1 <= tagged Valid f1(inQ.first()); inQ.deq(); end
else sReg1 <= tagged Invalid;
    case (sReg1)
        tagged Valid .sx1: sReg2 <= tagged Valid f2(sx1);
        tagged Invalid: sReg2 <= tagged Invalid; endcase
    endrule
```

When is this rule enabled?

```
rule sync-pipeline (True);
    if (inQ.notEmpty())
    begin sReg1 <= tagged Valid f0(inQ.first()); inQ.deq(); end
else sReg1 <= tagged Invalid;
    case (sReg1)
        tagged Valid .sx1: sReg2 <= tagged Valid f1(sx1);
        tagged Invalid: sReg2 <= tagged Invalid; endcase
    endrule
```

Pipelining a block

Clock: \( C < P \approx FP \)  
Area: \( FP < C < P \)  
Throughput: \( FP < C < P \)

Folded pipeline

```
rule folded-pipeline (True);
  if (stage==0)
    begin sxIn= inQ.first(); inQ.deq(); end
  else sxIn = sReg;
  sxOut = f(stage, sxIn);
  if (stage==n-1) outQ.enq(sxOut);
  else sReg <= sxOut;
  stage <= (stage==n-1)? 0 : stage+1;
endrule
```

Need type declarations for sxIn and sxOut
Superfolded pipeline

One Bfly-4 case

- $f$ will be invoked for 48 dynamic values of stage
  - each invocation will modify 4 numbers in sReg
  - after 16 invocations a permutation would be done on the whole sReg

```verbatim
Superfolded pipeline: stage function f

function Vector#(64, Complex) stage_f (Bit#(2) stage, Vector#(64, Complex) stage_in);
begin
  for (Integer i = 0; i < 16; i = i + 1)
    begin
      Bit#(2) stage
      Integer idx = i * 4;
      let twid = getTwiddle(stage, fromInteger(i));
      let y = bfly4(twid, stage_in[idx:idx+3]);
      stage_temp[idx] = y[0]; stage_temp[idx+1] = y[1];
      stage_temp[idx+2] = y[2]; stage_temp[idx+3] = y[3];
    end
  //Permutation
  for (Integer i = 0; i < 64; i = i + 1)
    stage_out[i] = stage_temp[permute[i]];
end
return(stage_out);
```
Code for the Superfolded pipeline stage function

```
Function Vector#{64, Complex} f 
    (Bit#{6} stagei, Vector#{64, Complex} stage_in);
    let i = stagei `mod` 16;
    let twid = getTwiddle(stagei `div` 16, i);
    let y = bfly4(twid, stage_in[i:i+3]);
    let stage_temp = stage_in;
    stage_temp[i] = y[0];
    stage_temp[i+1] = y[1];
    stage_temp[i+2] = y[2];
    stage_temp[i+3] = y[3];
    let stage_out = stage_temp;
    if (i == 15)
        for (Integer i = 0; i < 64; i = i + 1)
            stage_out[i] = stage_temp[permute[i]];
    return(stage_out);
endfunction
```

Folded pipeline: stage function f

The Twiddle constants can be expressed in a table or in a case or nested case expression.

The rest of stage_f, i.e. Bfly-4s and permutations (shared)
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>Combinational</td>
<td><strong>4.91</strong></td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Folded (16 Bfly-4s)</td>
<td><strong>3.97</strong></td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Super-Folded (8 Bfly-4s)</td>
<td>3.69</td>
<td>06</td>
<td>1.5 MHz</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>

All these designs were done in less than 24 hours!

Why are the areas so similar

Folding should have given a 3x improvement in IFFT area

Elastic pipeline

Use FIFOs instead of pipeline registers

rule stage1 (True);
    fifo1.enq(f1(inQ.first()));
inQ.deq();
endrule

rule stage2 (True);
    fifo2.enq(f2(fifo1.first()));
fifo1.deq();
endrule

rule stage3 (True);
    outQ.enq(f3(fifo2.first()));
fifo2.deq();
endrule

Firing conditions?
Can tokens be left inside the pipeline?
No Maybe types?
Easier to write?
Can all three rules fire concurrently?
What behavior do we want?

![Diagram of FIFO](image)

<table>
<thead>
<tr>
<th>inQ</th>
<th>fifo1</th>
<th>fifo2</th>
<th>outQ</th>
<th>rule1</th>
<th>rule2</th>
<th>rule3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>NE,NF</td>
<td>NE,NF</td>
<td>NF</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NE</td>
<td>NE,NF</td>
<td>NE,NF</td>
<td>F</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NE</td>
<td>NE,NF</td>
<td>NE,F</td>
<td>NF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NE</td>
<td>NE,NF</td>
<td>NE,F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
```

