DATAFLOW COMPUTING

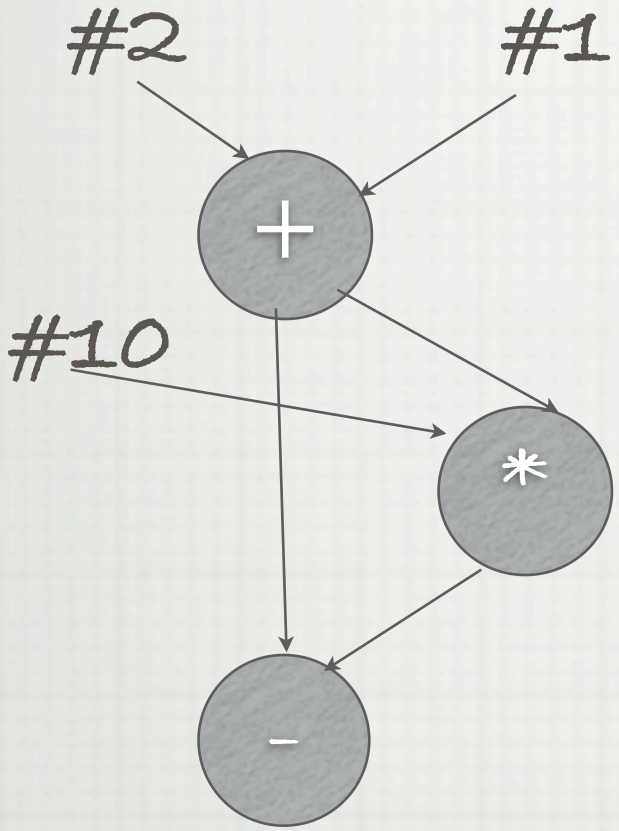
590A LECTURE 2

EXECUTION ALGORITHM

```
WHILE(AVAILABLE_OPERATIONS(STATE)) {
  STATE = EXEC(AVAILABLE-OPERATION(STATE),STATE)
  }
```

OPERATIONS "FIRE" WHEN ALL INPUTS ARE AVAIALBLE -- ALSO KNOWN AS THE DATAFLOW FIRING RULE

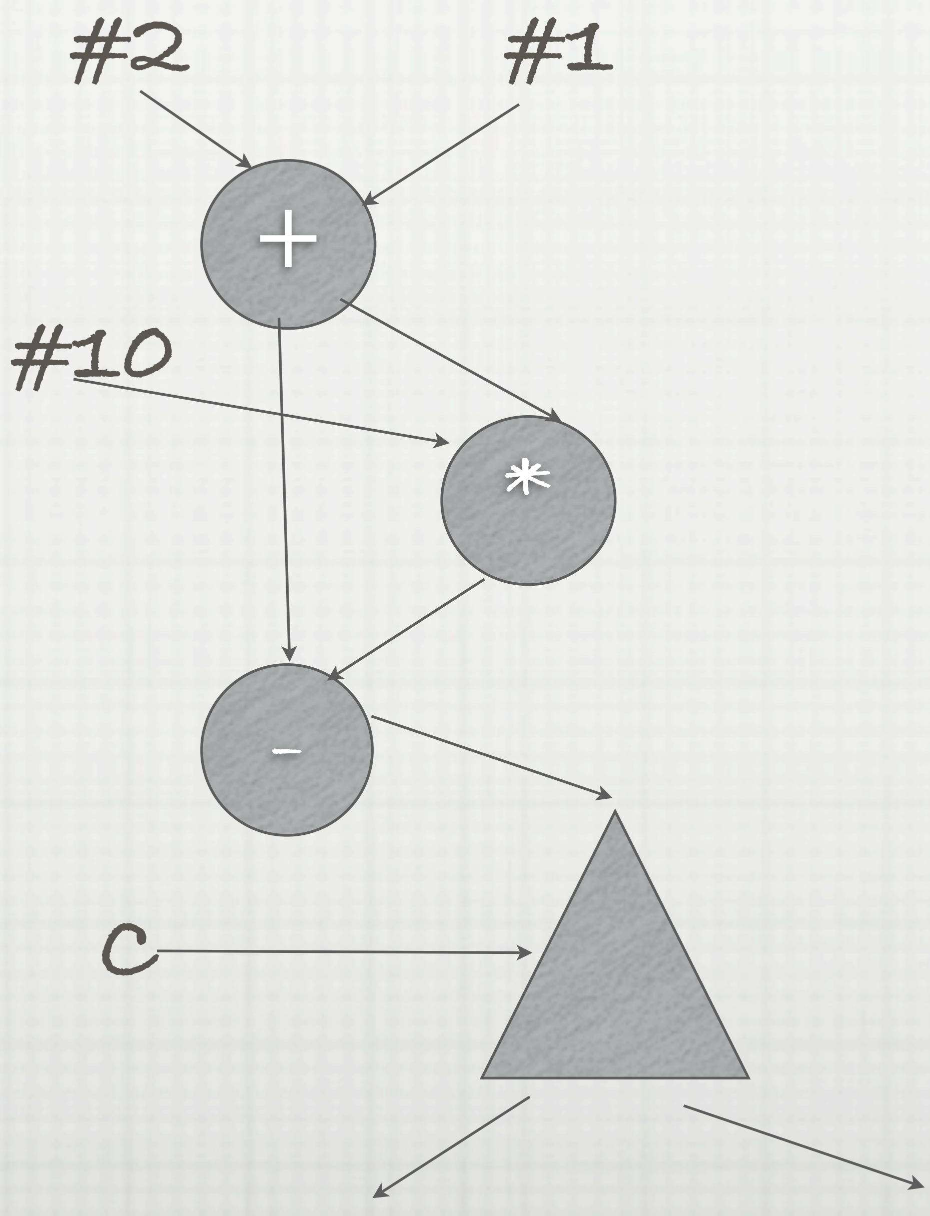
E.G. STATIC DATAFLOW



TOKEN-STORE

ADD	{1.R,2.L}	#1	#2
MUL	{2.R}	#10	
SUB	{ }		

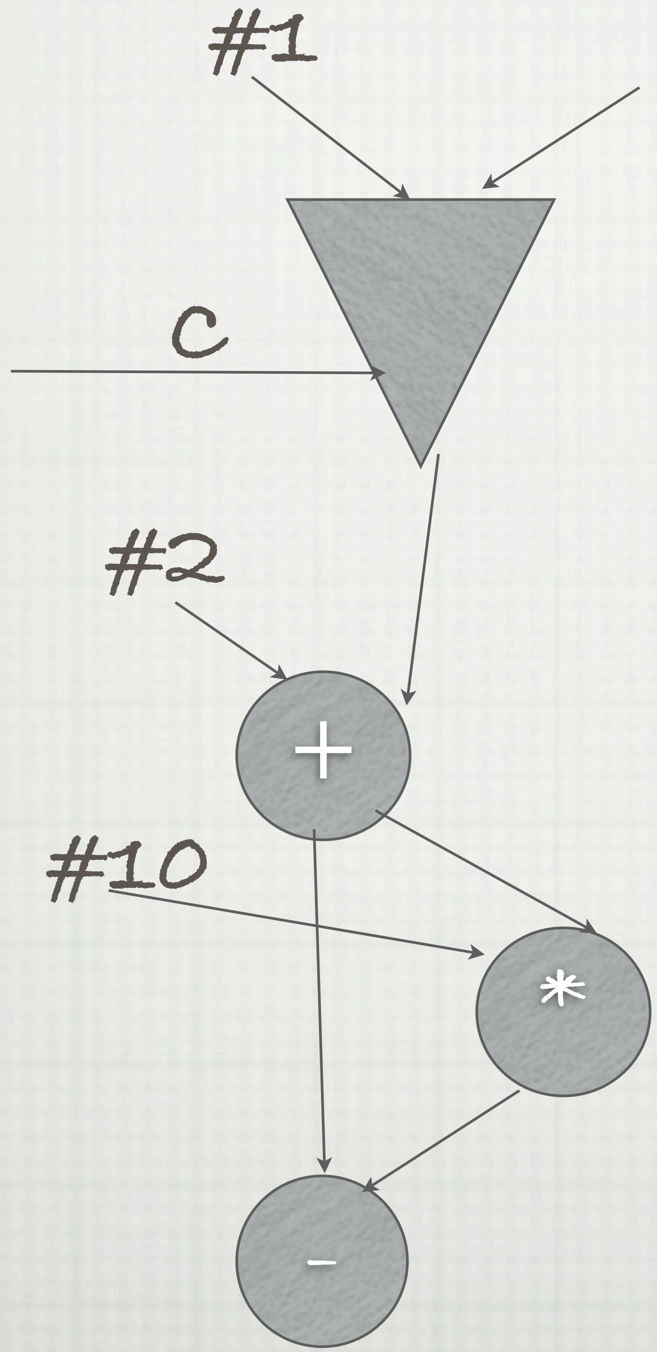
USE DATA FOR CONTROL



ALSO CALLED A STEER OR BRANCH

IF C == 0
SEND LEFT
ELSE
SEND RIGHT

USE DATA FOR CONTROL



IF C == 0

PASS LEFT

ELSE

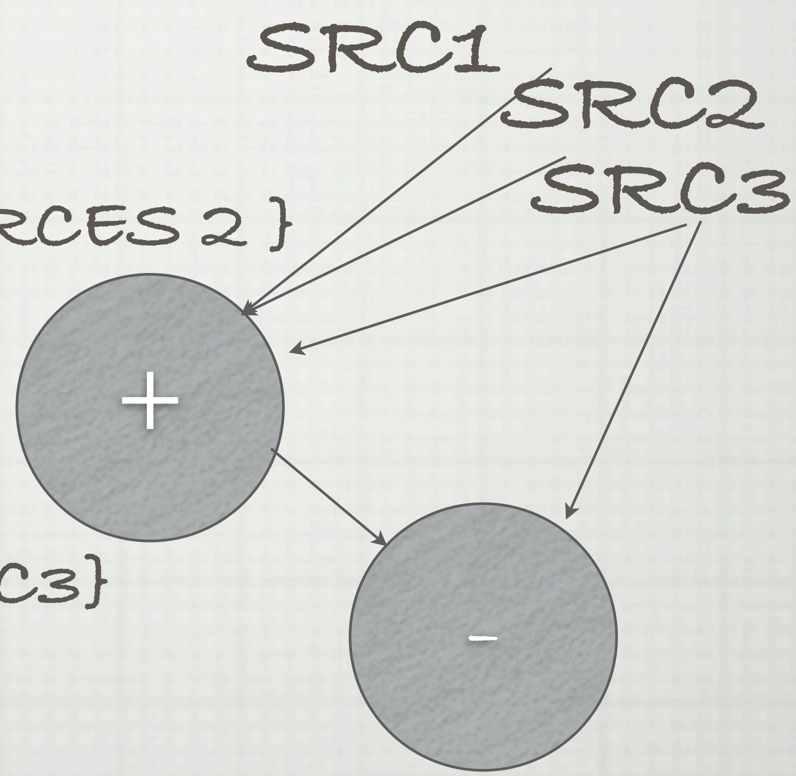
PASS RIGHT

