Folded and Pipelined Sequential circuits

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology
Expressing a loop using registers

```c
int s = s0;
while (p(s)) {
    s = f(s);
}
return s;
```

- Such a loop cannot be implemented by unfolding because the number of iterations is input-data dependent
- A register is needed to hold the value of `s` from one iteration to the next
- `s` has to be initialized when the computation starts, and updated every cycle until the computation terminates

```
sel = start
en = start | notDone
```
Expressing a loop in BSV

- When a rule executes:
  - the register s is read at the beginning of a clock cycle
  - computations to evaluate the next value of the register and the \( s_{en} \) are performed
  - If \( s_{en} \) is True then s is updated at the end of the clock cycle
- A mux is needed to initialize the register

How should this circuit be packaged for proper use?

```
Reg#(n) s <- mkRegU();
rule step;
  if (p(s)) s <= f(s);
endrule
```

```
select = start
en = start | notDone
```
Packaging a computation as a Latency-Insensitive Module

interface GFMI#(n);
  method Action start (Bit#(n) x);
  method ActionValue#(Bit#(n)) getResult;
endinterface

module mkF (GFMI#(n));
  Reg#(n) s <- mkRegU();
  Reg#(Bool) busy <- mkReg(False);
  rule step if (p(s))&&busy;
    s <= f(s);
  endrule
  method Action start(Bit#(n) x) if (!busy);
    s <= x; busy <= True;
  endmethod
  method ActionValue Bit#(n) getResult if (!p(s) & & busy);
    busy <= False; return s;
  endmethod
endmodule

The user of this module does not know how long it takes to execute the loop.
Combinational 32-bit multiply

```haskell
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
    Bit#(32) tp = 0;
    Bit#(32) prod = 0;
    for (Integer i = 0; i < 32; i = i+1)
        begin
            Bit#(32) m = (a[i]==0)? 0 : b;
            Bit#(33) sum = add32(m,tp,0);
            prod[i]    = sum[0];
            tp         = sum[32:1];
        end
    return {tp,prod};
endfunction
```

- We can reuse the same add32 circuit if we store the partial results, e.g., sum, in a `register`
- Need registers to hold a, b, tp, prod and i
- Update the registers every cycle until we are done

This circuit uses 32 add32 circuits

Lot of gates!
The user of this module does not know how long it takes to execute a particular multiply.
Multiply Module

Module mkMultiply (Multiply);
    Reg#(Bit#(32)) a<-mkRegU(); Reg#(Bit#(32)) b<-mkRegU();
    Reg#(Bit#(32)) prod <-mkRegU(); Reg#(Bit#(32)) tp <- mkReg(0);
    Reg#(Bit#(6)) i <- mkReg(32);
    Reg#(Bool) busy <- mkReg(False);

rule mulStep if (i < 32);
    Bit#(32) m = (a[i]==0)? 0 : b;
    Bit#(33) sum = add32(m,tp,0);
    prod[i] <= sum[0];
    tp <= sum[32:1];
    i <= i+1;
endrule; like the loop body in the combinational version

method Action start(Bit#(32) x, Bit#(32) y) if (!busy);
    a <= x; b <= y; busy <= True; i <= 0; endmethod

method ActionValue Bit#(64) getResult if ((i==32) && busy);
    busy <= False; return {tp,prod}; endmethod

We use a 6-bit i register and initialize it to 32 to make sure that the rule has no effect until i is set to some value < 32
Dynamic selection requires a mux

