Lecture 06: Rule Scheduling

- A rule has no internal sequencing
- Untimed rule semantics are one-rule-at-a-time steps
- Hardware intuitions in the simple one-rule-at-a-time case
- Scheduling multiple rule steps into a clock cycle
- Conflicts
- Controlling urgency, and determinism
- Example
A rule has no internal sequencing

- Recall: a rule condition is a Bool expression:
  - Purely combinational
  - No side effects (no state update)

- Recall: a rule body contains one or more *Actions*, all of which execute simultaneously (atomically) in a rule *step*

```plaintext
rule ruleName [ ( condition ) ];
  actions
endrule [: ruleName ]
```

One 'rule at a time' intuitions

- Untimed rule semantics are one-rule-at-a-time steps
- The Bluespec compiler schedules multiple rule steps into each clock, producing a timed semantics and resolving non-determinism
- But before we go there, let’s build some HW intuitions based on one-rule-at-a-time
- We’ll use the following example
  - (Can you guess what it computes? Hint: “Euclid”)

```plaintext
rule decr ( x <= y && y != 0 );
  y <= y - x;
endrule : decr

rule swap ( x > y && y != 0 );
  x <= y; y <= x;
endrule: swap
```

Answer: Euclid’s algorithm for computing GCD (greatest common divisor) of initial values in x and y registers: result is in x.
Suppose we want to execute just one rule

- The HW would be quite simple

```
rule decr ( x <= y && y != 0 );
y <= y - x;
endrule : decr
```

Two mutually exclusive rules

- (Later: what to do if they’re not mutually exclusive)

```
rule decr ( x <= y && y != 0 );
y <= y - x;
endrule : decr

rule swap ( x > y && y != 0 );
x <= y; y <= x;
endrule: swap
```
Observations: manifest and generated circuits

- The state elements are taken directly from the source code (no implicit state)
- The rule condition and action logic are taken directly from the source code
- To control execution of rules, we add two pieces of combinational logic:
  - *Data select* logic:
    - Inputs: candidate "next state" values from all rule bodies
    - Outputs: mux inputs into actual "next state" values (reg D inputs)
  - *Scheduler* logic:
    - Inputs: rule condition outputs
    - Outputs: control inputs of muxes in data select block, and state "enables"
- The scheduler is combinational control logic for the rules

---

Observations: datapaths and control logic

- Each rule specifies part of the datapath
- The complete datapath is obtained from all the rules, together with the *data select* block
- Each rule specifies part of the control logic (rule conditions)
- The complete control logic is obtained from all the rules, together with the *scheduler*
From one-rule-at-a-time to multiple-rules-per-clock-cycle

- **Rules**
  - Semantics: “Untimed” (one rule at a time)

- **Scheduling and Synthesis**
  - By BSV compiler

- **Verilog RTL**
  - Semantics: clocked synchronous HW (multiple rules per clock)

**Associated Verification**
- Using Rule Semantics, establish functional correctness
- Using Schedules, establish performance correctness

---

From one-rule-at-a-time to multiple-rules-per-clock-cycle

- What does it mean to execute multiple rules in a clock cycle? (Terminology: “concurrently”)
- In principle, *any* set of rules can be composed to execute concurrently (assuming we could meet timing)

**data select logic**

- current state
- next state

**scheduler logic**

- Rule1
- Rule2

**Notation:**
- “Rule1<Rule2”, Rule1 before Rule2

---

Copyright © Bluespec Inc. 2005-2008
L06 - 9
Practical rule composition

- In practice, we only do restricted rule compositions
- Every rule executes entirely within one cycle
- A rule fires at most once in a cycle (no Rule1<Rule1)
- Greedy: as many rules as possible per clock cycle
- Rule1 body does not feed into Rule2 body
  - Therefore, Rule2 can use state from beginning of cycle
- Later = Rule1 body can feed into Rule2 body using wires (but this has timing and scheduling implications)

Practical Rule Composition

- No intra-cycle communication between rules, in the HW
- Correctness: HW scheduler only allows rules Ri, Rj, ..., Rk to execute concurrently when net state change is equivalent to Ri<Rj<...<Rk
- Thus, never get into inconsistent state (no race conditions)
  - Every state in the HW exists in the one-rule-at-a-time semantics
Practical rule composition: why?

- Our restrictions on rule composition exist for pragmatic reasons
- Predictability and transparency: designer has good intuition about HW that will be produced from a set of rules
  - Intuitive and natural mapping to HW (edge-triggered state)
- Timing predictability: arbitrary rule composition would lengthen logic critical paths
- (Topic in later lecture: there is a designer-controlled mechanism—RWire—allowing more complex rule compositions)

Multiple Rules and Conflicts

- If two rules are enabled in a particular cycle, what prevents them from executing concurrently?
- Answer: *conflicts*
  - Since state read/write ordering is different in rule-at-a-time semantics compared to concurrent execution, certain concurrent executions would result in inconsistent states
  - Certain *resource sharings* prevent concurrency
No conflict

- Rule untimed semantics:
  - r1 and then r2 ("r1<r2"), or
  - r2 and then r1 ("r2<r1")

- HW: concurrent execution of r1 and r2
  - Each rule reads state (x, y) at start of cycle
  - Each rule updates state (x+1, y+2) at end of cycle

Net state change same as (r1<r2) or (r2<r1)  
⇒ ok to execute concurrently
Conflict

