# Pipelining combinational circuits

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
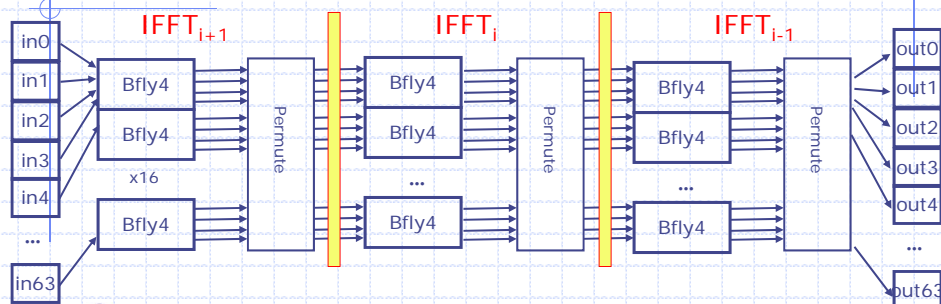
---

# Combinational IFFT
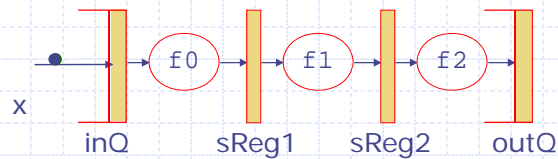
3 different datasets in the pipeline



- ◈ Lot of area and long combinational delay
- ◈ Folded or multi-cycle version can save area and reduce the combinational delay but throughput per clock cycle gets worse
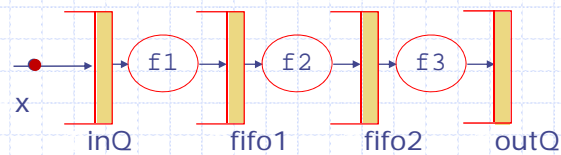- ◈ Pipelining: a method to increase the circuit throughput by evaluating multiple IFFTs

# Inelastic vs Elastic pipeline



Inelastic: all pipeline stages move synchronously



Elastic: A pipeline stage can process data if its
input FIFO is not empty and output FIFO is not Full

**Most complex processor pipelines are a combination of the two styles**

---

# Inelastic vs Elastic Pipelines

◆ 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 mistakes; difficult to make changes
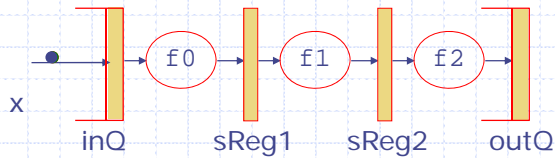
◆ Elastic pipeline:
  - several smaller rules, each easy to write, easier to make changes
  - *downside:* sometimes rules do not fire concurrently when they should

# Inelastic pipeline



x → inQ → f0 → sReg1 → f1 → sReg2 → f2 → outQ

```
rule sync-pipeline (True);
    inQ.deq();
    sReg1 <= f0(inQ.first());
    sReg2 <= f1(sReg1);
    outQ.enq(f2(sReg2));
endrule
```

This rule can fire only if

Atomicity: Either *all* or *none* of the state elements inQ, outQ, sReg1 and sReg2 will be updated

---

# Inelastic pipeline
## Making implicit guard conditions explicit



x → inQ → f0 → sReg1 → f1 → sReg2 → f2 → outQ

```
rule sync-pipeline (!inQ.empty() && !outQ.full);
    inQ.deq();
    sReg1 <= f0(inQ.first());
    sReg2 <= f1(sReg1);
    outQ.enq(f2(sReg2));
endrule
```

Suppose sReg1 and sReg2 have data, outQ is not full but inQ is empty. What behavior do you expect?

