Constructive Computer Architecture

Modules with Guarded Interfaces

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

- Make the life of the programmers easier: Include some checks (readyness, fullness, ...) in the method definition itself, so that the user does not have to test the applicability of the method from outside

**Guarded Interface:**
- Every method has a *guard* (rdy wire)
- The value returned by a method is meaningful only if its guard is true
- Every action method has an *enable signal* (en wire) and it can be invoked (en can be set to true) only if its guard is true

```verilog
interface Fifo#(numeric type size, type t);
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
```

notice, en and rdy wires are implicit
One-Element FIFO Implementation with guards

module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Reg#(Bool) v <- mkReg(False);
method Action enq(t x)
    v <= True; d <= x;
endmethod
method Action deq
    v <= False;
endmethod
method t first
    return d;
endmethod
endmodule

Notice, no semicolon turns the if into a guard
Rules with guards

- Like a method, a rule can also have a guard

```plaintext
rule foo (p);
begin x1 <= e1; x2 <= e2 end
endrule
```

- A rule can execute only if its guard is true, i.e., if the guard is false the rule has no effect

- True guards can be omitted

No if before the guard for rules!
Streaming a function using a FIFO with guarded interfaces

The implicit guards of the method call are sufficient here.
rule switch;
    if (inQ.notEmpty)
        if (inQ.first.color == Red) begin$_1$
            if (redQ.notFull) begin$_2$
                redQ.enq(inQ.first.value); inQ.deq;
            end$_2$
        end$_1$
        else begin$_3$
            if (greenQ.notFull) begin$_4$
                greenQ.enq(inQ.first.value); inQ.deq;
            end$_4$
        end$_3$
endrule

All the red stuff can be deleted.
Switch using FIFOs with guarded interfaces

```
rule switch;
  if (inQ.first.color == Red) begin
    redQ.enq (inQ.first.value); inQ.deq;
  end else begin
    greenQ.enq(inQ.first.value); inQ.deq;
  end
endrule
```

What is the implicit guard?

```
```
Switch using FIFOs with guarded interfaces

```
rule switch;
    if (inQ.first.color == Red) begin
        redQ.enq (inQ.first.value); inQ.deq;
    end
    else begin
        greenQ.enq(inQ.first.value); inQ.deq;
    end
endrule
```

Does this code still work?
GCD with and without guards

Interface without guards

Interface with guards

```
interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
    method Bool busy;
    method Bool ready;
endinterface
```
Using GCD module with guarded interfaces

A rule can be executed only if guards of all of its actions are true
GCD with guarded interfaces
implementation

module mkGCD (GCD);
Reg#(Bit#(32)) x <- mkReg(0);
Reg#(Bit#(32)) y <- mkReg(0);
Reg#(Bool) busy <- mkReg(False);

rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x <= y; y <= x; end //swap
endrule

method Action start(Bit#(32) a, Bit#(32) b) if (!busy);
x <= a; y <= b; busy <= True;
endmethod

method ActionValue (Bit#(32)) getResult if (x==0);
    busy <= False; return y;
endmethod
endmodule

interface GCD;
    method Action start
        (Bit#(32) a, Bit#(32) b);
    method ActionValue (Bit#(32))
        getResult;
endinterface

Assume b /= 0
Guards vs Ifs

**method Action enq(t x) if (!v);**

v <= True; d <= x;

**endmethod**

**guard is !v; enq can be applied only if v is false**

**versus**

**method Action enq(t x);**

**if (!v) begin v <= True; d <= x; end**

**endmethod**

**guard is True, i.e., the method is always applicable.**

**if v is true then x would get lost; bad**
Pipelining combinational circuits
Pipelining Combinational Functions

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
Elastic pipeline
Use FIFOs instead of pipeline registers

When can stage1 rule fire?
- inQ has an element
- fifo1 has space

Can tokens be left in the pipeline?
No

Can these rules execute concurrently?
Elastic pipeline

If these rules cannot execute concurrently, it is hardly a pipelined system

When can rules execute concurrently?

What hardware is synthesized to execute rules concurrently?
Multi-rule Systems

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

Non-deterministic choice; User annotations can be used in rule selection

One-rule-at-a-time-semantics: Any legal behavior of a Bluespec program can be explained by observing the state updates obtained by applying only one rule at a time

However, for performance we execute multiple rules concurrently whenever possible

stay tuned ...