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

<table>
<thead>
<tr>
<th>when ( \pi \implies \Action )</th>
<th>when ( \pi \implies \Action_{MCPath} 4 )</th>
<th>when ( \pi \implies \Action_{Pipe} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Action )</td>
<td>( \Action )</td>
<td>( \Action )</td>
</tr>
<tr>
<td>( r1 := r2 + r3 )</td>
<td>( r1 := r2 \times r3 )</td>
<td>( r1 := \text{rf}.read 0 )</td>
</tr>
<tr>
<td>( r4 := 0 )</td>
<td></td>
<td>( r2 := \text{rf}.read 1 )</td>
</tr>
</tbody>
</table>

- Execute in a single cycle
- Execute over many cycles
- 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

**Burden on Compiler for Performance**
- Relax constraints on parallel execution while maintaining correctness

---

**Multicycle Path Compilation**

**Original Circuit:**

\[
\text{when } \pi_1 \Rightarrow \text{action } r1 = r2 \times r3
\]

**Multicycle Path Circuit:**

\[
\text{when } \pi_1 \Rightarrow \text{actionMCP}ath 4 \quad r1 := r2 \times r3
\]
Multicycle Path Compilation (continued)

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

  \[
  \text{when } \pi_i, \Rightarrow \\
  \text{actionMCP} 4 \\
  r1 := r2 \ast r3
  \]

  \[
  \text{when } (\text{counter.val} == 3), \Rightarrow \\
  \text{action} \\
  r1 := r2 \ast r3 \\
  \text{counter.reset} \\
  \text{global_lock.clear}
  \]

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

  \[
  \text{when } \pi_i, \Rightarrow \\
  \text{action} \\
  \text{global_lock.set} \\
  \text{counter.start}
  \]

  \[
  \text{when } (\text{counter.val} == 3), \Rightarrow \\
  \text{action} \\
  r1 := r2 \ast r3 \\
  \text{counter.reset} \\
  \text{global_lock.clear}
  \]

\[\forall \text{rules } R_i, \pi_{\text{inew}} = \pi_i \& \text{global_lock.isnotset}\]

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

January 17, 2003
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

\[
\begin{align*}
\text{when } \pi_1 & \implies \\
\text{action} & \\
r1\_read\_lock\_set & \\
r1\_write\_lock\_set & \\
r2\_write\_lock\_set & \\
r3\_write\_lock\_set & \\
counter\_start & \\
\text{when } (\text{counter.val} == 3) & \implies \\
\text{action} & \\
r1 & := r2 \times r3 & \\
counter\_reset & \\
r1\_write\_lock\_clear & \\
r2\_write\_lock\_clear & \\
r3\_write\_lock\_clear & \\
\end{align*}
\]

∀ rules \( R_i \), if \( R_i \) reads \( r1 \), \( \pi_{i_{\text{new}}} = \pi_i \& r1\_read\_lock\_isnotset \)

∀ rules \( R_i \), if \( R_i \) writes \( r1 \), \( \pi_{i_{\text{new}}} = \pi_i \& r1\_write\_lock\_isnotset \)

∀ rules \( R_i \), if \( R_i \) writes \( r2 \), \( \pi_{i_{\text{new}}} = \pi_i \& r2\_write\_lock\_isnotset \)

∀ rules \( R_i \), if \( R_i \) writes \( r3 \), \( \pi_{i_{\text{new}}} = \pi_i \& r3\_write\_lock\_isnotset \)

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

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