# Pipeline bubbles



```
rule sync-pipeline (True);
    inQ.deq();
    sReg1 <= f0(inQ.first());
    sReg2 <= f1(sReg1);
    outQ.enq(f2(sReg2));
endrule
```
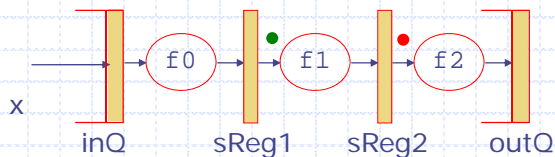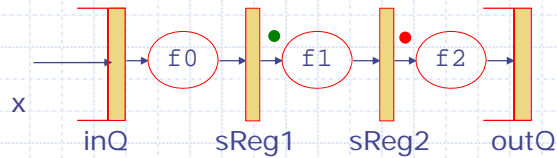
Red and Green tokens must move even if there is nothing in inQ!

Also if there is no token in sReg2 then nothing should be enqueued in the outQ

Modify the rule to deal with these conditions

Valid bits or the Maybe type

---

# Explicit encoding of Valid/Invalid data
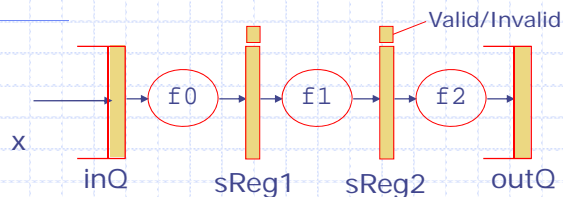


```
typedef union tagged {void Valid; void Invalid;
} Validbit deriving (Eq, Bits);
```

```
rule sync-pipeline (True);
  if (inQ.notEmpty())
   begin sReg1 <= f0(inQ.first()); inQ.deq();
         sReg1f <= Valid end
   else  sReg1f <= Invalid;
  sReg2 <= f1(sReg1); sReg2f <= sReg1f;
  if (sReg2f == Valid) outQ.enq(f2(sReg2));
endrule
```
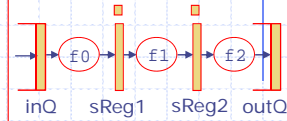
4

# When is this rule enabled?

```
rule sync-pipeline (True);
  if (inQ.notEmpty())
    begin sReg1 <= f0(inQ.first()); inQ.deq();
          sReg1f <= Valid end
    else  sReg1f <= Invalid;
    sReg2 <= f1(sReg1); sReg2f <= sReg1f;
  if (sReg2f == Valid) outQ.enq(f2(sReg2));
endrule
```

inQ → |→ f0 →|→ f1 →|→ f2 →|
       inQ   sReg1  sReg2 outQ

| inQ | sReg1f | sReg2f | outQ |     | inQ | sReg1f | sReg2f | outQ |     |
|-----|--------|--------|------|-----|-----|--------|--------|------|-----|
| NE  | V      | V      | NF   | yes | E   | V      | V      | NF   | yes |
| NE  | V      | V      | F    |     | E   | V      | V      | F    |     |
| NE  | V      | I      | NF   |     | E   | V      | I      | NF   |     |
| NE  | V      | I      | F    |     | E   | V      | I      | F    |     |
| NE  | I      | V      | NF   |     | E   | I      | V      | NF   |     |
| NE  | I      | V      | F    |     | E   | I      | V      | F    |     |
| NE  | I      | I      | NF   |     | E   | I      | I      | NF   |     |
| NE  | I      | I      | F    |     | E   | I      | I      | F    |     |

Yes1 = yes
but no change

# The Maybe type
### A useful type to capture valid/invalid data

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

|  | data |

valid/invalid

Registers contain Maybe
type values

Some useful functions on Maybe type:

isValid(x) returns true if x is Valid

fromMaybe(d,x) returns

the data value in x if x is Valid

the default value d if x is Invalid

# Using the Maybe type

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

|   | data |
|---|------|

valid/invalid

Registers contain Maybe type values

```
rule sync-pipeline if (True);
  if (inQ.notEmpty())
    begin sReg1 <= Valid f0(inQ.first()); inQ.deq(); end
    else  sReg1 <= Invalid;
    sReg2 <= isValid(sReg1)? Valid f1(fromMaybe(d, sReg1)) :
                             Invalid;
  if isValid(sReg2) outQ.enq(f2(fromMaybe(d, sReg2)));
endrule
```

