Constructive Computer Architecture:

Pipelining combinational circuits

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

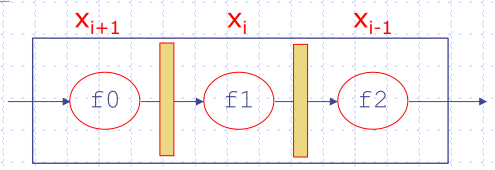
Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

Contents

- Inelastic versus Elastic pipelines
- The role of FIFOs
- Concurrency issues
- BSV Concepts
  - The Maybe Type
  - Concurrency analysis

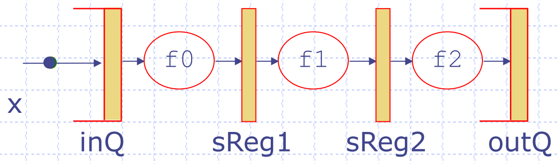
Complex Combinational Functions



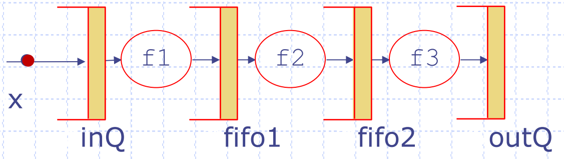
3 different datasets in the pipeline

- 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 inputs

Inelastic vs Elastic pipeline



Inelastic: all pipeline stages move synchronously



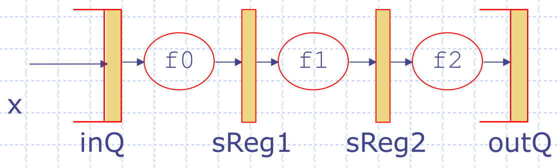
Elastic: A pipeline stage can process data if its input FIFO is not empty and output FIFO is not Full

Most complex processor pipelines are a combination of the two styles

Inelastic vs Elastic Pipelines

- Inelastic pipeline:
  - typically only one rule or mutually exclusive rules; the designer controls precisely which activities go on in parallel
  - *downside:* The designer must program the starting and draining of the pipeline. The rule can get 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

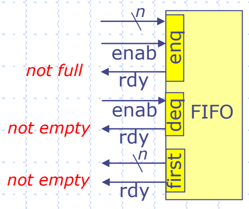


```
rule sync-pipeline (True);
  inQ.deq();
  sReg1 <= f0(inQ.first());
  sReg2 <= f1(sReg1);
  outQ.enq(f2(sReg2));
endrule
```

This rule can fire only if
- inQ has an element
- outQ has space

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

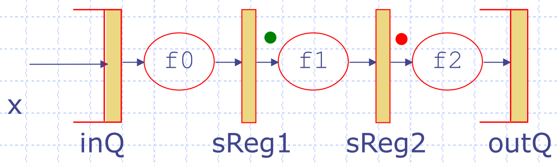
FIFO Module: methods with guarded interfaces



fifo.enq(x); // action method
fifo.deq(); // action method
y=fifo.first() // value method

Inelastic pipeline

Making implicit guard conditions explicit

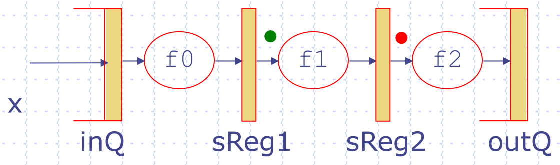


```
rule sync-pipeline (!inQ.empty() && !outQ.full);
  inQ.deq();
  sReg1 <= f0(inQ.first());
  sReg2 <= f1(sReg1);
  outQ.enq(f2(sReg2));
endrule
```

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

Leave green and red data in the pipeline?

Pipeline bubbles



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

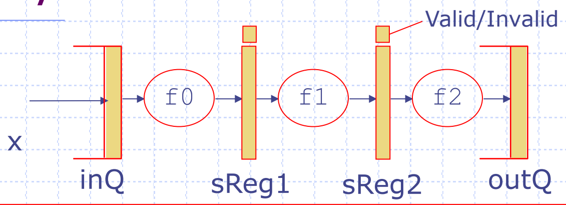
Red and Green tokens must move even if there is nothing in inQ!