When the selection indices follow a regular pattern then it is cheaper to use a shift operator (no gates!) instead of a mux.
Replacing repeated selections by shifts

```vhdl
rule mulStep if (i < 32);
    Bit#(32) m = (a[0]==0)? 0 : b;
a <= a >> 1;
    Bit#(33) sum = add32(m,tp,0);
    prod <= {sum[0], prod[31:1]};
    tp <= sum[32:1];
i <= i+1;
endrule
```
Circuit for Sequential Multiply

\[
\begin{align*}
    bIn & \quad aIn \\
    \downarrow & \quad \downarrow \\
    b & \quad a \\
    \downarrow & \quad \downarrow \\
    \text{add} & \quad \text{add} \\
    \downarrow & \quad \downarrow \\
    \text{prod} & \quad \text{prod} \\
    \downarrow & \quad \downarrow \\
    \text{result (high)} & \quad \text{result (low)} \\
\end{align*}
\]

\[
s_1 = \text{start\_en} \\
\]

\[
s_2 = \text{start\_en} \mid \neg \text{done}
\]
Multiply Module

\[ s_1 = \text{start}_{\text{en}} \]
\[ s_2 = \text{start}_{\text{en}} | !\text{done} \]
\[ s_3 = \text{start}_{\text{en}} | \text{getResult}_{\text{en}} \]

\[ \text{start}_{\text{rdy}} = !\text{busy} \]
\[ \text{getResult}_{\text{rdy}} = \text{busy} \& \text{done} \]

\[ \text{t}_{\text{p}} \]
\[ \text{prod} \]

[Diagram showing the flow of data and control signals in the Multiply Module]
Sequential Multiply analysis

- Number of add32 circuits has been reduced from 31 to one, though some registers and muxes have been added
- The longest combinational path has been reduced from 62 FAs to one add32 plus a few muxes
- The sequential circuit will take 31 clock cycles to compute an answer
Suppose a module is required to process inputs in the FIFO order
- Internally the module processes several inputs simultaneously and different inputs take different amount of time
- Completion buffer is used to restore the order in which the processing of inputs was done
Completion buffer: Implementation

- A circular buffer with two pointers iidx and ridx, and a counter cnt
- Each data element has a valid bit associated with it

```verilog
module mkCompletionBuffer(CompletionBuffer#(t));
    Vector#(32, Reg#(Bool)) cbv <- replicateM(mkReg(False));
    Vector#(32, Reg#(t)) cbData <- replicateM(mkRegU());
    Reg#(Bit#(5)) iidx <- mkReg(0);
    Reg#(Bit#(5)) ridx <- mkReg(0);
    Reg#(Bit#(6)) cnt <- mkReg(0);

    rules and methods...
endmodule
```
method ActionValue#(Bit#(5)) getToken() if (cnt < 32);
    cbv[iidx] <= False;
    iidx <= (iidx==31) ? 0 : iidx + 1;
    cnt <= cnt + 1;
    return iidx;
endmethod

method Action put(Token idx, t data);
    cbData[idx] <= data;
    cbv[idx] <= True;
endmethod

method ActionValue#(t) getResult() if ((cnt > 0) && (cbv[ridx]));
    cbv[ridx] <= False;
    ridx <= (ridx==31) ? 0 : ridx + 1;
    cnt <= cnt - 1;
    return cbData[ridx];
endmethod
Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)

Reuse a block, multicycle

Clock: $C < P \approx F$

Area: $F < C < P$

Throughput: $F < C < P$
Rules for Pipeline

```verbatim
rule stage1;
    fifo1.enq(f0(inQ.first));
    inQ.deq;
endrule

rule stage2;
    fifo2.enq(f1(fifo1.first));
    fifo1.deq;
endrule

rule stage3;
    outQ.enq(f2(fifo2.first));
    fifo2.deq;
endrule
```
rule folded-pipeline (True);
let sxIn = ?;
if (stage==0)
  begin sxIn= inQ.first(); inQ.deq(); end
else    sxIn= sReg;
let sxOut = f(stage,sxIn);
if (stage==n-1) outQ.enq(sxOut);
else    sReg <= sxOut;
stage <= (stage==n-1)? 0 : stage+1;
endrule

notice stage is a dynamic parameter now!
Lot of area and long combinational delay

Pipelining: Reduces critical-path delay. Increases throughput by evaluating multiple items in parallel
Combinational IFFT

