Folded Combinational Circuits as an example of Sequential Circuits

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 19, 2013
http://csg.csail.mit.edu/6.375

Multiplication by repeated addition

\[
\begin{array}{c}
\text{b Multiplicand} \\
\text{a Multiplier} \\
1101 \\
\times 1011
\end{array}
\]

\[
\begin{array}{c}
1101 \\
+ 1101 \\
+ 0000 \\
+ 1101 \\
\text{10001111} \quad (143)
\end{array}
\]

\[
m_i = (a[i] == 0) \? 0 : b;
\]
Combinational 32-bit multiply

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

Design issues with combinational multiply

- Lot of hardware
  - 32-bit multiply uses 31 add32 circuits
- Long chains of gates
  - 32-bit ripple carry adder has a 31-long chain of gates
  - 32-bit multiply has 31 ripple carry adders in sequence!

The speed of a combinational circuit is determined by its longest input-to-output path

Can we do better?
We can reuse the same add32 circuit if we can store the partial results in some storage device, e.g., register.

Combinational circuits

Such circuits have no cycles (feedback) or state elements.
A simple synchronous state element

**Edge-Triggered Flip-flop**

Data is sampled at the rising edge of the clock

---

Flip-flops with Write Enables

Data is captured only if EN is on

dangerous!
Registers

Register: A group of flip-flops with a common clock and enable

Register file: A group of registers with a common clock, input and output port(s)

We can build useful and compact circuits using registers

Circuits containing state elements are called sequential circuits
Expressing a loop using registers

```c
int s = s0;
for (int i = 0; i < 32; i = i+1) {
    s = f(s);
}
return s;               C-code
```

We need two registers to hold s and i values from one iteration to the next. These registers are initialized when the computation starts and updated every cycle until the computation terminates.

We need two registers to hold s and i values from one iteration to the next. These registers are initialized when the computation starts and updated every cycle until the computation terminates.

```
+1 sel
+1

0

i

< 32 notDone

f s0

en

s

en

sel

notDone

en = start | notDone
```

Expressing sequential circuits in BSV

- Sequential circuits, unlike combinational circuits, are not expressed structurally (as wiring diagrams) in BSV.
- For sequential circuits a designer defines:
  - **State elements** by instantiating modules
    ```
    Reg#(Bit#(32)) s <- mkRegU();
    Reg#(Bit#(6))  i <- mkReg(32);
    ```
  - **Rules** which define how state is to be transformed atomically
    ```
rule step if (i < 32);
    s <= f(s);
    i <= i+1;
endrule
```
Rule Execution

- When a rule executes:
  - all the registers are read at the beginning of a clock cycle
  - the guard and computations to evaluate the next value of the registers are performed
  - at the end of the clock cycle registers are updated iff the guard is true
- Muxes are needed to initialize the registers

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6)) i <- mkReg(32);

rule step if (i < 32);
    s <= f(s);
    i <= i+1;
endrule
```

Multiply using registers

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

Need registers to hold a, b, tp, prod and i

Update the registers every cycle until we are done
Sequential multiply

Reg#(Bit#(32)) a <- mkRegU();
Reg#(Bit#(32)) b <- mkRegU();
Reg#(Bit#(32)) prod <- mkRegU();
Reg#(Bit#(32)) tp <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);

rule mulStep if (i < 32);
let m = (a[i]==0)? 0 : b;
let sum = add32(m, tp, 0);
prod[i] <= sum[0];
tp <= sum[32:1];
i <= i+1;
endrule

Dynamic selection requires a mux

when the selection indices are regular then it is better to use a shift operator (no gates!)
Replacing repeated selections by shifts

Reg#(Bit#(32)) a <- mkRegU();
Reg#(Bit#(32)) b <- mkRegU();
Reg#(Bit#(32)) prod <- mkRegU();
Reg#(Bit#(32)) tp <- mkRegU();
Reg#(Bit#(6)) i <- mkReg(32);

rule mulStep if (i < 32);
let m = (a[0]==0)? 0 : b;
    a <= a >> 1;
let sum = add32(m, tp, 0);
    prod <= {sum[0], (prod >> 1)[30:0]};
    tp <= sum[32:1];
    i <= i+1;
endrule

Circuit for Sequential Multiply

s1 = start_en
s2 = start_en | !done
Circuit 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 31 serial add32’s to one add32 plus a few muxes.
- The sequential circuit will take 31 clock cycles to compute an answer.

Combinational IFFT

Reuse the same circuit three times to reduce area.
BSV Code for stage_f

```haskell
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
```

twid's are mathematically derivable constants

Higher-order functions:
Stage functions f1, f2 and f3

```haskell
function f0(x)= stage_f(0,x);
function f1(x)= stage_f(1,x);
function f2(x)= stage_f(2,x);
```

What is the type of f0(x) ?

```haskell
function Vector#(64, Complex) f0
  (Vector#(64, Complex) x);
```
Folded Combinational Ckts

```
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!

no for-loop

Shared Circuit

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)
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 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
```

Bit#(2+4) (stage,i) should be done only when i=15
Code for the Superfolded stage function

```pascal
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
```

One Bfly-4 case

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!