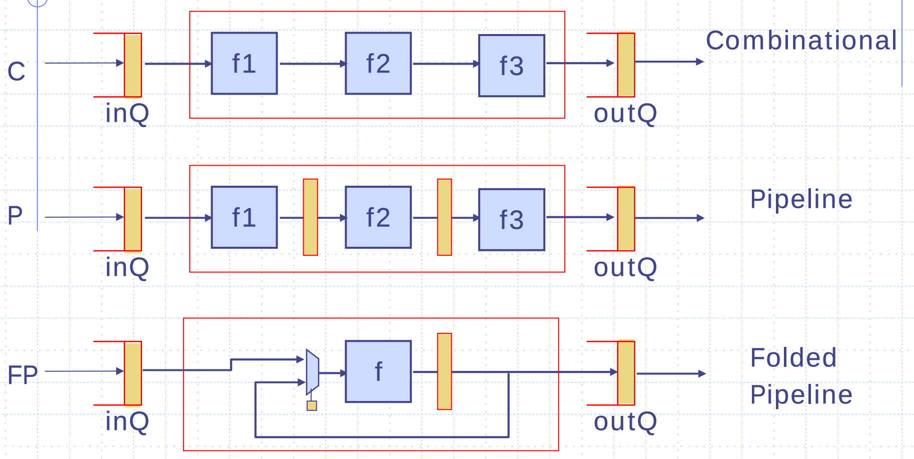
Simple Synchronous Pipelines

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Pipelining a block

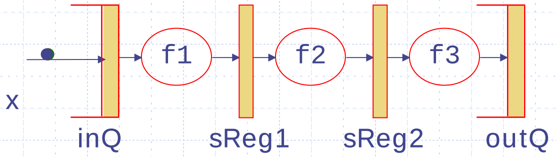


Clock: C < P ≈ FP

Area: FP < C < P

Throughput: FP < C < P

Synchronous pipeline



```
rule sync-pipeline (True);
  inQ.deq();
  sReg1 <= f1(inQ.first());
  sReg2 <= f2(sReg1);
  outQ.enq(f3(sReg2));
endrule
```

This rule can fire only if

Stage functions f1, f2 and f3

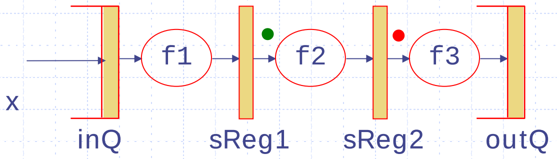
```
function f1(x);
         return (stage_f(1,x));
endfunction

function f2(x);
         return (stage_f(2,x));
endfunction

function f3(x);
         return (stage_f(3,x));
endfunction
```

The stage_f function was given earlier

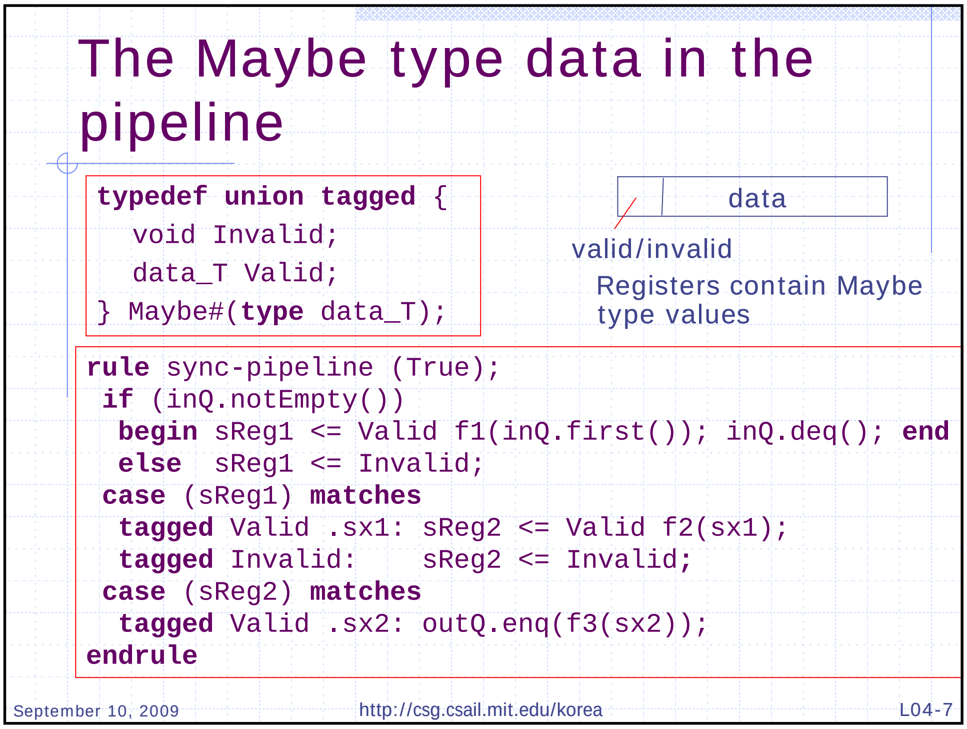
Bluespec Code for stage_f

```
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
      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);
```

Problem: What about pipeline bubbles?



```
rule sync-pipeline (True);
  inQ.deq();
  sReg1 <= f1(inQ.first());
  sReg2 <= f2(sReg1);
  outQ.enq(f3(sReg2));
endrule
```



