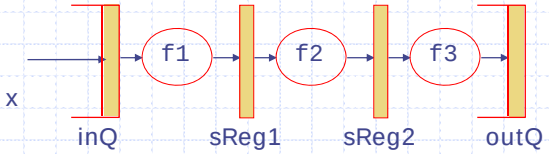
Asynchronous Pipelines: *Concurrency Issues*

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Synchronous vs Asynchronous Pipelines

- In a synchronous 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 asynchronous pipeline:
  - several smaller rules, each easy to write, easier to make changes
  - *downside:* sometimes rules do not fire concurrently when they should

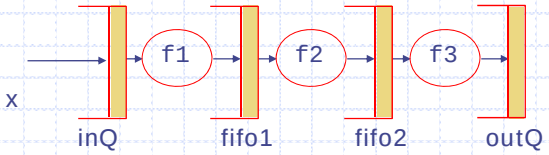
Synchronous Pipeline




```
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
```


Asynchronous pipeline

Use FIFOs instead of pipeline registers



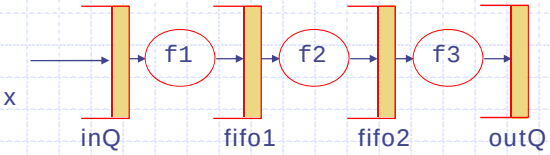

```
rule stage1 (True);
  fifo1.enq(f1(inQ.first());
  inQ.deq();  endrule
rule stage2 (True);
  fifo2.enq(f2(fifo1.first());
  fifo1.deq();        endrule
rule stage3 (True);
  outQ.enq(f3(fifo2.first());
  fifo2.deq();        endrule
```

Can all three rules fire concurrently?

Consider rules stage1 and stage2:

No conflict around inQ or fifo2.

What behavior do we want?



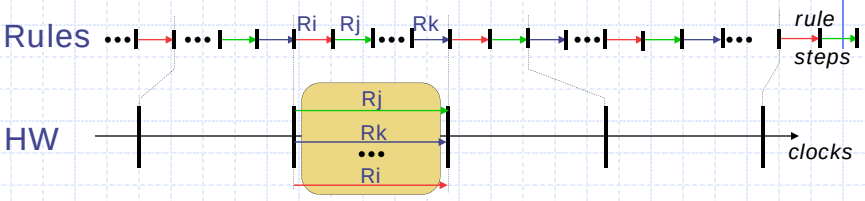
- If inQ, fifo1 and fifo2 are not empty and fifo1, fifo2 and outQ are not full then we want all three rules to fire
- If inQ is empty, fifo1 and fifo2 are not empty and fifo2 and outQ are not full then we want rules stage2 and stage3 to fire
- ...

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 wrong results can be produced

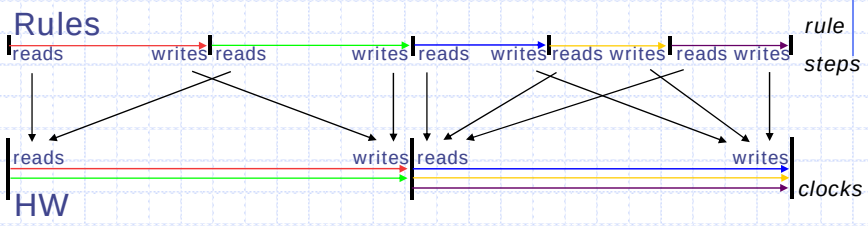
some insight into

Concurrent rule firing



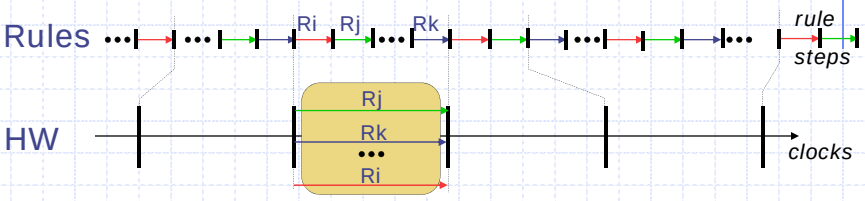
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

Parallel execution reorders reads and writes



- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks

Correctness



- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution
- Consequence: the HW can never reach a state unexpected in the rule semantics

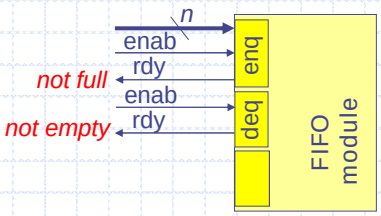
The compiler issue

- Can the compiler detect all the conflicting conditions?
  - Important for correctness
- Does the compiler detect conflicts that do not exist in reality?
  - False positives lower the performance
  - The main reason is that sometimes the compiler cannot detect under what conditions the two rules are mutually exclusive or conflict free
- What can the user specify easily?
  - Rule priorities to resolve nondeterministic choice

