EXECUTION ALGORITHM

WHILE(AVAILABLE_OPERATIONS (STATE)) {
    STATE = EXEC(AVAILABLE_OPERATION(STATE), STATE)
}

OPERATIONS “FIRE” WHEN ALL INPUTS ARE AVAILABLE
-- ALSO KNOWN AS THE DATAFLOW FIRING RULE
E.G. STATIC DATAFLOW

TOKEN-STORE

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>{1.R,2.L}</td>
<td>#1</td>
<td>#2</td>
</tr>
<tr>
<td>MUL</td>
<td>{2.R}</td>
<td>#10</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>{}</td>
<td>#10</td>
<td></td>
</tr>
</tbody>
</table>
USE DATA FOR CONTROL

ALSO CALLED A STEER OR BRANCH

IF C == 0
SEND LEFT
ELSE
SEND RIGHT
USE DATA FOR CONTROL

IF C == 0
PASS LEFT
ELSE
PASS RIGHT

ALSO CALLED A PHI OR SELECT
DATAFLOW ASSEMBLY

- GRAPH DESCRIPTION LANGUAGE

- INSTRUCTIONS:

  - NAME: OP { TARGETS }

  - OP { TARGETS } { SOURCES1 } { SOURCES 2 }

  - E.G.

     - ADD { OUTPUT }, { SRC1, SRC2 }, { SRC3}
     - SUB { OUTPUT2 }, { OUTPUT } { SRC3 }
HERE'S WHAT IT REALLY LOOKS LIKE
A THOUGHT EXPERIMENT

CONVERT A VON NEUMANN BINARY INTO DATAFLOW (IGNORE MEMORY FOR NOW)
BENEFITS

- HIGHLY PARALLEL BY ITS NATURE
- NOT CONSTRAINED BY ARTIFICIAL DEPENDENCIES
- ELEGANT
  - AS ELEGANT AS VON NEUMANN, BUT IN THE OTHER EXTREME
MAJOR DATAFLOW MODELS

- STATIC
  - EXTENSION: FIFO-STATIC
- DYNAMIC - ALSO CALLED TAGGED-TOKEN
- DEMAND-DRIVEN
  - THE MOST BIZARRE / LEAST IMPORTANT
STATIC DATAFLOW

**TOKEN-STORE**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Tokens</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>{1.R,2.L}</td>
<td>#1</td>
<td>#2</td>
</tr>
<tr>
<td>MUL</td>
<td>{2.R}</td>
<td>#10</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>{ }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIFO/STATIC DATAFLOW

TOKEN-STORE

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>{1.R, 2.L}</td>
<td>{#1, ...}</td>
<td>{#2, ...}</td>
</tr>
<tr>
<td>MUL</td>
<td>{2.R}</td>
<td>{#10, ...}</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>{}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DYNAMIC DATAFLOW

<table>
<thead>
<tr>
<th>tag</th>
<th>Operation</th>
<th>Tokens</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag1</td>
<td>ADD</td>
<td>{1.R,2.L}</td>
<td>#1</td>
<td>#2</td>
</tr>
<tr>
<td>tag1</td>
<td>MUL</td>
<td>{2.R}</td>
<td>#10</td>
<td></td>
</tr>
<tr>
<td>tag1</td>
<td>SUB</td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag2</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag3</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TOKEN-STORE

- #2
- #1
- #10
DEMAND-DRIVEN DATAFLOW

+  #1
#2

*  
#10

-  
#10
#1
#2

TOKEN-STORE

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>{1.R,2.L}</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>{2.R}</td>
<td></td>
</tr>
<tr>
<td>#10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
COMPLICATION #1: MESSYNESS

- How does + figure out that the input arrives from #1 sometimes and from - other times?

- And does this loop ever stop?
COMPLICATION #1: MESSYNESS

- Some dataflow models require a merge, others its implicit.
- All dataflow models (except the ridiculous) need some control mechanisms.
COMPLICATION #2: TIME

WHAT IF ITERATION 1 IS SLOW ABOUT SENDING THE + TO THE / RESULT, AND ITERATION 2 IS FAST?
COMPLICATION #2: TIME

SOLUTIONS OPEN UP PANDORAS BOX:

- ORDERED NETWORKS (STATIC DATAFLOW)
- NAMED VALUES (TAGGED TOKEN DATAFLOW)
COMPLICATION #3: MEMORY

PROGRAMMERS LIKE THEIR SIDE-EFFECTS
COMPLICATION #3: MEMORY

PAST SOLUTIONS:

- WRITE-ONCE MEMORY (VALUE-ORIENTATED LANGUAGES)
- I-STRUCTURES
- READ-LOCK MEMORY
- FULL/EMPTY BITS
- M-STRUCTURES
CANONICAL DATAFLOW MACHINE
IMPLEMENTATION PROBLEM #1: THE MEMORY WALL

