Conflict Matrix (CM)
BSV compiler generates the pairwise conflict information

Example 1
rule ra;  
x <= x+1;  
endrule
rule rb;  
y <= y+2;  
endrule

Example 2
rule ra;  
x <= y+1;  
endrule
rule rb;  
y <= x+2;  
endrule

Example 3
rule ra;  
x <= y+1;  
endrule
rule rb;  
y <= y+2;  
endrule

<table>
<thead>
<tr>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
</tr>
<tr>
<td>rb</td>
<td>CF</td>
</tr>
<tr>
<td>ra</td>
<td>C</td>
</tr>
<tr>
<td>rb</td>
<td>C</td>
</tr>
<tr>
<td>rb</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

- C : rules can’t be executed concurrently
- CF: rules can be executed concurrently; the net effect is the same as if ra executed before rb (ra<rb) or (rb<ra)
- ra < rb : rules can be executed concurrently; the net effect is as if ra executed before rb

Concurrent rule execution
- This circuit will execute rules ra and rb concurrently
- This circuit is correct only if rules ra and rb do not conflict (⇒ methods f and g of m do not conflict)
- Suppose rules ra and rb do conflict!

Need for a scheduler
- Guards of all rules are fed to a scheduler
- Using the CM, the scheduler lets only non-conflicting rules proceed
- Scheduler is a pure combinational circuit with a small number of gates
- A correct but low performance scheduler may schedule only one rule at a time
The rule scheduler

Guards (gs1 ... gsn) of many rules may be true simultaneously, and some of them may conflict.

BSV compiler constructs a combinational scheduler circuit with the following property:

- for all i and j, if wfsi and wfsj are true then the corresponding gsi and gsj must be true and rules i and j must not conflict with each other.

Example schedulers

<table>
<thead>
<tr>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>CF</td>
</tr>
<tr>
<td>rb</td>
<td>C</td>
</tr>
<tr>
<td>ra</td>
<td>C</td>
</tr>
<tr>
<td>rb</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

The effect ra<rb would be achieved automatically.

Example schedulers

- ra
- rb

- ra
- rb

- ra
- rb

- ra
- rb

- ra
- rb

- ra
- rb

- ra
- rb

- ra
- rb

Take-home problem

- Draw the hardware circuit for this design

```
rule stage1;
  fifo.enq(f0(inQ.first));
  inQ.deq;
endrule
rule stage2;
  outQ.enq(f1(fifo.first));
  fifo.deq;
endrule
```

Additional Problem

- Can all three rules execute concurrently?
- Can any two rules execute concurrently?

```
rule ra(p1);
  x <= y + 1;
endrule
rule rb(p2);
  y <= z + 2;
endrule
rule rc(p3);
  z <= x + 2;
endrule
```
Epoch: a method to manage control hazards

- Attach an epoch (color) to each instruction as it flows through the pipeline
- Change the epoch if the Execute stage detects a misprediction and set the pc to the correct pc
- Discard instructions with the old epoch, that is, if its epoch does not match the current epoch

Two-Stage Pipeline processor
first attempt - code

```plaintext
rule doFetch;
  iMem.req(MemReq{op: Ld, addr: pc, data: ?});
  let ppc = pc + 4;  pc <= ppc;
  f2d.enq(F2D {pc: pc, ppc: ppc, epoch: epoch});
endrule

rule doExecute if (state == Execute);
  let inst <- iMem.resp;
  let x = f2d.first; f2d.deq;
  let pcD = x.pc; let ppc = x.ppc; let epochD = x.epoch;
  if (epochD == epoch) begin // right-path instruction
    code to compute eInst from inst
    let mispred = eInst.nextPC != ppc;
    if (mispred) begin pc <= eInst.nextPC; epoch <= !epoch; end
    code to update the state;
    in case of a memory op, initiate memory req and
    in case of Ld go to LoadWait
  end
endrule

rule doLoadWait if (state == LoadWait); ... go to Execute ...
```

mkProcTwoStageBad analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Multi-Cycle (Cycles)</th>
<th>Two-Stage Bad (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd_sw.S</td>
<td>3022</td>
<td>4403</td>
</tr>
<tr>
<td>no_hazard.S</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>all_control_hazard.S</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

Why?