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 }

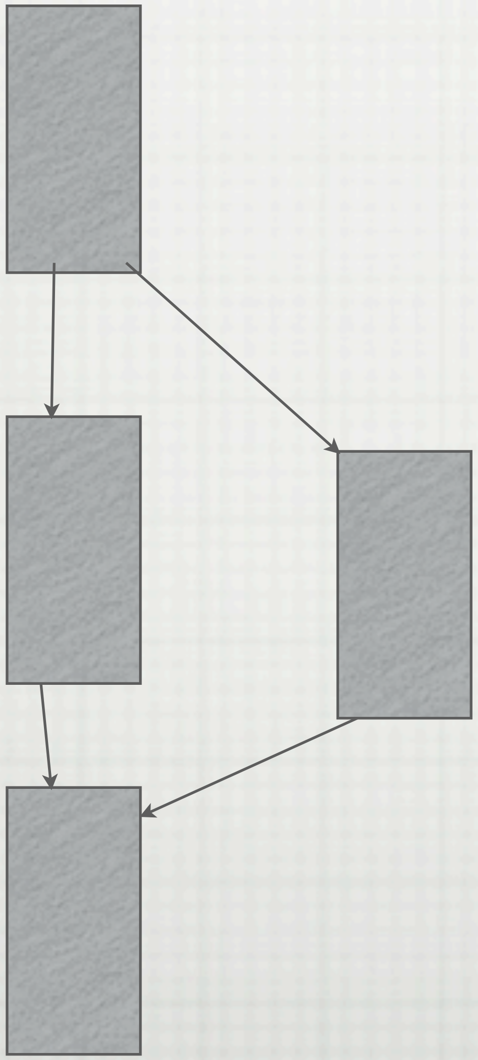


HERE'S WHAT IT REALLY LOOKS LIKE

```
        NOP CALCLOOP_SYNC_NEXT, CALCLOOP_SYNC_LOOP

        DTW RW_T_END, CALCLOOP_SYNC_FINI, RW_T
        DTW RT_T_END, CALCLOOP_SYNC_FINI, RT_T
        DTW C_T_END, CALCLOOP_SYNC_FINI, C_T

        WA I_F, I_END
WA RW_T_F, RW_T_END
WA RT_T_F, RT_T_END
WA C_T_F, C_T_END
        WA R_F, R_END
        STQ_U I_F_DONE, R_F, C_T_F, 0


        CNST RESULT, I_F_DONE, 1
        DTTW THREAD_RESULT, RT_T_F, RW_T_F, RESULT
        ADDQ F_LOOP_NEXT, F_LOOP_LOOP,THREAD_RESULT
        MULL NUM_LOOPS, SIZE_X_ARG_LOOP_LOOP, SIZE_Y_ARG_LOOP_LOOP
        CMPLT LOOPS_DONE, F_LOOP_NEXT, NUM_LOOPS