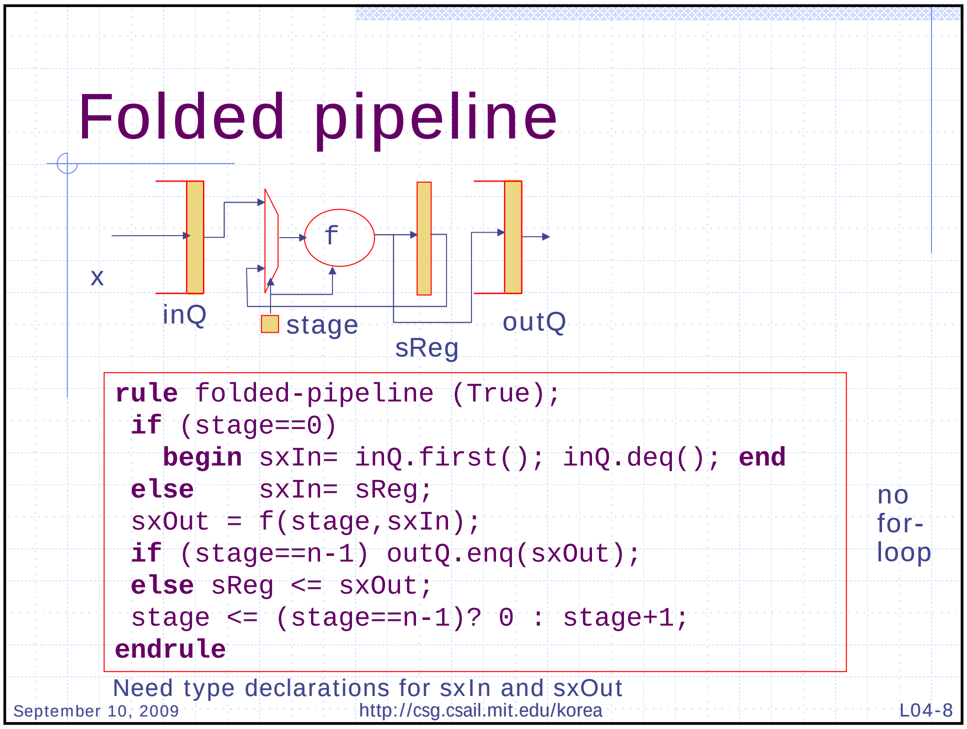
The Maybe type data in the pipeline

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

data
valid/invalid

Registers contain Maybe type values

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



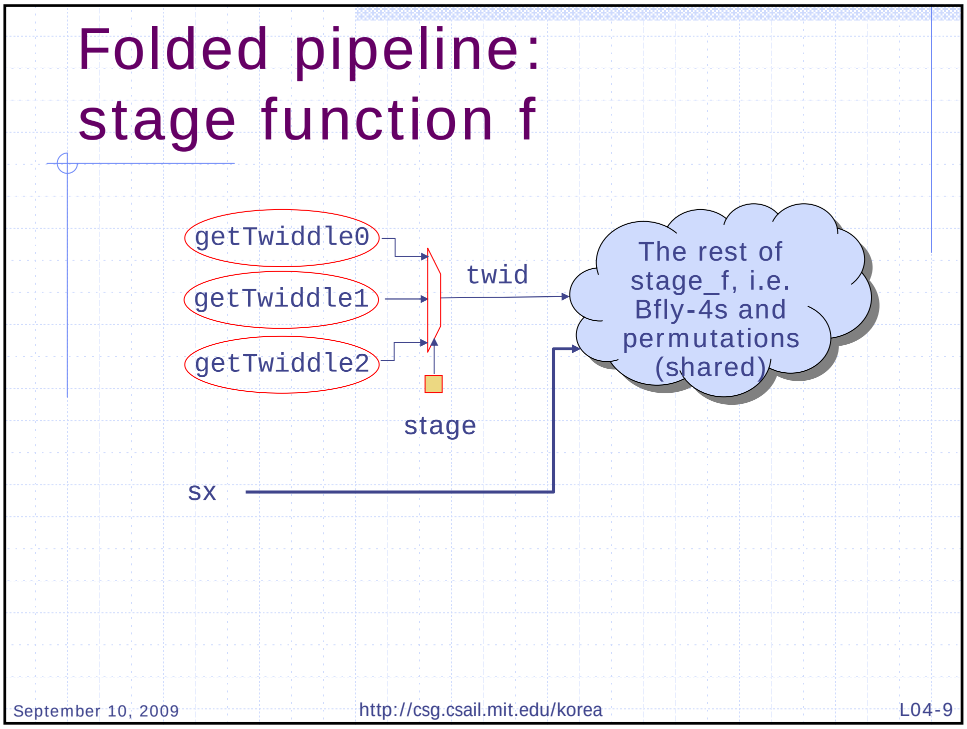
Folded pipeline

f
x
inQ
stage
sReg
outQ

```
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
```

