Multicycle Operations

Daniel L. Rosenband
danlief@mit.edu
Laboratory for Computer Science
MIT

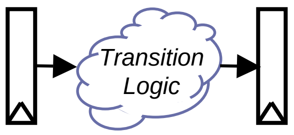
January 17, 2003

Outline

- What are multicycle operations
- Compiling multicycle operations
- Challenges
- Implementation strategy

Multicycle Operations

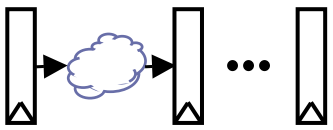
Multicycle Path



Combinational Delay > 1 cycle

- Examples:
  - Multiply
  - Slow memory
- Challenges:
  - Atomicity
  - Rest of the design should not stall

Pipelined Multicycle Operation



Multiple stages that form a pipeline

- Examples:
  - Processor pipeline
  - Resource limitations
- Challenges:
  - Atomicity
  - Throughput

Execution Semantics

```
when π ==>
   action
     r1 := r2 + r3
     r4 := 0
```

- Execute in a single cycle

```
when π ==>
   actionMCPath 4
     r1 := r2 * r3
```

- Execute over many cycles

```
when π ==>
   actionPipe
     action
       r1 := rf.read 0
     action
       r2 := rf.read 1
     action
       rf.write 2 (r1+r2)
```

- Actions execute in sequence
- Each action executes in a single cycle
- Entire rule is atomic with respect to other rules

- Rules execute atomically (all cases)
  - Final state must match a sequential execution of the rules

Why use multicycle rather than single cycle operations?

- Multicycle Paths
  - Some structures not easily pipelineable
  - Block being interfaced to has a long combinational delay
- Pipelined Multicycle Operations
  - Natural to express many hardware structures as a sequence of events
  - Atomicity / coherence enforced by the compiler
  - Automate buffer sizing(?)

Burden on User for Correctness

- Consider rule interactions to determine if there is a coherence problem – create interlocks
- Ensure that ordering is maintained

$\rightarrow$

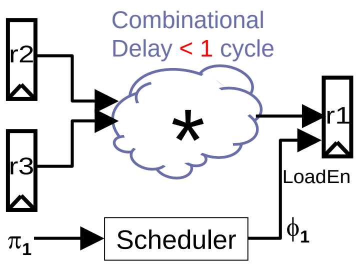
Burden on Compiler for Performance

- Relax constraints on parallel execution while maintaining correctness

Multicycle Path Compilation

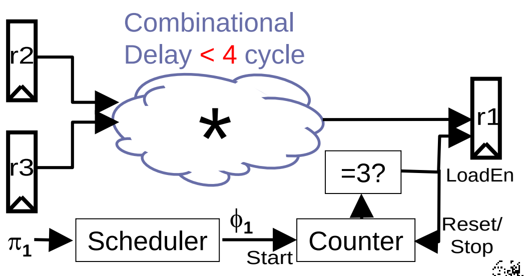
Original Circuit:

when π_1 **==>**
  action
    r1 = r2 * r3



Multicycle Path Circuit:

when π_1 **==>**
  actionMCPath 4
    r1 := r2 * r3



Multicycle Path Compilation (continued)

- Compilation Strategy
  - Source to source transformations
  - Break multicycle rule into multiple single cycle rules

```
when π1 ==>
   actionMCPath 4
     r1 := r2 * r3
```

$\longrightarrow$

```
when π1 ==>
   action
     counter.start
when (counter.val == 3) ==>
   action
     r1 := r2 * r3
     counter.reset_and_stop
```

Is this correct? **NO!!!**

- Need to protect the state
  - What happens when another rule tries to read / write r1?
  - What happens when another rule tries to write to r2/r3?