        RHO NOT_DONE, DONE, F_LOOP_NEXT, LOOPS_DONE
        ADDI T_LOOP_NEXT, T_LOOP_LOOP, 1
        ADDQ SW_LOOP_NEXT1, SW_LOOP_LOOP, SIZE_Y_ARG_LOOP_LOOP
        ADDQ SW_LOOP_NEXT, SW_LOOP_NEXT1, SIZE_Y_ARG_LOOP_LOOP
        ADDI X_NEXT, X_LOOP, 1
        CMPLT XLOOP_EQ, X_NEXT, SIZE_X_ARG_LOOP_LOOP
    NOP XLOOP_SYNC_NEXT, XLOOP_SYNC_LOOP
    DTW YLOOP_SYNC_NEXT, XLOOP_SYNC_FINI, YLOOP_SYNC_LOOP

    NOP SIZE_Y_ARG_NEXT, SIZE_Y_ARG_LOOP_END
NOP SIZE_X_ARG_NEXT, SIZE_X_ARG_LOOP_END
NOP A_ARG_NEXT, A_ARG_LOOP_END
NOP B_ARG_NEXT, B_ARG_LOOP_END
NOP C_ARG_NEXT, C_ARG_LOOP_END
NOP T_NEXT, T_LOOP_END
NOP F_NEXT, F_LOOP_END
NOP SW_NEXT, SW_LOOP_END
```