no for-loop

Need type declarations for sxIn and sxOut



Folded pipeline: stage function f

getTwiddle0

getTwiddle1

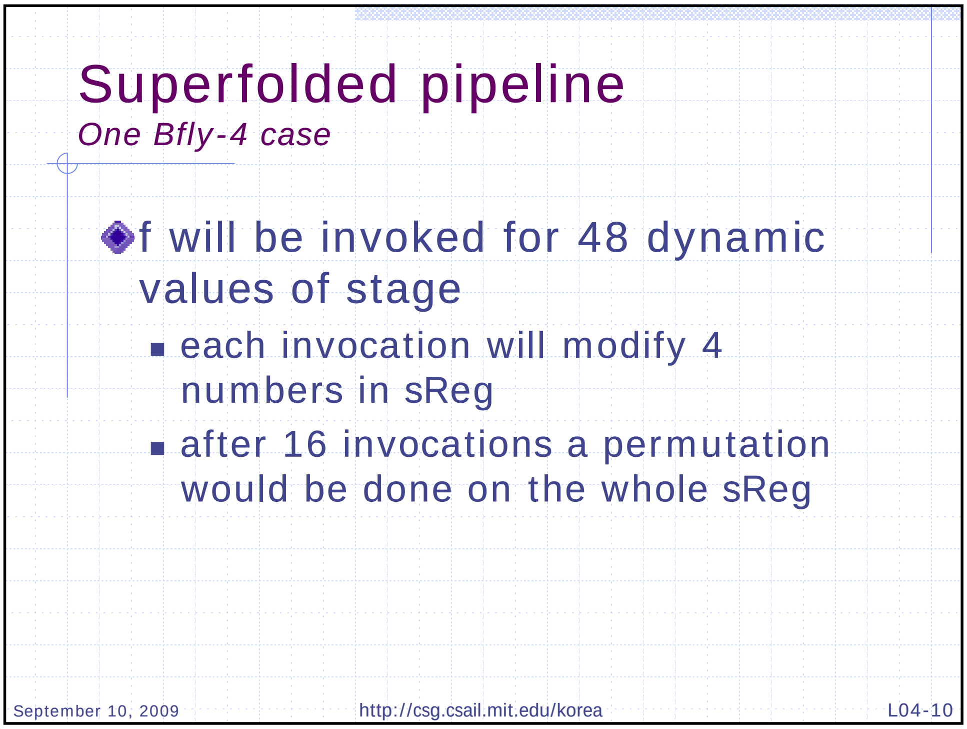
getTwiddle2

twid

stage

sx

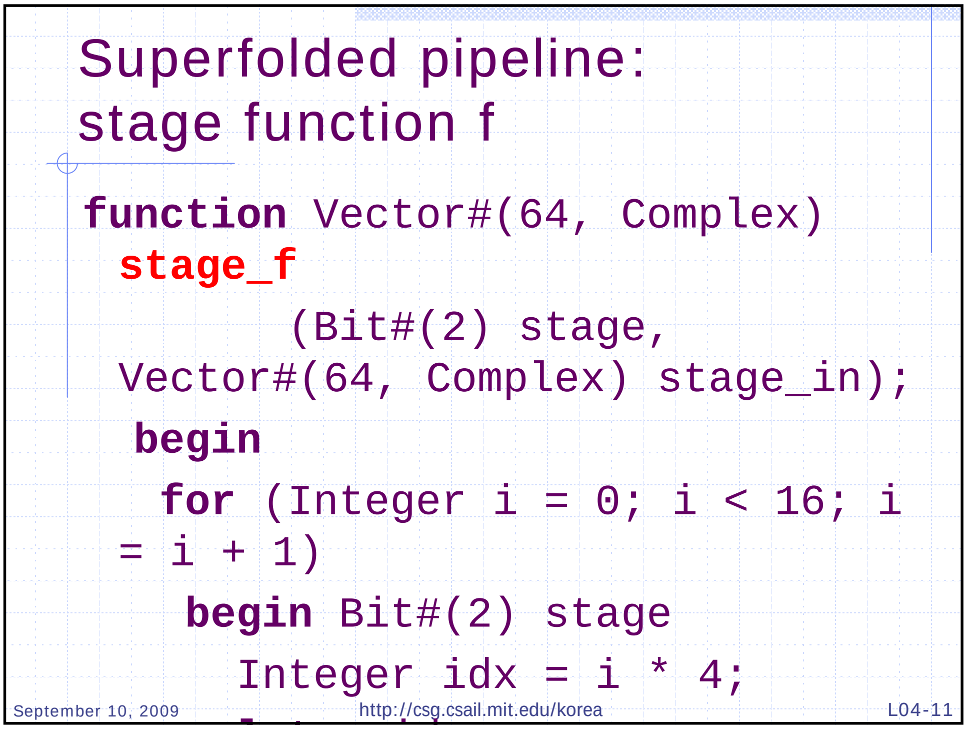
The rest of stage_f, i.e. Bfly-4s and permutations (shared)



