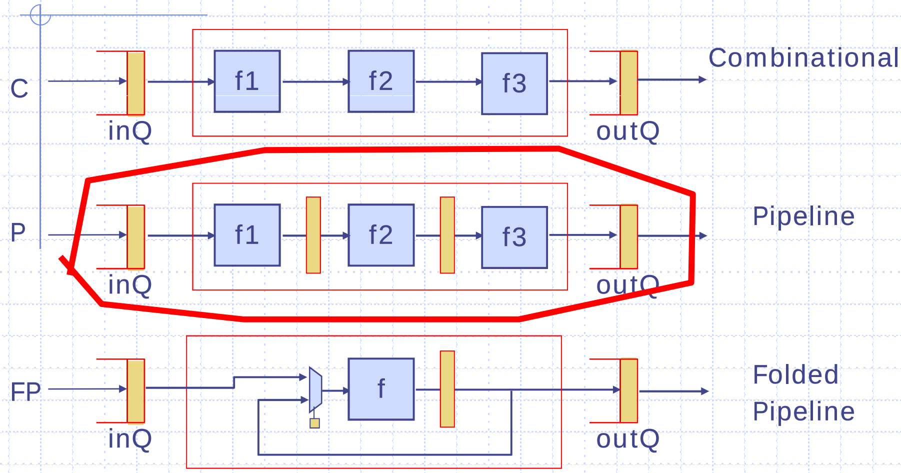
Simple Inelastic and Folded Pipelines

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Pipelining a block

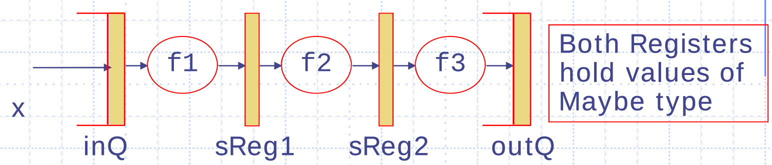


Clock: C < P ≈ FP

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) matches
  tagged Valid .sx1: sReg2 <= tagged Valid f2(sx1);
  tagged Invalid:    sReg2 <= tagged Invalid; endcase
 case (sReg2) matches
  tagged Valid .sx2: outQ.enq(f3(sx2)); endcase
endrule
```

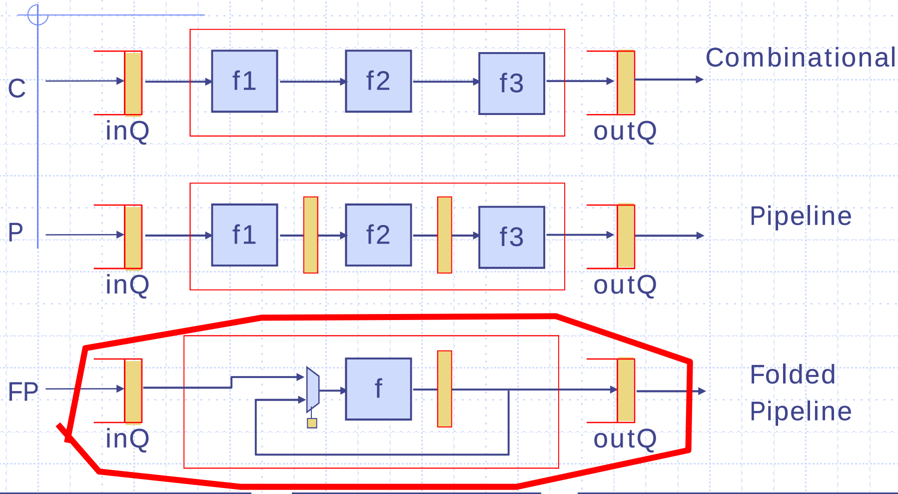

When is this rule enabled?