A THOUGHT EXPERIMENT

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



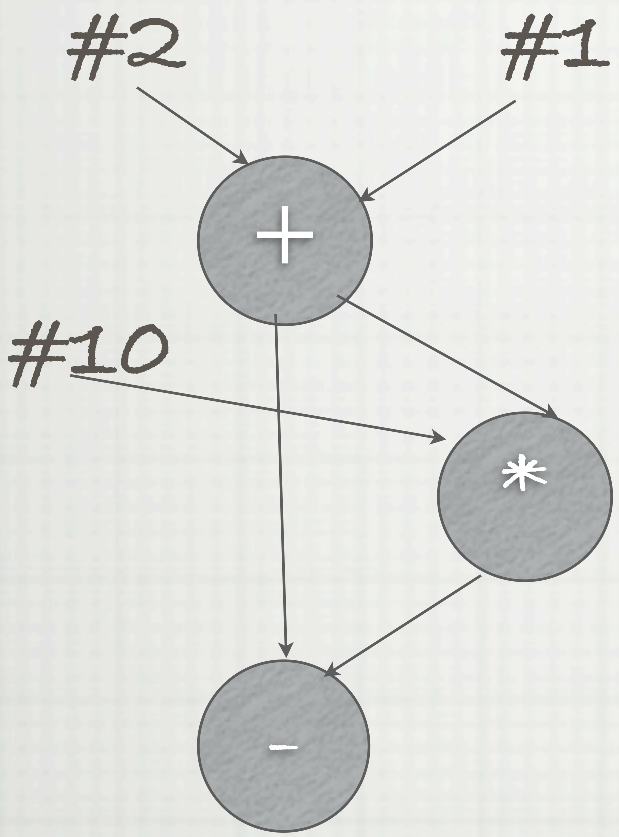
BENEFITS

- 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

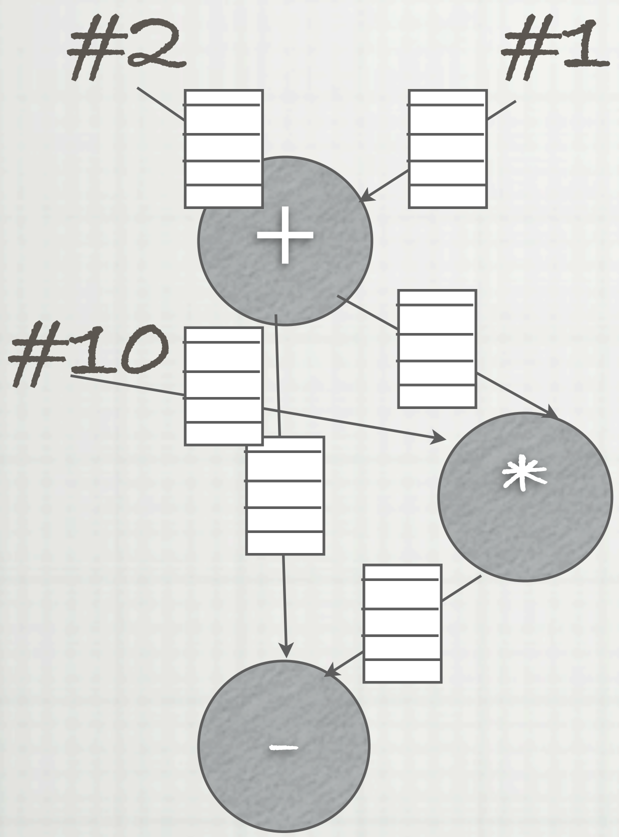
STATIC DATAFLOW



TOKEN-STORE