Superfolded pipeline

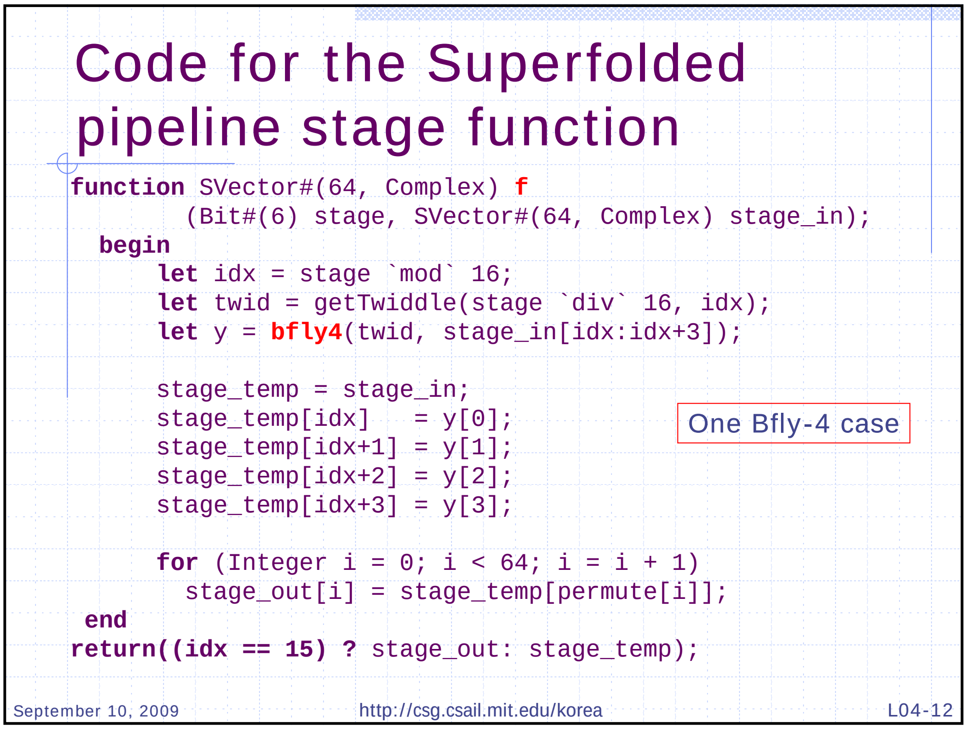
One Bfly-4 case

- 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 pipeline: stage function f

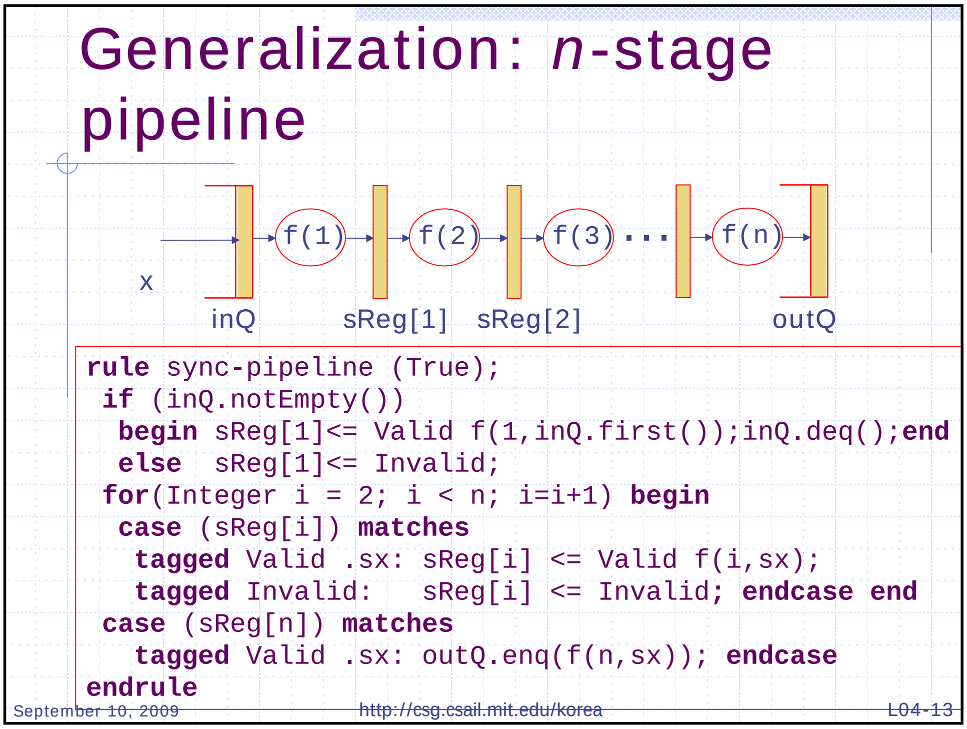
```
function Vector#(64, Complex)
  stage_f
             (Bit#(2) stage,
  Vector#(64, Complex) stage_in);
   begin
     for (Integer i = 0; i < 16; i
  = i + 1)
      begin Bit#(2) stage
          Integer idx = i * 4;
```



Code for the Superfolded pipeline stage function

```
function SVector#(64, Complex) f
        (Bit#(6) stage, SVector#(64, Complex) stage_in);
  begin
      let idx = stage `mod` 16;
      let twid = getTwiddle(stage `div` 16, idx);
      let y = bfly4(twid, stage_in[idx:idx+3]);

      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];

      for (Integer i = 0; i < 64; i = i + 1)
        stage_out[i] = stage_temp[permute[i]];
 end
return((idx == 15) ? stage_out: stage_temp);
```