Multicycle Path Compilation (continued)

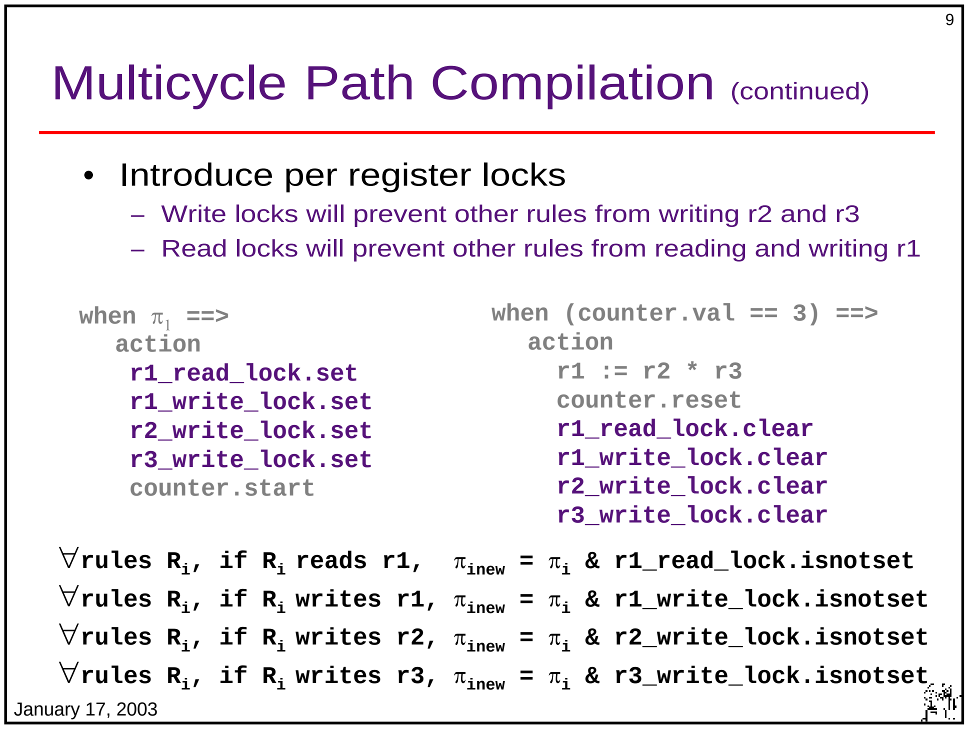
- Introduce a global lock
  - Set lock when multicycle path rule begins executing
  - Clear lock when multicycle path rule finishes executing
  - No other rule can execute while the global lock is set

```
when π1 ==>
   action
     global_lock.set
     counter.start
```

```
when (counter.val == 3) ==>
   action
     r1 := r2 * r3
     counter.reset
     global_lock.clear
```

$\forall$**rules** R_i, $\pi_{inew} = \pi_i$ **& global_lock.isnotset**

- Performance is poor since the entire system stalls when the multicycle path rule is executing



Multicycle Path Compilation (continued)

- Introduce per register locks
  - Write locks will prevent other rules from writing r2 and r3
  - Read locks will prevent other rules from reading and writing r1

```
when π1 ==>
  action
    r1_read_lock.set
    r1_write_lock.set
    r2_write_lock.set
    r3_write_lock.set
    counter.start
```