---

# Pattern-matching: An alternative syntax to extract datastructure components

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

6

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

sx1 will get bound to the appropriate part of sReg1

---

## Generalization: *n*-stage pipeline



X

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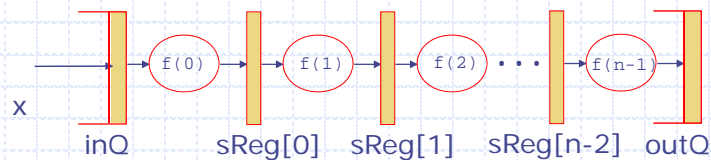
```
rule sync-pipeline (True);
  if (inQ.notEmpty())
    begin sReg[0]<= Valid f(1,inQ.first());inQ.deq();end
    else  sReg[0]<= Invalid;
  for(Integer i = 1; i < n-1; i=i+1) begin
    case (sReg[i-1]) matches
      tagged Valid .sx: sReg[i] <= Valid f(i-1,sx);
      tagged Invalid:   sReg[i] <= Invalid;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

7

# Elastic pipeline

Use FIFOs instead of pipeline registers
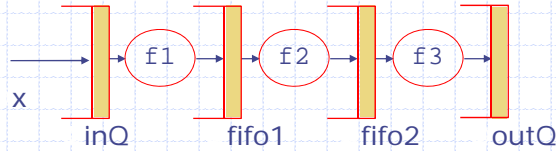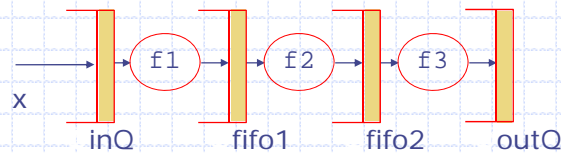