```
rule sync-pipeline (True);
 if (inQ.notEmpty())
  begin sReg1 <= tagged Valid f0(inQ.first()); inQ.deq(); end
  else  sReg1 <= tagged Invalid;
 case (sReg1) matches
  tagged Valid .sx1: sReg2 <= tagged Valid f1(sx1);
  tagged Invalid:    sReg2 <= tagged Invalid; endcase
 case (sReg2) matches
  tagged Valid .sx2: outQ.enq(f2(sx2));
 endcase
endrule
```

inQ	sReg1	sReg2	outQ	
NE	V	V	NF	yes
NE	V	V	F	No
NE	V	I	NF	
NE	V	I	F	
NE	I	V	NF	
NE	I	V	F	
NE	I	I	NF	
NE	I	I	F	

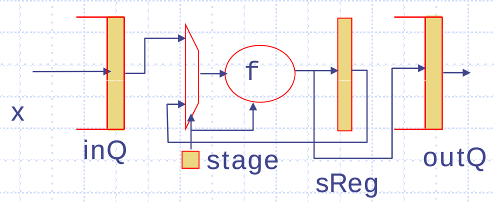
inQ	sReg1	sReg2	outQ	
E	V	V	NF	Yes
E	V	V	F	
E	V	I	NF	
E	V	I	F	
E	I	V	NF	
E	I	V	F	
E	I	I	NF	
E	I	I	F	

Pipelining a block



Clock: C < P ≈ 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
```

notice stage is a dynamic parameter now!

no for-loop

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

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

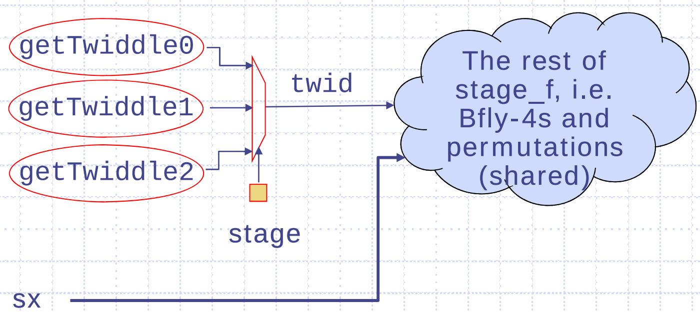
```
Function Vector#(64, Complex) f
        (Bit#(6) stagei, Vector#(64, Complex) stage_in);
   let i = stagei `mod` 16;
   let twid = getTwiddle(stagei `div` 16, i);
   let y = bfly4(twid, stage_in[i:i+3]);

   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
```

One Bfly-4 case

Folded pipeline: stage function f



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

802.11a Transmitter

[MEMOCODE 2006] Dave, Gerding, Pellauer, Arvind

Design Block	Lines of Code (BSV)	Relative Area
Controller	49	0%
Scrambler	40	0%
Conv. Encoder	113	0%
Interleaver	76	1%
Mapper	112	11%
IFFT	95	85%
Cyc. Extender	23	3%

Complex arithmetic libraries constitute another 200 lines of code

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

IFFT Design	Area (mm^2)	Throughput Latency (CLKs/sym)	Min. Freq Required
Pipelined	5.25	04	1.0 MHz
Combinational	**4.91**	04	1.0 MHz
Folded (16 Bfly-4s)	**3.97**	04	1.0 MHz
Super-Folded (8 Bfly-4s)	3.69	06	1.5 MHz
SF(4 Bfly-4s)	2.45	12	3.0 MHz
SF(2 Bfly-4s)	1.84	24	6.0 MHz
SF (1 Bfly4)	1.52	48	12 MHZ

The same source code

All these designs were done in less than 24 hours!

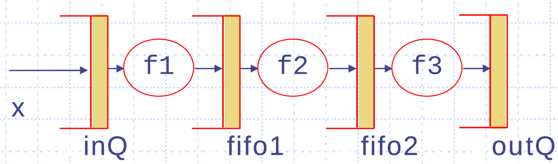
TSMC .18 micron; numbers reported are before place and route.

