Folded Combinational Circuits as an example of Sequential Circuits

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 10, 2016 http://csg.csail.mit.edu/6.375

Folding large combinational circuits

- A common way to implement large combinational circuits is by folding where registers hold the state from one iteration to the next
  - Implementing imperative loops
  - Multiplication
  - IFFT
Flip flop: The basic building block of Sequential Circuits

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
**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)

---

**Expressing a loop using registers**

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

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.
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.
Multiplication by repeated addition

b Multiplier: \[1101\] (13)
a Multiplier: \[1011\] (11)

\[
\begin{align*}
\text{tp} & : 0000 \\
m0 & + 1101 \\
\text{tp} & : 01101 \\
m1 & + 1101 \\
\text{tp} & : 100111 \\
m2 & + 0000 \\
\text{tp} & : 0100111 \\
m3 & + 1101 \\
\text{tp} & : 10001111 (143)
\end{align*}
\]

\[m_i = (a[i]==0)? 0 : b;\]

Combinational 32-bit multiply

```plaintext
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:i] = sum[0];
    tp = sum[32:1];
end
return {tp,prod};
endfunction
```

Combinational multiply uses 31 add32 circuits

We can reuse the same add32 circuit if we store the partial results in a register
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! Total delay?

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

Can we do better?

Multiply using registers

```plaintext
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
        Bit#(32) m = (a[i] == 0)? 0 : b;
        Bit#(33) sum = add32(m, tp, 0);
        prod[i:i] = sum[0];
        tp = sum[32:1];
    end
    return {tp, prod};
endfunction
```

Need registers to hold a, b, ...?

Update the registers every cycle until we are done
Sequential Circuit for 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);

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

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 <- mkReg(0);
Reg#(Bit#(6)) i <- mkReg(32);

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

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 62 FAs 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
Folded IFFT: Reusing the Stage computation

BSV Code for \textit{stage}_f

\begin{verbatim}
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
\end{verbatim}

\textit{twid}'s are mathematically derivable constants
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 $f_0(x)$?

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
```
Shared Circuit

- The Twiddle constants can be expressed in a table or in a case or nested 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.
Superfolded IFFT: stage function \( f \)

```verilog
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
```

One Bfly-4 case

```verilog
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
```

Code for the Superfolded stage function

```verilog
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
```
**Syntax: Vector of Registers**

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

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

  *Don’t ask me why*

---

**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>

All these designs were done in less than 24 hours!

The same source code

The same source code

February 10, 2016

http://csg.csail.mit.edu/6.375

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