---


January 18, 2011  http://csg.csail.mit.edu/SNU  L4-16
Concurrency when the FIFOs do not permit concurrent enq and deq

Inelastic vs Elastic Pipelines

- **In an Inelastic pipeline:**
  - typically only one rule; the designer controls precisely which activities go on in parallel
  - *downside:* The rule can get too complicated -- easy to make a mistake; difficult to make changes

- **In an Elastic pipeline:**
  - several smaller rules, each easy to write, easier to make changes
  - *downside:* sometimes rules do not fire concurrently when they should
The tension

- If multiple rules never fire in the same cycle then the machine can hardly be called a pipelined machine.
- If all rules fire in parallel every cycle when they are enabled, then, in general, wrong results can be produced.

More on this in the next lecture.

Language notes

- Pattern matching syntax
- Vector syntax
- Implicit conditions
- Static vs dynamic expression
Pattern-matching: A convenient way to extract datastructure components

definition union tagged {
  void Invalid;
  t Valid;
} Maybe#(type t);

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

i if (m matches (Valid .x) && (x > 10))
- The &&& is a conjunction, and allows pattern-variables to come into scope from left to right

Syntax: Vector of Registers
- Register
  - suppose x and y are both of type Reg. Then
    x <= y means x._write(y._read())

- Vector of Int
  - x[i] means sel(x,i)
  - x[i] = y[j] means x = update(x, i, sel(y,j))

- Vector of Registers
  - x[i] <= y[j] does not work. The parser thinks it means
    sel(x,i)._read).write(sel(y,j)._read), which will not type check
  - (x[i]) <= y[j] parses as
    sel(x,i).write(sel(y,j)._read), and works correctly

Don't ask me why
Making guards explicit

```plaintext
rule recirculate (True);
  if (p) fifo.enq(8);
  r <= 7;
endrule

rule recirculate ((p && fifo.enqG) || !p);
  if (p) fifo.enqB(8);
  r <= 7;
endrule
```

Effectively, all implicit conditions (guards) are lifted and conjoined to the rule guard

Implicit guards (conditions)

- Rule
  ```plaintext
  rule <name> (<guard>); <action>; endrule
  ```

- where
  ```plaintext
  <action> ::= r <= <exp>
  while m.gG(<exp>) when m.gG
  m.g(<exp>)
  if (<exp>) <action> endif
  <action> ; <action>
  ```
Guards vs If’s

- A guard on one action of a parallel group of actions affects every action within the group
  \[(a1 \text{ when } p1); (a2 \text{ when } p2)\]
  \[\Rightarrow (a1; a2) \text{ when } (p1 \&\& p2)\]
- A condition of a Conditional action only affects the actions within the scope of the conditional action
  \[(\text{if } (p1) a1); a2\]
  \[p1 \text{ has no effect on } a2 \ldots\]
- Mixing ifs and whens
  \[(\text{if } (p) (a1 \text{ when } q)); a2\]
  \[\equiv ((\text{if } (p) a1); a2) \text{ when } ((p \&\& q) | !p)\]

Static vs dynamic expressions

- Expressions that can be evaluated at compile time will be evaluated at compile-time
  - \[3+4 \Rightarrow 7\]
- Some expressions do not have run-time representations and must be evaluated away at compile time; an error will occur if the compile-time evaluation does not succeed
  - Integers, reals, loops, lists, functions, …
Generalization: \(n\)-stage pipeline

\[
\begin{align*}
\text{rule} \ & \text{sync-pipeline (True);} \\
\text{if} \ & \text{(inQ.isEmpty())} \\
\ & \begin{cases} 
\text{sReg[0]} & \leftarrow \text{tagged Valid} \\
\text{f(1,inQ.first());inQ.deq();} & \text{end} \\
\text{else} & \text{sReg[0]} & \leftarrow \text{tagged Invalid;} \\
\text{for} \ & \text{Integer i = 1; i < n-1; i=i+1 begin} \\
\ & \begin{cases} 
\text{case} \ & \text{(sReg[i-1]) matches} \\
\text{tagged Valid .sx: sReg[i] & \leftarrow \text{tagged Valid f(i-1,sx);} \\
\text{tagged Invalid: sReg[i] & \leftarrow \text{tagged Invalid; endcase end}} \\
\text{case} \ & \text{(sReg[n-2]) matches} \\
\text{tagged Valid .sx: outQ.enq(f(n-1,sx));} & \text{endcase} \\
\text{endrule}
\end{cases}
\end{cases}
\end{align*}
\]

February 14, 2011 L04-27
http://csg.csail.mit.edu/6.375

Next lecture
Concurrent analysis