Why are the areas so similar

- Folding should have given a 3x improvement in IFFT area

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?

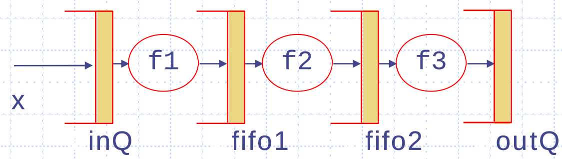
Can tokens be left inside the pipeline?

No Maybe types?

Easier to write?

Can all three rules fire concurrently?

What behavior do we want?

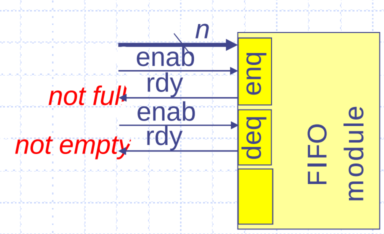


inQ	fifo1	fifo2	outQ
NE	NE,NF	NE,NF	NF
NE	NE,NF	NE,NF	F
NE	NE,NF	NE,F	NF
NE	NE,NF	NE,F	F
….			

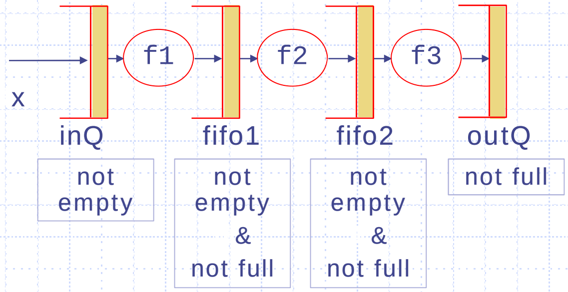
rule1	rule2	rule3
Yes	Yes	Yes
Yes	Yes	No

One-Element FIFO

```
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
```



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

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

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

x will get bound to the appropriate part of m

```
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

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

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

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

Implicit guards (conditions)

- Rule

  rule <name> (<guard>); <action>; **endrule**

  where

  <action> ::= r <= <exp>

  | ~~m.g(<exp>)~~ → $m.g_B$(<exp>) **when** $m.g_G$

  | **if** (<exp>) <action> **endif**

  | <action> ; <action>

make implicit guards explicit

Guards vs If's

- A guard on one action of a parallel group of actions affects every action within the group

  (a1 when p1); (a2 when p2)

  ==> (a1; a2) when (p1 && p2)

- A condition of a Conditional action only affects the actions within the scope of the conditional action

  (if (p1) a1); a2

  p1 has no effect on a2 ...

- Mixing ifs and whens

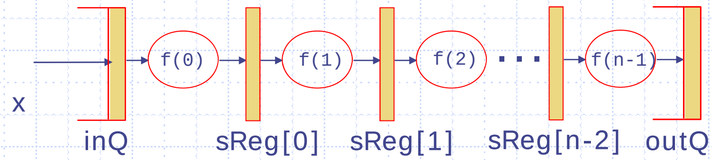
  (if (p) (a1 when q)) ; a2

  ≡ ((if (p) a1); a2) when ((p && q) | !p)

Static vs dynamic expressions

- Expressions that can be evaluated at compile time will be evaluated at compile-time
  - 3+4 → 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




```
rule sync-pipeline (True);
 if (inQ.notEmpty())
  begin sReg[0]<= tagged Valid
f(1,inQ.first());inQ.deq();end
  else  sReg[0]<= tagged Invalid;
 for(Integer i = 1; i < n-1; i=i+1) begin
  case (sReg[i-1]) matches
   tagged Valid .sx: sReg[i] <= tagged Valid f(i-1,sx);
   tagged Invalid:   sReg[i] <= tagged Invalid; endcase end
 case (sReg[n-2]) matches
   tagged Valid .sx: outQ.enq(f(n-1,sx)); endcase
endrule
```


Next lecture

Concurrency analysis