One Bfly-4 case

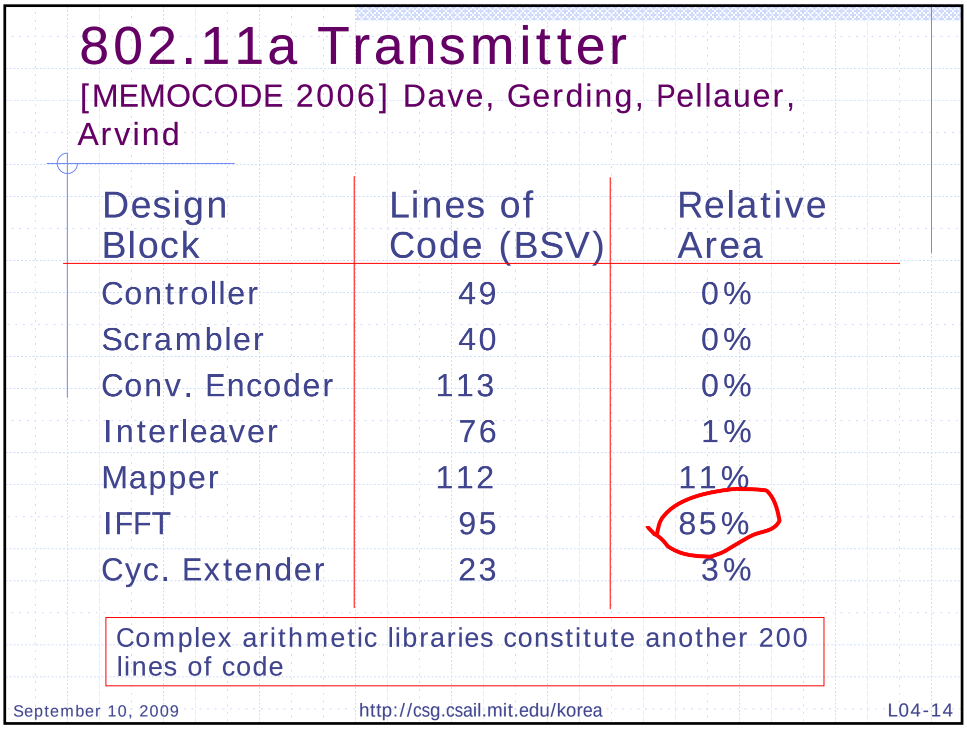


Generalization: *n*-stage pipeline

x
f(1)
f(2)
f(3)
...
f(n)
inQ
sReg[1]
sReg[2]
outQ

```
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]) matches
   tagged Valid .sx: sReg[i] <= Valid f(i,sx);
   tagged Invalid:   sReg[i] <= Invalid; endcase end
 case (sReg[n]) matches
   tagged Valid .sx: outQ.enq(f(n,sx)); endcase
endrule
```

September 10, 2009 http://csg.csail.mit.edu/korea L04-13



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

September 10, 2009 http://csg.csail.mit.edu/korea L04-14

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

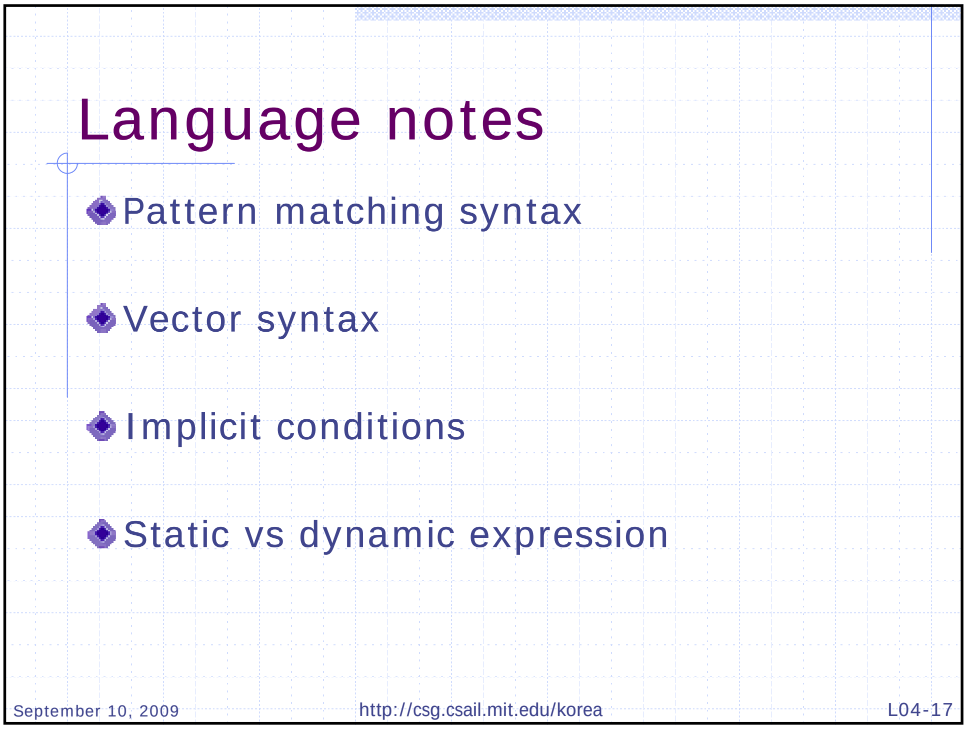
The same source code

All these designs were done in less than 24 hours!