```
rule stage1 if (True);
    fifo1.enq(f1(inQ.first()));
    inQ.deq();        endrule
rule stage2 if (True);
    fifo2.enq(f2(fifo1.first()));
    fifo1.deq();    endrule
rule stage3 if (True);
    outQ.enq(f3(fifo2.first()));
    fifo2.deq();    endrule
```

◆ What is the firing condition for each rule?

◆ Can tokens be left inside the pipeline?

◆ No need for Maybe types

---

# Firing conditions for reach rule



| inQ | fifo1 | fifo2 | outQ | rule1 | rule2 | rule3 |
|-----|-------|-------|------|-------|-------|-------|
| NE  | NE,NF | NE,NF | NF   | Yes   | Yes   | Yes   |
| NE  | NE,NF | NE,NF | F    | Yes   | Yes   | No    |
| NE  | NE,NF | NE,F  | NF   |       |       |       |
| NE  | NE,NF | NE,F  | F    |       |       |       |
| …   |       |       |      | …     |       |       |

◆ This is the first example we have seen where multiple rules may be ready to execute concurrently

◆ Can we execute multiple rules together?

8

## Informal analysis
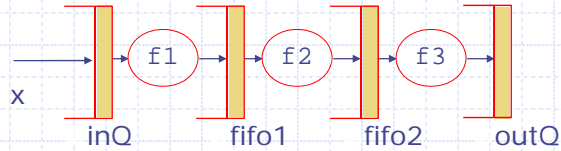
f1   f2   f3

x   inQ   fifo1   fifo2   outQ

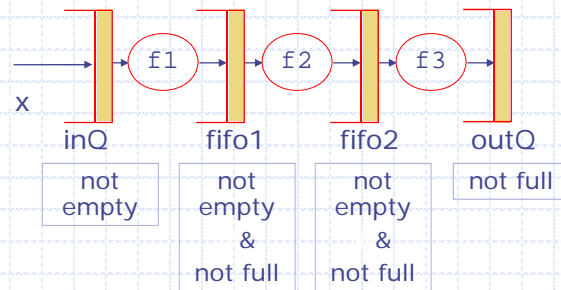| inQ | fifo1 | fifo2 | outQ | rule1 | rule2 | rule3 |
|-----|-------|-------|------|-------|-------|-------|
| NE | NE,NF | NE,NF | NF | Yes | Yes | Yes |
| NE | NE,NF | NE,NF | F | Yes | Yes | No |
| NE | NE,NF | NE,F | NF | Yes | No | Yes |
| NE | NE,NF | NE,F | F | Yes | No | No |
| …. | | | | …. | | |

FIFOs must permit concurrent enq and deq for all three
rules to fire concurrently

---

## Concurrency when the FIFOs do not permit concurrent enq and deq

f1   f2   f3

x   inQ   fifo1   fifo2   outQ

| not empty | not empty & not full | not empty & not full | not full |
|-----------|----------------------|----------------------|----------|

At best alternate stages in the pipeline will
be able to fire concurrently

9

# Pipelined designs expressed using Multiple rules

- ◆ If rules for different pipeline stages never fire in the same cycle then the design can hardly be called a pipelined design

- ◆ If all the enabled rules fire in parallel every cycle then, in general, wrong results can be produced

We need a clean model for concurrent firing of rules

# BSV Rule Execution

- ◆ A BSV program consists of state elements and rules, aka, Guarded Atomic Actions (GAA) that operate on the state elements
- ◆ Application of a rule modifies some state elements of the system in a deterministic manner

10
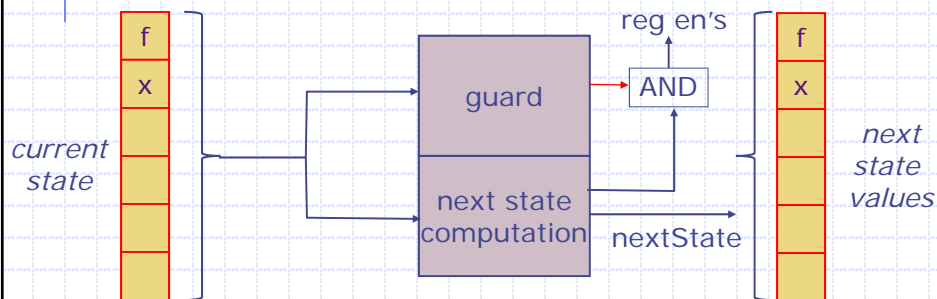
# BSV Execution Model

*Repeatedly:*

- ◆ Select a rule to execute    ← Highly non-deterministic
- ◆ Compute the state updates
- ◆ Make the state updates

User annotations can help in rule selection

---

# One-rule-at-time-semantics

- ◆ The legal behavior of a BSV program can always be explained by observing the state updates obtained by applying only one rule at a time

Implementation concern: Schedule multiple rules concurrently without violating one-rule-at-a-time semantics

# Guard lifting

◆ For concurrent scheduling of rules, we only need to consider those rules that can be concurrently enabled, i.e., whose guards are true

◆ In order to understand when a rule can be enabled, we need to understand precisely how implicit guards are lifted precisely to form the rule guard

# Making guards explicit

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

```
rule foo if ((p && fifo.notFull) || !p);
    if (p) fifo.enq(8);
    r <= 7;
endrule
```

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

## Implicit guards (conditions)

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

```
<action> ::= r <= <exp>
           | if (<exp>) <action>
           | <action> ; <action>
           | m.g(<exp>)        m.g_B(<exp>) when m.g_G
           | t = <exp>
```

make implicit
guards explicit

```
<action> ::= r <= <exp>
           | if (<exp>) <action>
           | <action> when (<exp>)
           | <action> ; <action>
           | m.g_B(<exp>)
           | t = <exp>
```

February 20, 2013                http://csg.csail.mit.edu/6.375                L05-25

---

## 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
    ==>  (a1; a2) when p1

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

February 20, 2013                http://csg.csail.mit.edu/6.375                L05-26

13

# Guard Lifting rules

◆ All the guards can be "lifted" to the top of a rule

- (a1 when p) ; a2 $\Rightarrow$
- a1 ; (a2 when p) $\Rightarrow$

- if (p when q) a $\Rightarrow$
- if (p) (a when q) $\Rightarrow$

- (a when p1) when p2 $\Rightarrow$
- x <= (e when p) $\Rightarrow$

similarly for expressions …

- Rule r (a when p) $\Rightarrow$

---

BSV provides a primitive (impCondOf) to make guards explicit and lift them to the top

From now on in concurrency discussions we will assume that all guards have been lifted to the top in every rule