ADD	{1.R,2.L}	#1	#2
MUL	{2.R}	#10	
SUB	{ }		

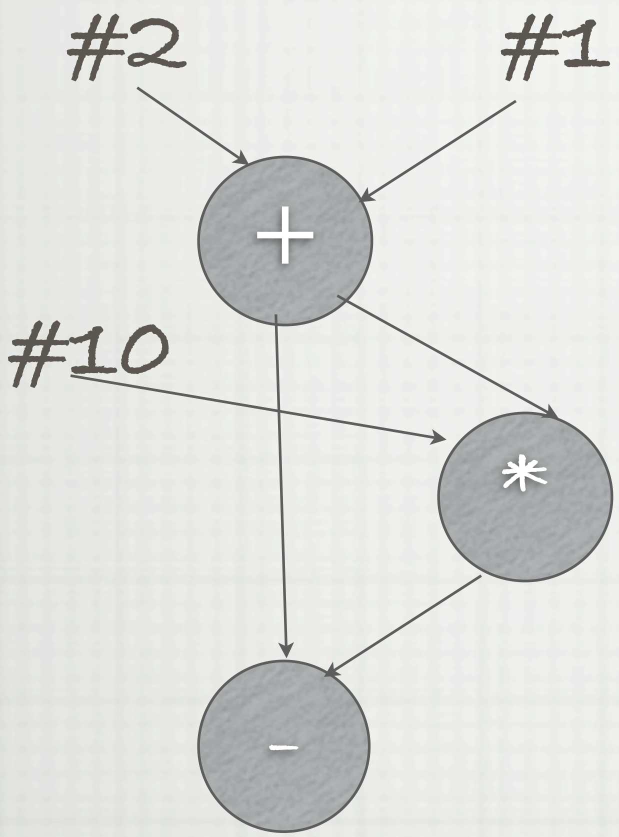
FIFO/STATIC DATAFLOW



TOKEN-STORE

ADD	{1.R,2.L}	{#1, ...}	{#2,...}
MUL	{2.R}	{#10, ...}	
SUB	{ }		

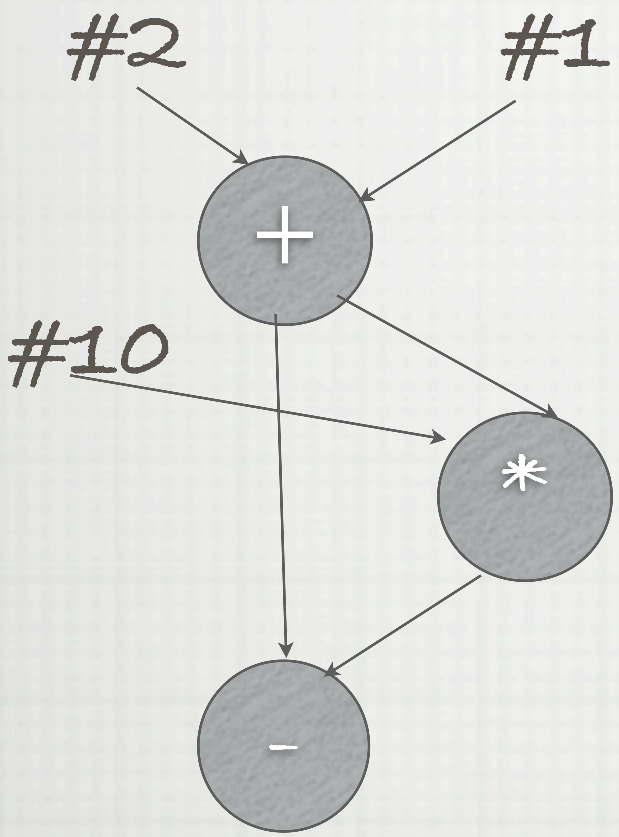
DYNAMIC DATAFLOW



TOKEN-STORE