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

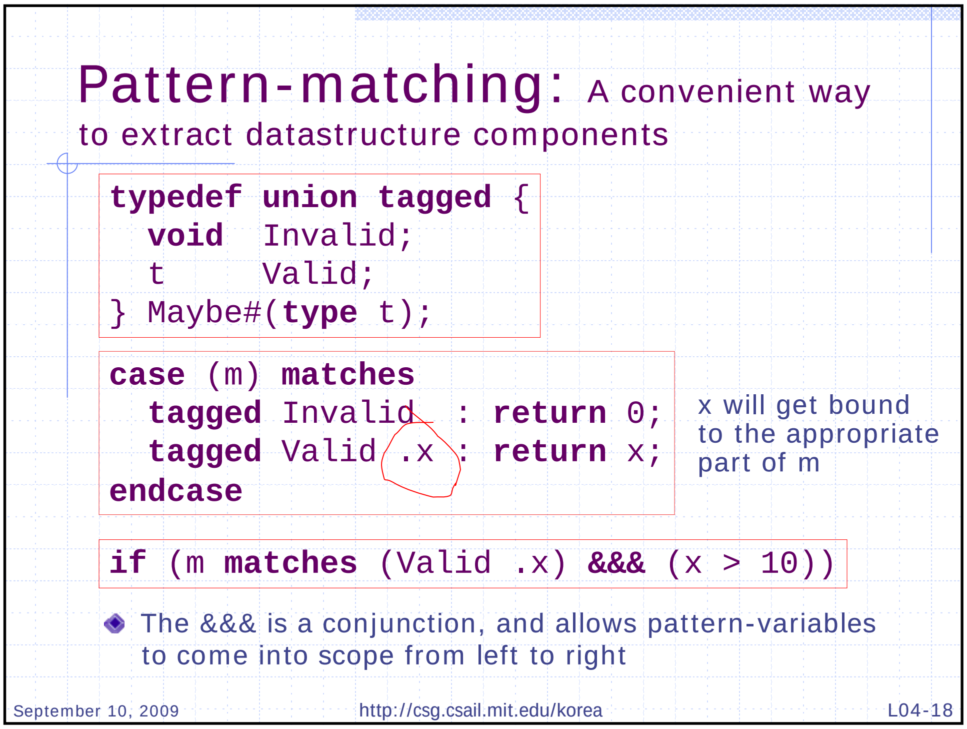
Why are the areas so similar

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



Language notes

- Pattern matching syntax
- Vector syntax
- Implicit conditions
- Static vs dynamic expression



Pattern-matching: A convenient way to extract datastructure components

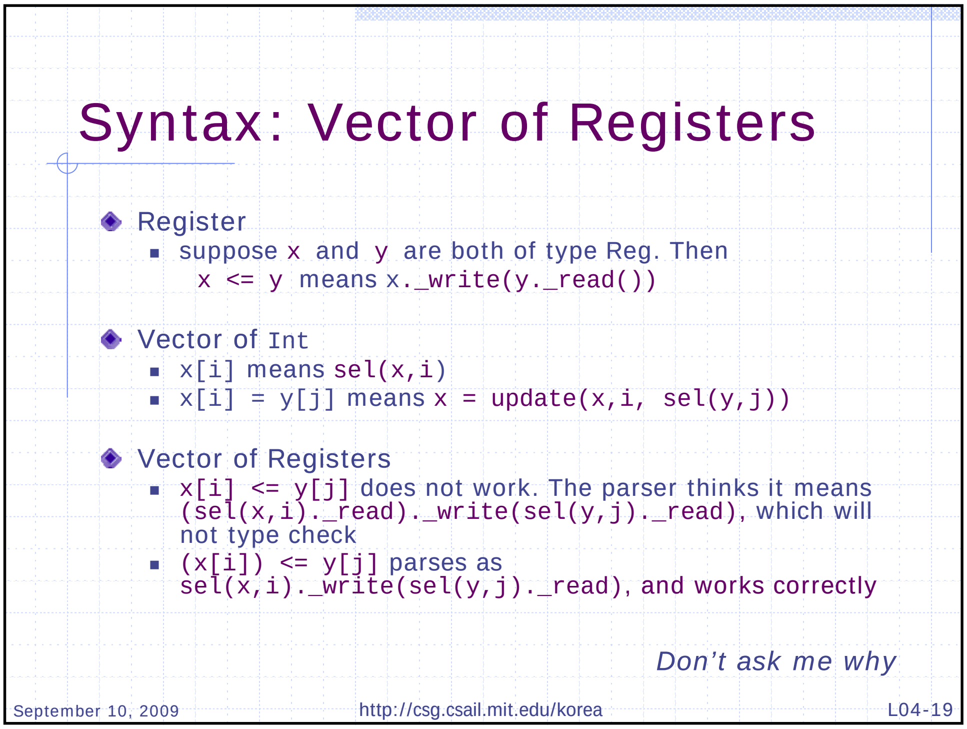
```
typedef union tagged {
   void  Invalid;
   t     Valid;
} Maybe#(type t);
```

```
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))
```

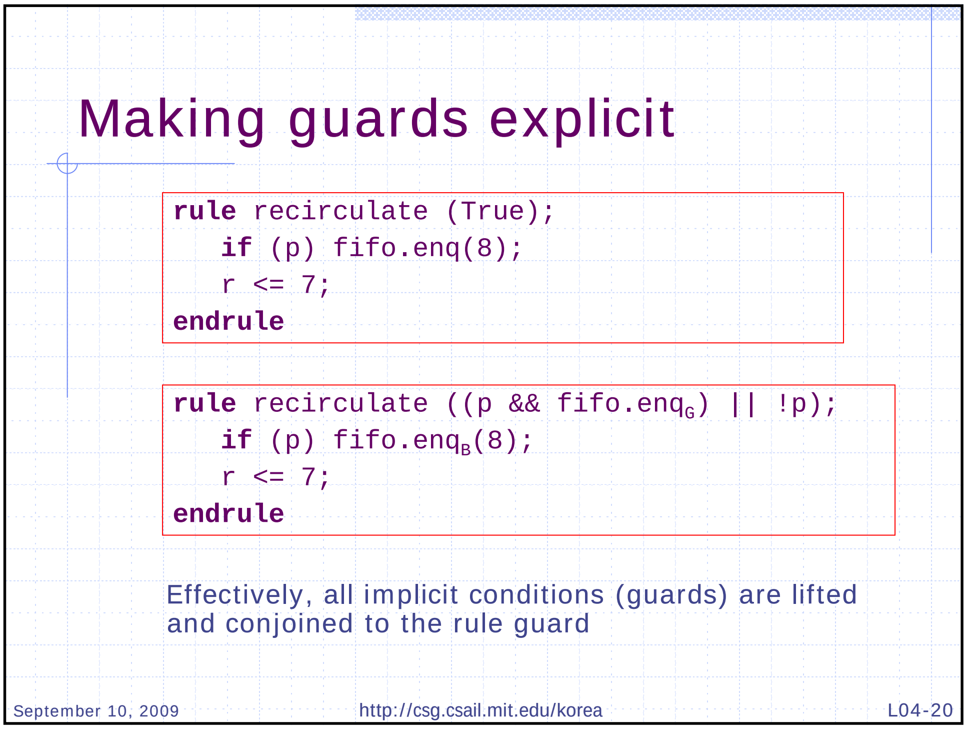
- The &&& is a conjunction, and allows pattern-variables to come into scope from left to right