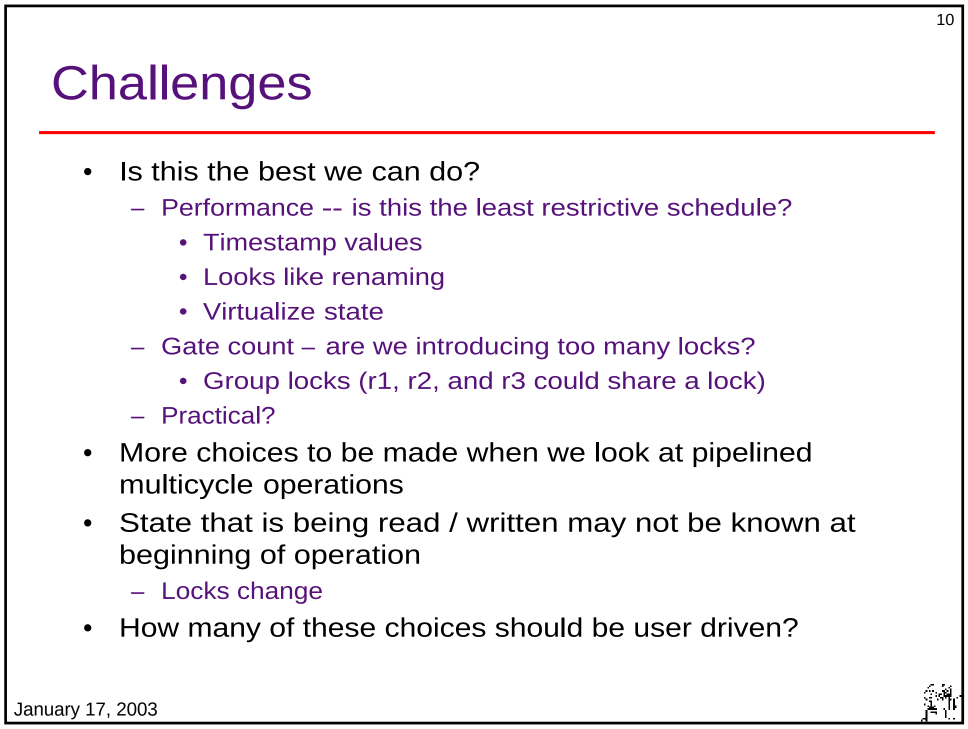
```
when (counter.val == 3) ==>
  action
    r1 := r2 * r3
    counter.reset
    r1_read_lock.clear
    r1_write_lock.clear
    r2_write_lock.clear
    r3_write_lock.clear
```

$\forall$**rules** R_i**, if** R_i **reads r1,** $\pi_{inew} = \pi_i$ **& r1_read_lock.isnotset**

$\forall$**rules** R_i**, if** R_i **writes r1,** $\pi_{inew} = \pi_i$ **& r1_write_lock.isnotset**

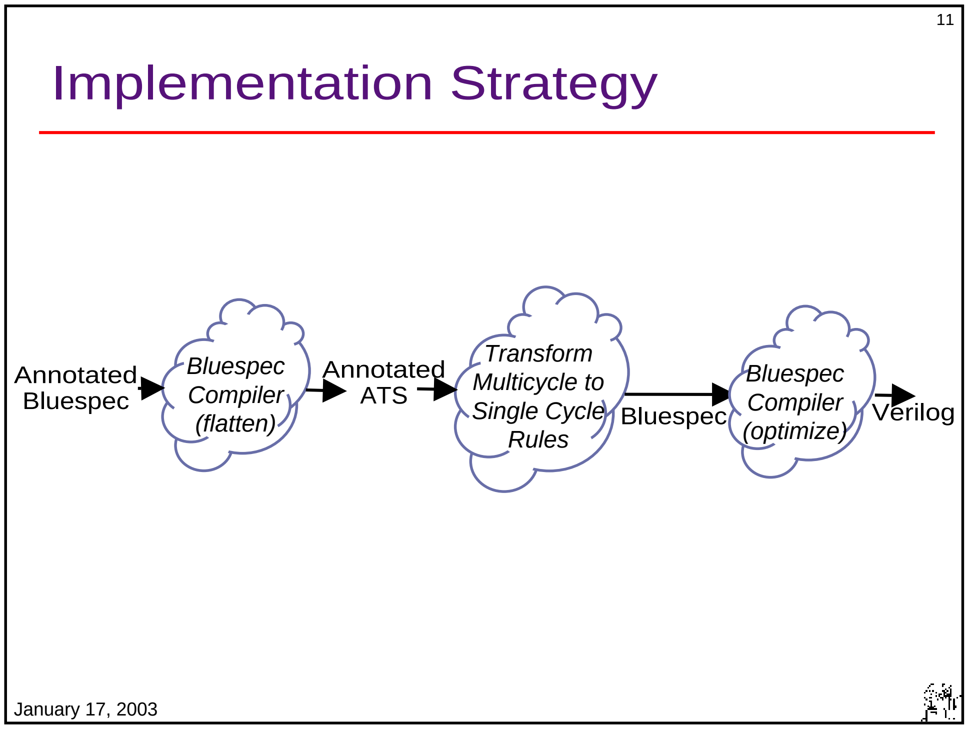
$\forall$**rules** R_i**, if** R_i **writes r2,** $\pi_{inew} = \pi_i$ **& r2_write_lock.isnotset**