- Rule untimed semantics:
  - r1 and then r2 ("r1<r2"), or
  - r2 and then r1 ("r2<r1")

- HW: concurrent execution of r1 and r2
  - Each rule reads state (y,x) at start of cycle
  - Each rule updates state (y+1, x+2) at end of cycle

Net HW state change ≠ any order (r1<r2 or r2<r1)  ➔ not ok to execute concurrently

A rule is not a Verilog “always” block!
Bluespec HW scheduler will prevent these firing together

Resource conflict

In HW, cannot simultaneously store into x from two rules  ➔ not ok to execute concurrently

Bluespec HW scheduler will prevent these firing together
Resource conflict

rule r1 (c1);
  x <= mem.read(addr1);
endrule

rule r2 (c2);
  y <= mem.read(addr2);
endrule

In HW, can only do one mem read per cycle
\(\rightarrow\) not ok to execute concurrently

Bluespec HW scheduler will prevent these firing together

Resource conflict

rule r1 (c1);
  fifo.enq (e1);
endrule

rule r2 (c2);
  fifo.enq (e2);
endrule

In HW, fifo has only one enq port; can’t use it twice per cycle
\(\rightarrow\) not ok to execute concurrently

Bluespec HW scheduler will prevent these firing together
Scheduler incorporates conflict analysis by compiler
- CAN_FIRE is True → (rule_condition && all_ready_conditions)
- WILL_FIRE is True → CAN_FIRE && ! (WILL_FIRE of any higher priority rules that conflict with this rule)

Controlling ordering: urgency

```
(* descending_urgency = "r1, r2" *)
rule r1 (c1);
  fifo.enq (e1); // one enq per cycle
endrule

rule r2 (c2);
  fifo.enq (e2); // one enq per cycle
endrule
```

Urgency means the order in which the compiler determines if rules will fire or not.

Bluespec HW scheduler will determine if r1 will fire
Or not before even considering r2
But here, only one can fire so r1 fires if both 'can'
Controlling execution: preempts

Some times it is necessary to indicate that a if a rule fires then another/s should not fire

```plaintext
(* preempts = "r1, r2" *)
rule r1 (upA);
  x <= x + 3;
endrule
rule r2;
  y <= y + 1;
endrule
```

- If upA then increment x by 3
- Increment y only if x is not updated

Bluespec HW scheduler will prevent these firing together
r1 preempts r2

Controlling execution: execution order

Some times it is necessary to indicate that two rules should fire in a specific order in the schedule.

```plaintext
(* execution_order = "r1, r2" *)
rule r1;
  x <= 5;
endrule
rule r2;
  y <= 6;
endrule
```

We'll need to see a few special cases to see when this is really helpful

Bluespec HW scheduler will ensure these fire in the defined order
Asserting Execution: Mutually Exclusive

Some times it is necessary to indicate that two rules are mutually exclusive

- Example: Set to 1 the bit indicated by a one-hot signal

```
(* mutually_exclusive = "updateBit0, updateBit1" *)

rule updateBit0 (oneHotNumber[0] == 1);
  x[0] <= 1;
endrule

rule updateBit1 (oneHotNumber[1] == 1);
  x[1] <= 1;
endrule
```

Bluespec HW scheduler will not introduce any additional logic for conflict solving of shared resource x

Determinism

- The one-rule-at-a-time semantics is non-deterministic ("choose any enabled rule")

- But the HW generated by the Bluespec compiler is fully deterministic
  - Urgency resolved between conflicting rules

- We only execute (and verify) deterministic HW
Example: Pipelined Shifter

- **Goal:** implement: \( y = \text{shift}(x, s) \)
  where \( y \) is \( x \) shifted by \( s \) positions.
  Suppose \( s \) is a 3-bit value.

- **Strategy:**
  - Shift by \( s = \text{shift by 4 } (=2^2) \)
    and by \( 2 \ ) if \( s[2] \) is set,
    and by \( 1 \ ) if \( s[1] \) is set
  - A shift by \( 2^1 \) is trivial: it’s just a “lane change” made purely with wires

```

```

Synchronous Pipeline with Regs

```

```

```

```

rule all_together;
    sx1 <= step_1(sx0);
    sx2 <= step_2(sx1);
    sx3 <= step_3(sx2);
endrule

```
Asynchronous Pipeline with FIFOs (Regs with interlocks)

```
rule stage_1;
    sx0 <- fifo0.pop;  fifo1.enq (step_1(sx0));
endrule

rule stage_2;
    sx1 <- fifo1.pop;  fifo2.enq (step_2(sx1));
endrule

rule stage_3;
    sx2 <- fifo2.pop;  fifo3.enq (step_3(sx2));
endrule
```

Remarks

- In the synchronous pipeline, we compose actions in parallel
  - All stages move data simultaneously, in lockstep (atomic!)

- In the asynchronous pipeline, we compose rules in parallel
  - Stages can move independently (each stage can move when its input FIFO has data and its output FIFO has room)
  - If we had used parallel action composition instead, all stages would have to move in lockstep, and could only move when all stages were able to move

- Your design goals will suggest which kind of composition is appropriate in each situation
Summary: rule scheduling

- **Scheduler** ensures consistency with Rule semantics
- Usually the most error-prone part of hand-written RTL
  - Here, correct by construction
- Bluespec patented technology

Bluespec synthesis only adds this part

End of Lecture