tag1	ADD	{1.R,2.L}	#1	#2
tag1	MUL	{2.R}	#10	
tag1	SUB	{ }		
tag2	ADD			
tag3	ADD			

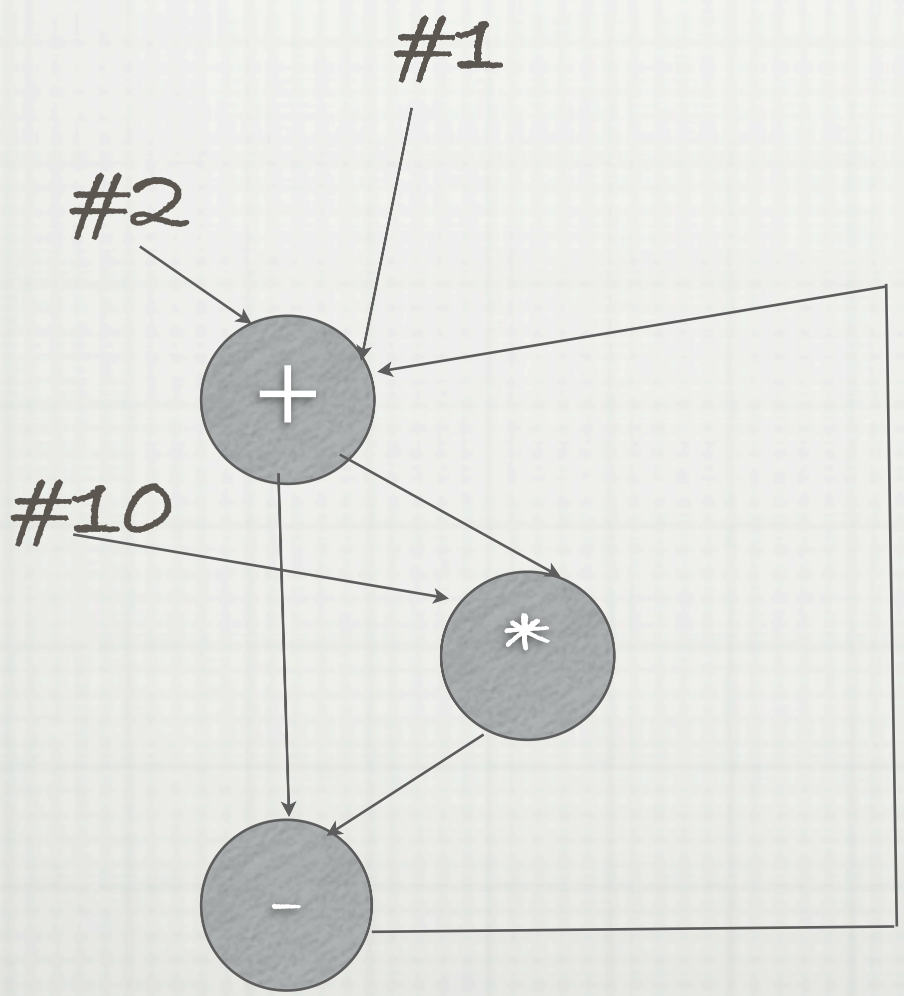
DEMAND-DRIVEN DATAFLOW



TOKEN-STORE

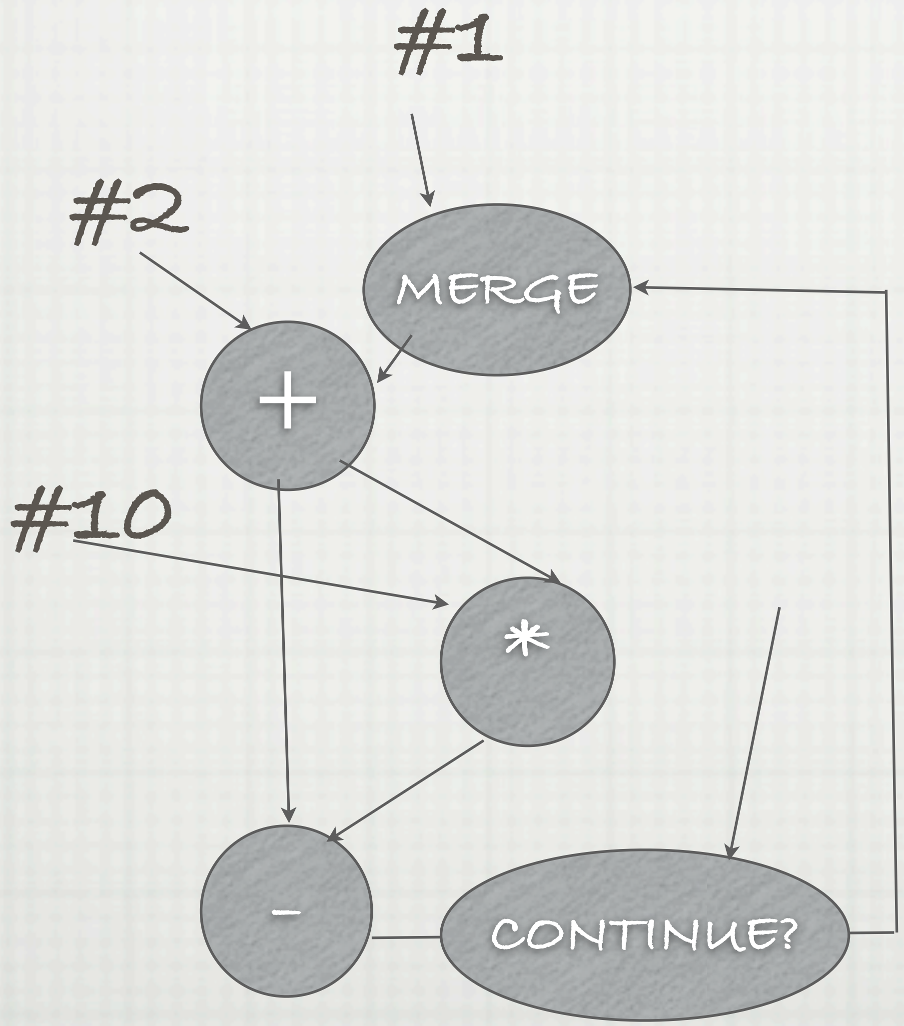
SUB	{ }		
ADD	{1.R,2.L}		
MUL	{2.R}		
#10			
#1			
#2			