TOKEN STORE

PE0
PE1
PE2
PE3

M-STRUCTURES
IMPLEMENTATION PROBLEM #1: THE MEMORY WALL

TOKEN STORE

M-STRUCTURES

PE0

PE1

PE2

PE3
IMPLEMENTATION PROBLEM #1: THE MEMORY WALL

TOKEN STORE

PE0

PE1

PE2

PE3

M-STRUCTURES
IMPLEMENTATION PROBLEM #1: THE SCHEDULING PROBLEM

- Do I iterate the loop or follow the / path?
- How do I know which computation is on the critical path?
IMPLEMENTATION PROBLEM #2: THE SCHEDULING PROBLEM

TOKEN STORE

M-STRUCTURES

FINITE PE RESOURCES

PE0

PE1

PE2

PE3

POLLUTED TOKEN-STORE
THE CENTRAL PROBLEM, IMHO
M/I-STRUCTURE HUH???

 HISTORY HAS SHOWN THAT PROGRAMMERS LIKE IMPERATIVE LANGUAGES

 (TAKE HEED CMP!)

 DATAFLOW MACHINES HAD NO MIGRATION PATH FOR CODE
VON NEUMANN EXAMPLE

\[ A[j + i*I] = i; \]

MUL \( T1 \leftarrow I, J \)
MUL \( T2 \leftarrow I, I \)
ADD \( T3 \leftarrow A, T1 \)
ADD \( T4 \leftarrow J, T2 \)
ADD \( T5 \leftarrow A, T4 \)
STORE \( (T5) \leftarrow I \)
LOAD \( B \leftarrow (T3) \)

\( B = A[i*J]; \)
\[ A[I] + I*J = I; \]

\[ B = A[I*J]; \]
\[ A[j] + i*i = i; \]

\[ B = A[i*j]; \]
DATAFLOW EXAMPLE

\[ A[j] + I*J = I; \]

\[ B = A[I*J]; \]
DATAFLOW EXAMPLE

\[ \text{A}_{IJ} + I \times I = I; \]

\[ B = \text{A}_{I \times J}; \]
DATAFLOW EXAMPLE

\[ A[I] + I*I = I; \]

\[ B = A[I*I]; \]
DATAFLOW EXAMPLE

\[ A[I] + I*I[I] = I; \]

\[ B = A[I*I[I]]; \]
DATAFLOW’S ACHILLES’ HEEL

- NO ORDERING FOR MEMORY OPERATIONS
- NO IMPERATIVE LANGUAGES (C, C++, JAVA)
- DESIGNERS RELIED ON FUNCTIONAL LANGUAGES INSTEAD

TO BE USEFUL, WAVESCALAR MUST SOLVE THE DATAFLOW MEMORY ORDERING PROBLEM
WAVESCALAR’S SOLUTION

ORDER MEMORY OPERATIONS

JUST ENOUGH ORDERING

PRESERVE PARALLELISM

A → J → I

* → + → LOAD → B

* → + → STORE
WAVE-ORDERED MEMORY

- Compiler annotates memory operations
  - Sequence #
  - Successor
  - Predecessor

- Send memory requests in any order

Diagram:
- Load 2 3 4
- Store 3 4 ?
- Load 4 7 8
- Store ? 8 9
WAVE-ORDERING EXAMPLE

LOAD 2 3 4
STORE 3 4 ?
STORE 4 5 6
LOAD 5 6 8
STORE ? 8 9
LOAD 4 7 8
STORE 3 4 ?
STORE 4 7 8
STORE 2 3 4

STORE BUFFER
WAVE-ORDERED MEMORY

- Waves are loop-free sections of the control flow graph.
- Each dynamic wave has a wave number.
- Each value carries its wave number.
- Total ordering.
- Ordering between waves.
WAVE-ORDERED MEMORY

- Annotations summarize the CFG
- Expressing parallelism
  - Reorder consecutive operations
- Alternative solution: Token passing [Beck, JPDC'91]
  - 1/2 the parallelism
WHAT HAPPENED TO DATAFLOW?

- LESSON FROM THE PAST: BACKWARD COMPATIBILITY MATTERS. BIZARRE PROGRAMMING LANGUAGES WONT WORK

- LESSON FROM WAVESCALAR: ONCE YOU SOLVE ONE PROBLEM (FALSE CONTROL DEPENDENCIES) ANOTHER APPEARS (INHERENT SERIALIZATION IN THE MEMORY INTERFACE)

- LOOKING FORWARD: SPECULATIVE DATAFLOW
A VIEW ON THE PAST

INSTRUCTIONS ➔ FULLY OUT-OF-ORDER EXECUTION ➔ MEMORY OPERATIONS