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

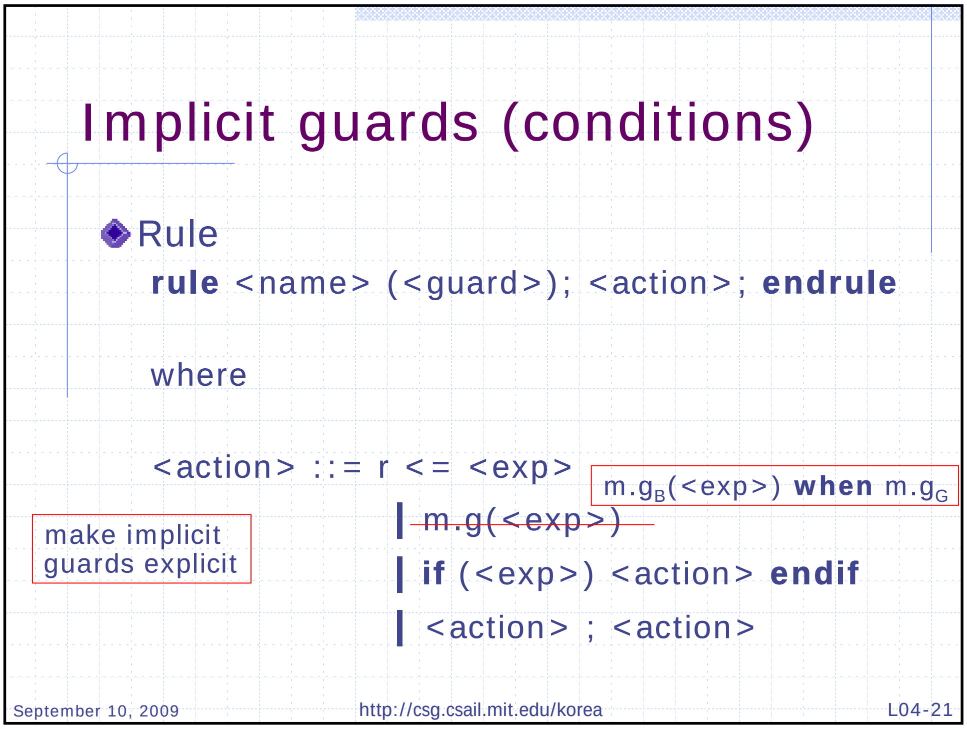


Making guards explicit

```
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

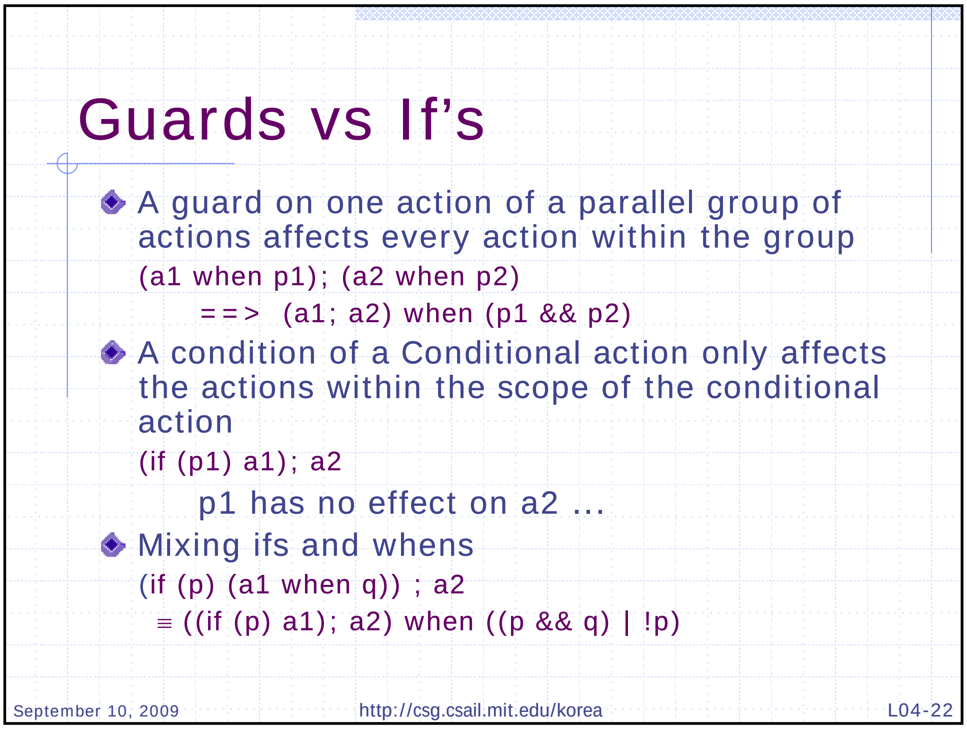
  <action> ::= r <= <exp>

  | ~~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