$\forall$**rules** R_i**, if** R_i **writes r3,** $\pi_{inew} = \pi_i$ **& r3_write_lock.isnotset**

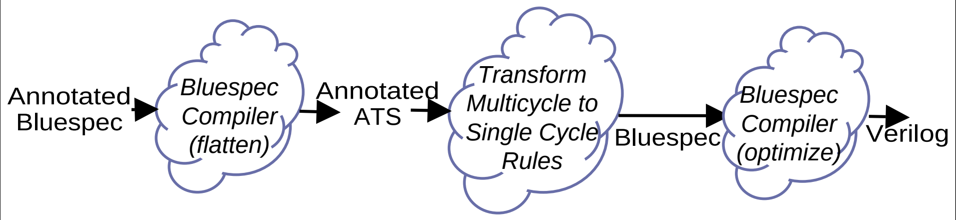


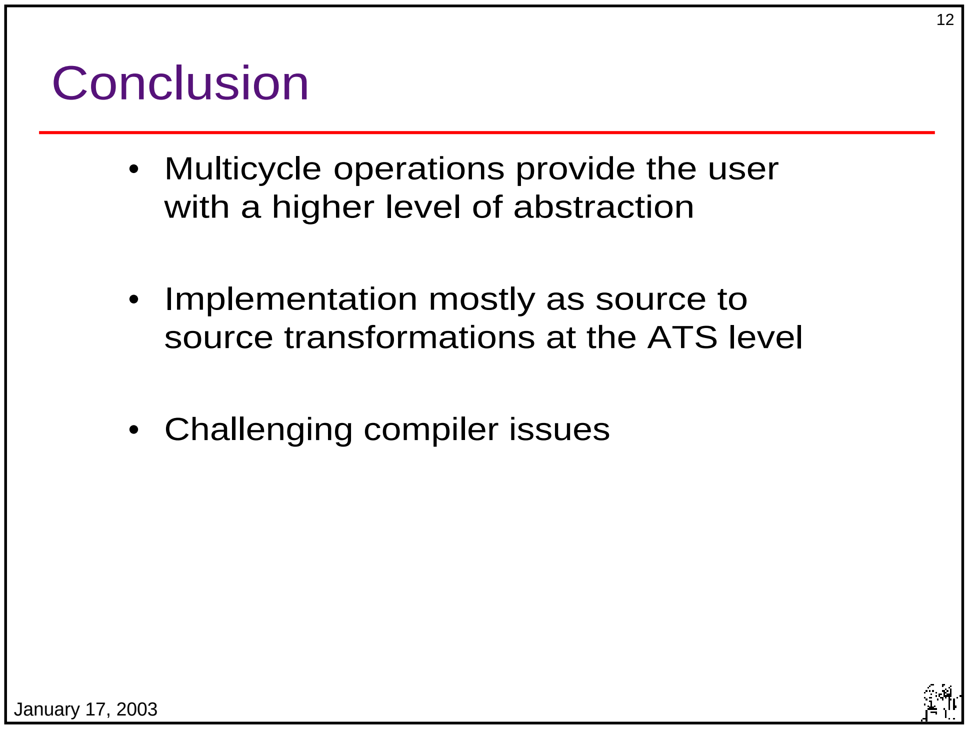
Challenges

- Is this the best we can do?
  - Performance -- is this the least restrictive schedule?
    - Timestamp values
    - Looks like renaming
    - Virtualize state
  - Gate count – are we introducing too many locks?
    - Group locks (r1, r2, and r3 could share a lock)
  - Practical?
- More choices to be made when we look at pipelined multicycle operations
- State that is being read / written may not be known at beginning of operation
  - Locks change
- How many of these choices should be user driven?



Implementation Strategy


Annotated Bluespec
Bluespec Compiler (flatten)
Annotated ATS
Transform Multicycle to Single Cycle Rules
Bluespec
Bluespec Compiler (optimize)
Verilog



Conclusion

- Multicycle operations provide the user with a higher level of abstraction
- Implementation mostly as source to source transformations at the ATS level
- Challenging compiler issues