- **Lot of area and long combinational delay**
- *Pipelining*: Reduces critical-path delay. Increases throughput by evaluating multiple items in parallel
- *Folded or multi-cycle Circuit*: Save area and reduces the combinational delay but makes throughput per clock cycle worse
Folded IFFT: Reusing the Stage computation

- input signals: in0, in1, in2, in3, in4, ..., in63
- output signals: out0, out1, out2, out3, out4, ..., out63
- Bfly4 blocks
- Permute block
- Stage Counter

September 13, 2019
http://csg.csail.mit.edu/6.375
L05-21
function Vector#(64, Complex#(n)) stage_f
  (Bit#(2) stage, Vector#(64, Complex#(n)) stage_in);
Vector#(64, Complex#(n)) stage_temp, stage_out;
  for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      Vector#(4, Complex#(n)) x;
      x[0] = stage_in[idx]; x[1] = stage_in[idx+1];
      x[2] = stage_in[idx+2]; x[3] = stage_in[idx+3];
      let twid = getTwiddle(stage, fromInteger(i));
      let y = bfly4(twid, x);
      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]];
return(stage_out);
endfunction
Higher-order functions: Stage functions $f_1$, $f_2$ and $f_3$

function $f_0(x) = \text{stage}_f(0,x)$;

function $f_1(x) = \text{stage}_f(1,x)$;

function $f_2(x) = \text{stage}_f(2,x)$;

What is the type of $f_0(x)$?

function Vector#(64, Complex) $f_0$

(Vector#(64, Complex) $x$);
This is the code for IFFT if you use stage_f function for functions for f0, f1 and f2!

from slide 17
Code for Folded IFFT

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

This is the code for IFFT if you use stage_f function for functions for f!

from slide 18
The Twiddle constants can be expressed as a table or using a case expression.
Superfolded IFFT: Just one Bfly-4 node!

- 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
Superfolding IFFT: stage function f

```plaintext
function Vector#(64, Complex) stage_f
    (Bit#(2) stage, Vector#(64, Complex) stage_in);
    Vector#(64, Complex#(n)) stage_temp, stage_out;
    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]];
    return(stage_out);
endfunction
```

should be done only when i=15
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 idx = i*4;
  let y = bfly4(twid, stage_in[idx:idx+3]);
  let stage_temp = stage_in;
  stage_temp[idx] = y[0];
  stage_temp[idx+1] = y[1];
  stage_temp[idx+2] = y[2];
  stage_temp[idx+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 Overview

Controller → Scrambler → Encoder → Interleaver → Mapper → IFFT → Cyclic Extend

- Controller: Scrambles and encodes data.
- Scrambler: Scrambles the data.
- Encoder: Encodes the scrambled data.
- Interleaver: Interleaves the encoded data.
- Mapper: Mappers the interleaved data.
- IFFT: Converts 64 frequency domain complex numbers into 64 time domain complex numbers.
- Cyclic Extend: Extends the data.

Must produce one OFDM symbol every 4 msec.

Depending upon the transmission rate, consumes 1, 2, or 4 tokens to produce one OFDM symbol.

IFFT Transforms 64 (frequency domain) complex numbers into 64 (time domain) complex numbers.

One OFDM symbol (64 Complex Numbers) accounts for 85% area.

September 13, 2019
http://csg.csail.mit.edu/6.375
### 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)

<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>4.91</td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Folded (16 Bfly-4s)</td>
<td>3.97</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>

The same source code

All these designs were done in less than 24 hours!

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!
Syntax: Vector of Registers

- **Register**
  - suppose $x$ and $y$ are both of type Reg. Then
    \[ x \leq y \text{ means } x._\text{write}(y._\text{read}()) \]

- **Vector of 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**
  - $x[i] \leq y[j]$ 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
  - $(x[i]) \leq y[j]$ parses as $\text{sel}(x,i)._\text{write}($sel(y,j)._\text{read}$)$, and works correctly

*Don’t ask me why*