- doFetch and doExecute conflict! Can't execute concurrently
- Fix?
Rule Splitting

**Rule Splitting**

- **Rule one:**
  - \( r_1 \leq e_1 \)
- **Rule two:**
  - if \( p \) then \( r_2 \leq e_2 \) else \( r_1 \leq e_3 \)

- **Split rule two:**
  - split rule two into two \( T \) and two \( F \)
  - rules two \( T \) and one do not conflict and can execute together

- **Difficult to turn \( p \) into a guard**
  - Create a separate rule for the conflicting action and delay its execution until the next cycle
  - Requires a register (\( p_{\text{Reg}} \)) to remember \( p \)

Another way of Rule Splitting

- **Rule one:**
  - \( r_1 \leq e_1 \)
- **Rule two:**
  - more actions;
  - \( p = e \)
  - if \( p \) then \( r_2 \leq e_2 \) else \( r_1 \leq e_3 \)

- **Split rule two:**
  - split rule two into two \( T \) and two \( F \)
  - rules two \( T \) and one do not conflict and can execute together

Fixing the two-stage pipeline

- **Rule doFetch:**
  - \( \ldots \; pc \leq ppc; \ldots \)
- **Rule doExecute if \( \text{state} \neq \text{LoadWait} \):**
  - \( \ldots \)
  - let mispred = eInst.nextPC != ppc;
  - if (mispred) then begin \( pc \leq eInst.nextPC; \) epoch <= !epoch; \end
  - code to initiate memory ops and go to LoadWait if necessary
- **Rule doLoadWait if \( \text{state} = \text{LoadWait} \):**
- **Rule doRedirect**

- **Improved two-stage pipeline**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Multi-Cycle</th>
<th>Two-Stage Bad</th>
<th>Two-Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd_sw.S</td>
<td>3022</td>
<td>4403</td>
<td>4317</td>
</tr>
<tr>
<td>no_hazard.S</td>
<td>26</td>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>all_control_hazard.S</td>
<td>16</td>
<td>31</td>
<td>37</td>
</tr>
</tbody>
</table>

- With no hazard, the pipeline behaves perfectly and throughput doubles
- With control hazard, we have two-cycle penalty and perform worse than Multi-cycle
- Improvement 1: a cycle can be saved in case of redirection by initiating the instruction fetch from the redirection rule itself
- Improvement 2: Bypass \( pc \) and epoch updates from doExecute to doFetch (need EHRs - L23)

Lab 7
L18 Review - Managing Data Hazards through a Scoreboard

Splitting Execute into two stages

- Execute step probably has the longest propagation delay (decode + register-file read + execute)
- Separate Execute into two stages:
  - Decode and register-file-read
  - Execute – including the initiation of memory instructions
- Will require us to introduce several new concepts in pipelining

Dealing with data hazards

- Detect data hazard (aka read-after-write (RAW) hazard)
  - Compare sources in the decoded inst with the destinations of older instructions in the pipeline
  - Requires a Scoreboard -- a data structure to keep track of the destinations of the instructions in the pipeline beyond the Decode stage
- Stall the Decode from dispatching the instruction as long as RAW hazard prevails
  - RAW hazard will disappear as the pipeline drains

Splitting doExecute

rule doFetch; ... endrule

rule doDecode;
  let inst <- iMem.resp;
  ... filter wrong-path instructions ...
  ... decode (inst) ...
  ... read rf ...
  d2e.enq(...);
endrule

rule doExecute (...);
  let dInst = d2e.first; d2e.deq;
  ... filter wrong-path instructions ...
  ... execute (dInst, rval1,rval2, pc) ...
  ... detect and handle misprediction ...
  ... launch memory instructions if needed ...
  ... save info for the LoadWait step ...
endrule

rule doLoadWait(...) ... endrule
**Scoreboard**

- **method insert(dst):** inserts the destination of an instruction or Invalid in the scoreboard when the instruction is decoded.
- **method search1(src):** searches the scoreboard for a data hazard, i.e., a dst that matches src.
- **method search2(src):** same as search1.
- **method remove:** deletes the oldest entry when an instruction commits.

We will provide you a scoreboard implementation.

---

**Fixing the doDecode rule**

```
rule doDecode;
  let inst <- iMem.resp;
  let x = f2d.first; f2d.deq;
  if (x.epoch == epoch) begin
    let dInst = decode2(inst); // src1 and src2 are Maybe types;
    if (!((sb.search1(dInst.src1)||sb.search2(dInst.src2))) begin
      read rVal1 and read rVal2 from rf
      sb.insert(dInst.dst); //to stall future inst for data hazard
      enqueue into e2d fifo: pc, ppc, epoch, rVal1, rVal2, dInst
    end
  end
endrule
```

No new instruction should be fetched in the stalled state.

Still not quite correct. Why?

We need to keep the fetched instruction while stalling!

Need a register to hold the fetched instruction while stalling.

---

**Execute rule**

```
rule doExecute (...);
  ... filter wrong-path instructions ...  
  ... execute (dInst, rval1, rval2, pc) ... 
  ... detect and handle misprediction ... 
  ... launch memory instructions if needed ... 
  ... save info for the LoadWait step ... 
endrule
```

**Rule doLoadWait (...):**

```
... 
endrule
```

sb.remove has to be inserted whenever an instruction completes execution.

---

Lab 7