Implementing FIFOs

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
```



Two-Element FIFO

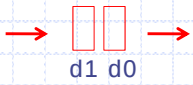
```
module mkFIFO (FIFO#(t));
  Reg#(t)    d0  <- mkRegU();
  Reg#(Bool) v0  <- mkReg(False);
  Reg#(t)    d1  <- mkRegU();
  Reg#(Bool) v1  <- mkReg(False);
  method Action enq(t x) if (!v1);
    if v0 then begin d1 <= x; v1 <= True; end
          else begin d0 <= x; v0 <= True; end endmethod
  method Action deq() if (v0);
    if v1 then begin d0 <= d1; v1 <= False; end
          else begin v0 <= False; end endmethod
  method t first() if (v0);
    return d0; endmethod
  method Action clear();
    v0<= False; v1 <= False; endmethod
endmodule
```

Assume, if there is only one element in the FIFO it resides in d0

Two-Element FIFO

another version

```
module mkFIFO (FIFO#(t));
  Reg#(t)    d0  <- mkRegU();
  Reg#(Bool) v0  <- mkReg(False);
  Reg#(t)    d1  <- mkRegU();
  Reg#(Bool) v1  <- mkReg(False);
  method Action enq(t x) if (!v1);
    v0 <= True; v1 <= v0;
    if v0 then d1 <= x; else d0 <= x; endmethod
  method Action deq() if (v0);
    v1 <= False; v0 <= v1; d0 <= d1; endmethod
  method t first() if (v0);
    return d0; endmethod
  method Action clear();
    v0<= False; v1 <= False; endmethod
endmodule
```



Assume, if there is only one element in the FIFO it resides in d0

RWire to rescue

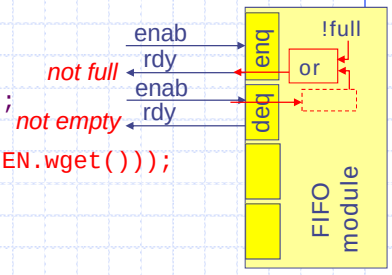
```
interface RWire#(type t);
        method Action wset(t x);
        method Maybe#(t) wget();
endinterface
```

Like a register in that you can read and write it but unlike a register

- read happens after write
- data disappears in the next cycle

One-Element Pipeline FIFO

```
module mkLFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  RWire#(void) deqEN <- mkRWire();
  Bool         deqp = isValid (deqEN.wget()));
  method Action enq(t x) if
               (!full || deqp);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False; deqEN.wset(?);
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod endmodule
```

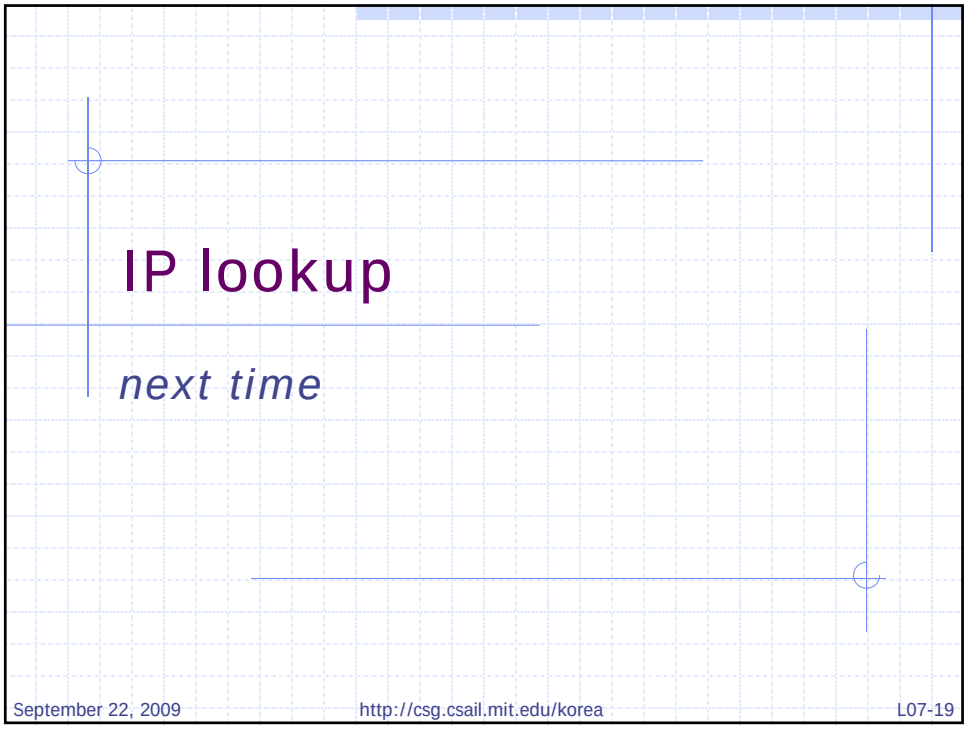


FIFOs

- Ordinary one element FIFO
  - deq & enq conflict – won't do
- Pipeline FIFO
  - first < deq < enq < clear
- Bypass FIFO
  - enq < first < deq < clear

Takeaway

- FIFOs with concurrent operations are quite difficult to design, though the amount of hardware involved is small
  - FIFOs with appropriate properties are in the BSV library
- Various FIFOs affect performance but not correctness
- For performance, concentrate on high-level design and then search for modules with appropriate properties


IP lookup
next time
September 22, 2009
http://csg.csail.mit.edu/korea
L07-19