COMPLICATION #1: MESSYNESS



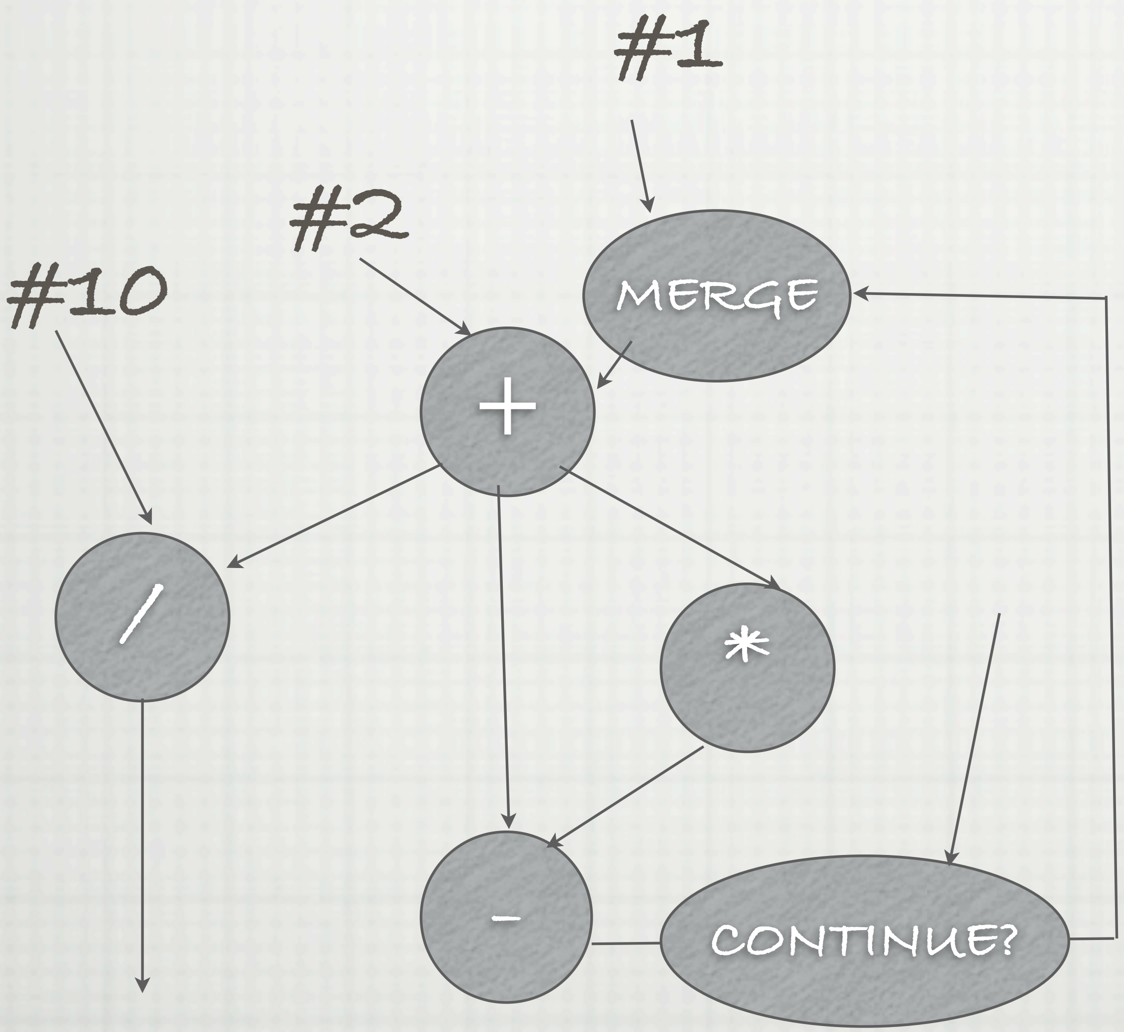
- HOW DOES + FIGURE OUT THAT THE INPUT ARRIVES FROM #1 SOMETIMES AND FROM - OTHER TIMES?
- AND DOES THIS LOOP EVER STOP?

COMPLICATION #1: MESSYNESS



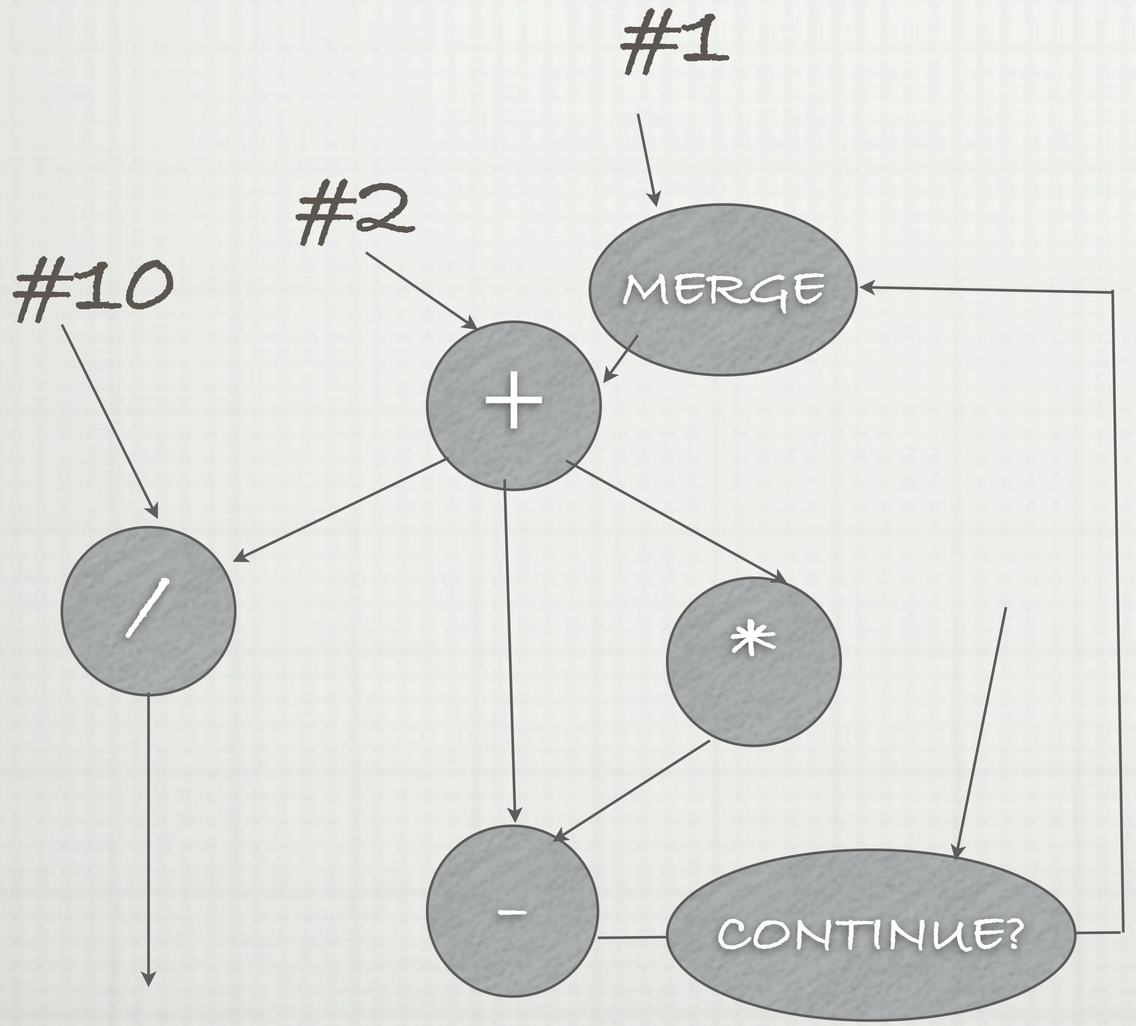
- SOME DATAFLOW MODELS REQUIRE A MERGE, OTHERS ITS IMPLICIT
- ALL DATAFLOW MODELS (EXCEPT THE RIDICULOUS) NEED SOME CONTROL MECHANISMS

