Simple Synchronous Pipelines

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Pipelining a block

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

```
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;
endcase end
```

Both Registers hold values of Maybe type

case (sReg2) matches
  tagged Valid .sx2: outQ.enq(f3(sx2));
endcase end
endrule

Generalization: $n$-stage pipeline

```
rule sync-pipeline (True);
if (inQ.notEmpty())
begin sReg[1]<= Valid f(1,inQ.first());inQ.deq();end
else sReg[1]<= Invalid;
for(Integer i = 2; i < n; i=i+1) begin
  case (sReg[i-1]) matches
    tagged Valid .sx: sReg[i] <= Valid f(i-1,sx);
    tagged Invalid: sReg[i] <= Invalid; endcase end
  case (sReg[n]) matches
    tagged Valid .sx: outQ.enq(f(n,sx)); 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
Folded pipeline: stage function f

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

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
Superfolded pipeline: stage function $f$

```plaintext
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

```plaintext
function SVector#(64, Complex) f
  (Bit#(6) stagei, SVector#(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
```
### 802.11a Transmitter

**[MEMOCODE 2006] Dave, Gerding, Pellauer, Arvind**

<table>
<thead>
<tr>
<th>Design Block</th>
<th>Lines of Code (BSV)</th>
<th>Relative Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>49</td>
<td>0%</td>
</tr>
<tr>
<td>Scrambler</td>
<td>40</td>
<td>0%</td>
</tr>
<tr>
<td>Conv. Encoder</td>
<td>113</td>
<td>0%</td>
</tr>
<tr>
<td>Interleaver</td>
<td>76</td>
<td>1%</td>
</tr>
<tr>
<td>Mapper</td>
<td>112</td>
<td>11%</td>
</tr>
<tr>
<td>IFFT</td>
<td>95</td>
<td>85%</td>
</tr>
<tr>
<td>Cyc. Extender</td>
<td>23</td>
<td>3%</td>
</tr>
</tbody>
</table>

Complex arithmetic libraries constitute another 200 lines of code.

---

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

- **Pipelined**: 5.25 mm², Throughput 04 CLks/sym, Min. Freq Required 1.0 MHz
- **Combinational**: 4.91 mm², Throughput 04 CLks/sym, Min. Freq Required 1.0 MHz
- **Folded (16 Bfly-4s)**: 3.97 mm², Throughput 04 CLks/sym, Min. Freq Required 1.0 MHz
- **Super-Folded (8 Bfly-4s)**: 3.69 mm², Throughput 06 CLks/sym, Min. Freq Required 1.5 MHz
- **SF(4 Bfly-4s)**: 2.45 mm², Throughput 12 CLks/sym, Min. Freq Required 3.0 MHz
- **SF(2 Bfly-4s)**: 1.84 mm², Throughput 24 CLks/sym, Min. Freq Required 6.0 MHz
- **SF (1 Bfly4)**: 1.52 mm², Throughput 48 CLks/sym, Min. Freq Required 12 MHz

All these designs were done in less than 24 hours! The same source code was used for all designs.

TSMC .18 micron; numbers reported are before place and route.

---

February 16, 2010  
http://csg.csail.mit.edu/6.375
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!

Language notes

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

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))
  x will get bound to the appropriate part of m

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.enq) || !p);
    if (p) fifo.enq(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>
                m.gB(<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:
  
  \[(a_1 \text{ when } p_1); (a_2 \text{ when } p_2)\]
  
  \[\Rightarrow (a_1; a_2) \text{ when } (p_1 \& \& p_2)\]

- A condition of a Conditional action only affects the actions within the scope of the conditional action:
  
  \[(\text{if } (p_1) a_1)\}; a_2\]
  
  \[p_1 \text{ has no effect on } a_2 \ldots\]

- Mixing ifs and whens:
  
  \[(\text{if } (p) (a_1 \text{ when } q)) \}; a_2\]
  
  \[\equiv ((\text{if } (p) a_1); a_2) \text{ when } ((p \& \& q) | !p)\]

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, ...
next time

Asynchronous pipelines...