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

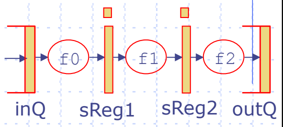


```
typedef union tagged {void Valid; void Invalid;
} Validbit deriving (Eq, Bits);
```

```
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
```

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	sReg1f	sReg2f	outQ	
NE	V	V	NF	yes
NE	V	V	F	No
NE	V	I	NF	Yes
NE	V	I	F	Yes
NE	I	V	NF	Yes
NE	I	V	F	No
NE	I	I	NF	Yes
NE	I	I	F	yes

inQ	sReg1f	sReg2f	outQ	
E	V	V	NF	yes
E	V	V	F	No
E	V	I	NF	Yes
E	V	I	F	Yes
E	I	V	NF	Yes
E	I	V	F	No
E	I	I	NF	Yes1
E	I	I	F	yes

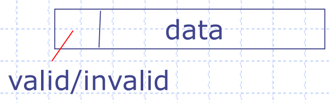
NE = Not Empty; NF = Not Full

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);
```



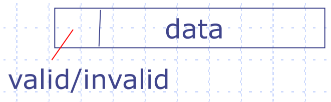
Registers contain Maybe type values

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);
```



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
```

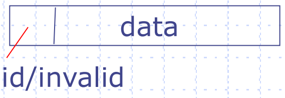
x will get bound to the appropriate part of m

```
if (m matches (Valid .x) &&& (x > 10))
```

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

The Maybe type data using the pattern matching syntax

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

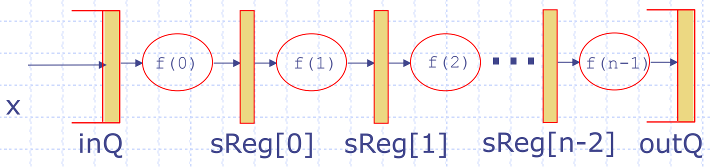


Registers contain Maybe type values

```
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

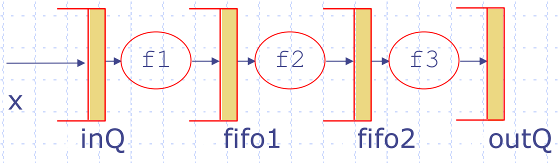
Generalization: *n*-stage pipeline



```
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,sx);
   tagged Invalid:   sReg[i] <= Invalid; endcase end
 case (sReg[n-2]) matches
   tagged Valid .sx: outQ.enq(f(n-1,sx)); endcase
endrule
```