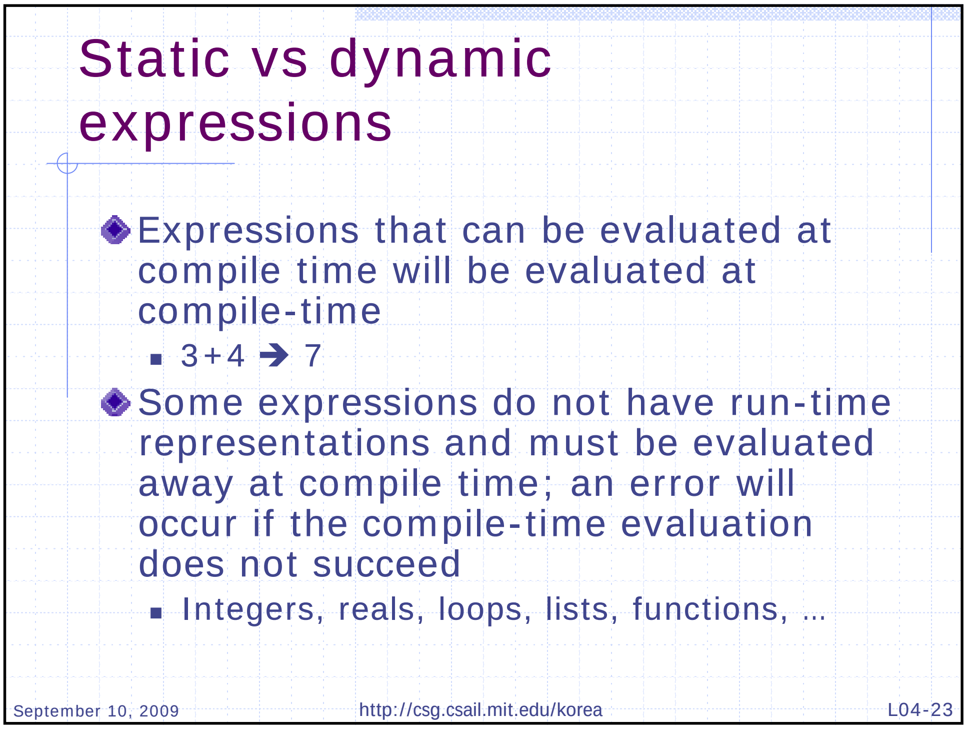
  (if (p1) a1); a2

  p1 has no effect on a2 ...

- Mixing ifs and whens

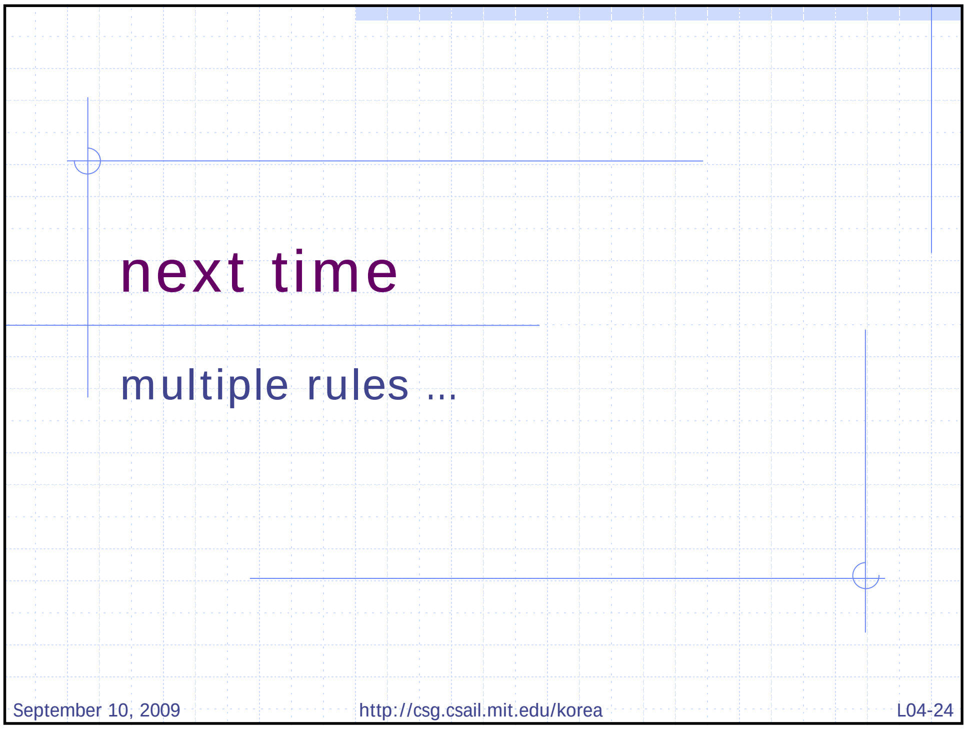
  (if (p) (a1 when q)) ; a2

  ≡ ((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, …



next time

multiple rules …