Motivation

- Processor Scaling Wall
  - Costly communication over slow wires
  - Processing speed fundamentally limited by wire delay
- Von Neumann architecture is limited
  - Executes commands sequentially via a program counter
  - Loads instructions from memory
    - Remember those slow wires?
  - Tricky to execute instructions in parallel
Objectives of WaveScalar

- High performance dataflow execution
- Minimize communication costs
  - Distributed execution model
- Allow for parallelism
  - Remove sequential restraints inherent in von Neumann architecture
WaveScalar

- A dataflow ISA
- Compiled programs execute according to the dataflow firing rule, rather than sequentially

![Diagram](image)

D = (A + B) / (C - 2)

.label begin
Add   temp_1 ← A, B
Sub   temp_2 ← C, #2
Div   D ← temp_1, temp_2

Fig. 1. A simple dataflow fragment: (a) a simple program statement; (b) its dataflow graph; and (c) the corresponding WaveScalar assembly. The order of the WaveScalar assembly statements is unimportant, since they will be executed in dataflow fashion.
Control Instructions

- WaveScalar uses special instructions to conditionally direct data flow.

![Control Flow Diagram](image.png)

Fig. 2. Implementing control in WaveScalar: (a) an If-THEN-ELSE construct and equivalent dataflow representations; (b) Steer instructions (triangles labeled “s”) ensure that only one side of the branch executes, while (c) computes both sides and a ϕ instruction selects the result to use.
Waves

- Programs are broken into “waves”
  - Single entry, non-looping segments of code
  - Inner body of a loop, for example
    - Iterations are distinct “dynamic waves,” which may execute out of order (but still following the dataflow firing rule)
  - Wave Numbers differentiate dynamic waves
Wave-Ordered Memory

- Within a wave, the WaveScalar compiler statically assigns unique sequence numbers to memory operations
- Determines “links”
  - The set of a memory operation’s predecessor, own sequence number, and successor
  - Ex: < 3, 4, 8 >
  - The ‘?’ wildcard may be used if branches make the predecessor or successor statically unknown
Wave-Ordered Memory

- Together, the links inside a wave form a control graph
- No Gap Rule
  - Execution must follow one of these paths
WaveCache

- Grid of 2k Processing Elements (PEs)
  - Each may hold up to 8 different instruction
  - Instructions are executed by PEs, as determined during program execution
Performance

- Single-threaded comparison to a superscalar arch.
Performance

- Speedup by number of threads
Comparison to TRIPS

• Similarities
  • Both are dataflow architectures
  • Like TRIPSs “blocks,” programs are broken into waves that execute according to the dataflow firing rule.
    • But waves are not ordered, and may execute in parallel
Comparison to TRIPS

- Contrasts
  - TRIPS, while integrating dataflow principles and techniques, is fundamentally a von Neumann architecture
    - Hyperblocks are sequenced by a program counter
  - WaveScalar completely removes the program counter and relies entirely on dataflow dependencies
Evaluation of WaveScalar

- High performance dataflow execution
- Minimize communication costs
  - Distributed execution model
- Allow for parallelism
  - Remove sequential restraints inherent in von Neumann architecture

Executes according to dataflow dependencies instead of program counter
Evaluation of WaveScalar

- High performance dataflow execution
- Minimize communication costs
  - Distributed execution model
- Allow for parallelism
  - Remove sequential restraints inherent in von Neumann architecture

The WaveCache structure removes the need to fetch an instruction for each execution, as they are already bound to specific PEs.
Evaluation of WaveScalar

- High performance dataflow execution
- Minimize communication costs
  - Distributed execution model
- Allow for parallelism
  - Remove sequential restraints inherent in von Neumann architecture

Dynamic waves execute in parallel, restricted only by the dataflow itself
Evaluation of WaveScalar

- Creative approach, very different than that discussed in CSE 141
- However, WaveScalar is very complex
  - Could get messy under certain circumstances
Questions?
Wave-Ordered Memory

- To enforce the No Gap Rule, links are hashed and tested as new instructions appear.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt; ?, 1, ? &gt;</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&lt; 1, 3, 7 &gt;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&lt; ?, 7, ? &gt;</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
WaveScalar Example

```c
function s(char in[10], char out[10]) {
    i = 0;
    j = 0;
    do {
        int t = in[i];
        if (t) {
            out[j] = t;
            j++;
        }
        i++;
    } while (i < 10);
    // no more uses of i
    // no more uses of in
}
```

Figure 4: Example code fragment: This simple loop copies in into out, ignoring zeros.
WaveScalar Example

Figure 5: WaveScalar example. The RISC assembly (left) for the program fragment in Figure 4, the WaveScalar version (center), and the WaveScalar version mapped onto a small WaveCache (right). To clarify the discussion in the text, the numbers in the column label rows of instruction, and the vertical, dashed line divides the graph into two parts.