Elastic pipeline

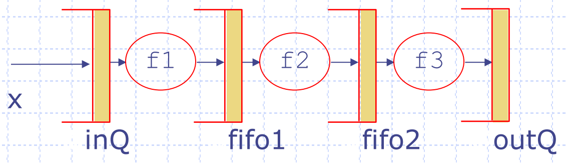
Use FIFOs instead of pipeline registers



```
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

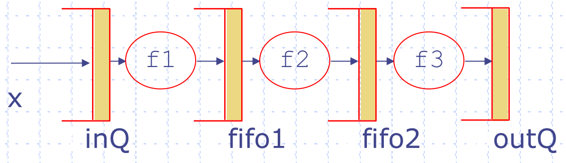


inQ	fifo1	fifo2	outQ
NE	NE,NF	NE,NF	NF
NE	NE,NF	NE,NF	F
NE	NE,NF	NE,F	NF
NE	NE,NF	NE,F	F
….			

rule1	rule2	rule3
Yes	Yes	Yes
Yes	Yes	No
Yes	No	Yes
Yes	No	No
….		

- This is the first example we have seen where multiple rules may be ready to execute concurrently
- Can we execute multiple rules together?

Informal analysis

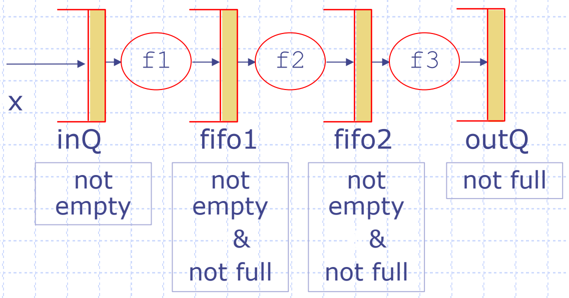


inQ	fifo1	fifo2	outQ
NE	NE,NF	NE,NF	NF
NE	NE,NF	NE,NF	F
NE	NE,NF	NE,F	NF
NE	NE,NF	NE,F	F
….			

rule1	rule2	rule3
Yes	Yes	Yes
Yes	Yes	No
Yes	No	Yes
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



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

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

BSV Execution Model

Repeatedly:

- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic; User annotations can be used in rule selection

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

One-rule-at-time semantics

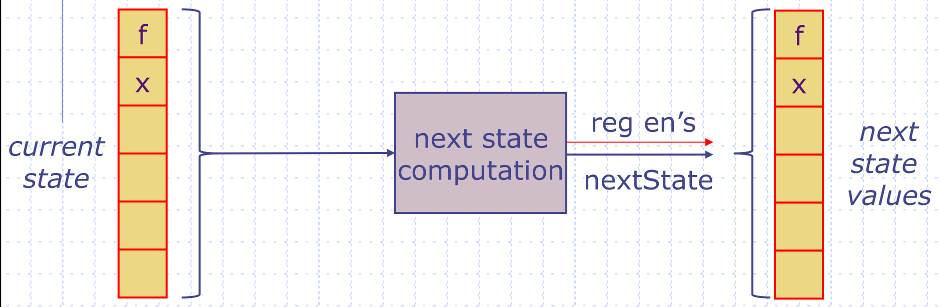
Concurrent scheduling of rules

- The one-rule-at-a-time semantics plays the central role in defining functional correctness and verification but for meaningful hardware design it is necessary to execute multiple rules concurrently without violating the one-rule-at-a-time semantics
- What do we mean by concurrent scheduling?
  - First - some hardware intuition
  - Later – the semantics of concurrent scheduling

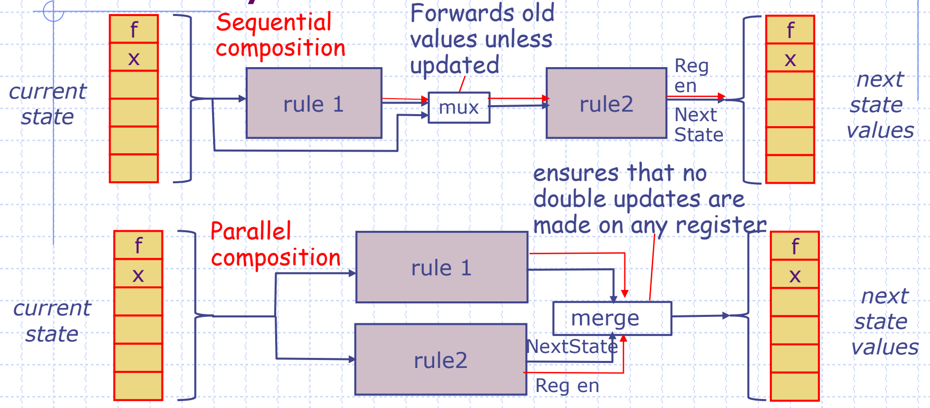
Hardware intuition for concurrent scheduling

Rule Execution

- Application of a rule modifies some state elements of the system in a deterministic manner



Executing multiple rules in one clock cycle



- Sequential composition preserves the one-rule-at-a-time semantics but generally increases the critical combinational path
- Parallel composition does not create longer combinational paths but may not preserve the one-rule-at-a-time semantics

Violation of sequential semantics

```
rule ra
   x <= y;
endrule

rule rb
   y <= x;
endrule
```

- Suppose initially x is x0 and y is y0
- {x0,y0} $\Rightarrow_{ra}$ {y0,y0} $\Rightarrow_{rb}$ {y0,y0}
- {x0,y0} $\Rightarrow_{rb}$ {x0,x0} $\Rightarrow_{ra}$ {x0,x0}
- {x0,y0} $\Rightarrow_{rb||ra}$ {y0,x0}

Parallel execution does not behave like either ra<rb or rb<ra

We do not want to allow concurrent execution of ra and rb

next time compiler analysis to determine which rules can be executed concurrently