COMPLICATION #2:TIME



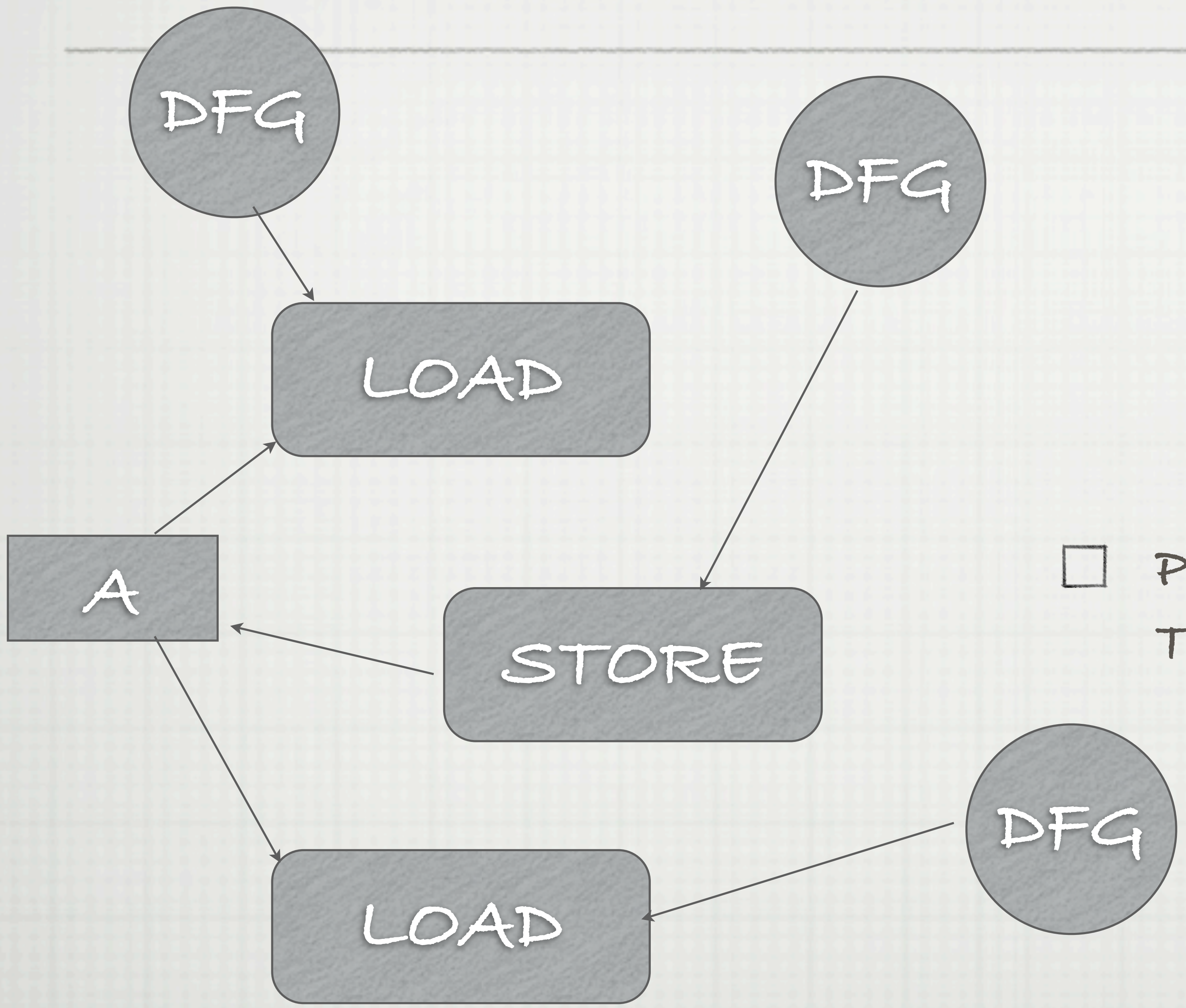
- WHAT IF ITERATION 1 IS SLOW ABOUT SENDING THE + TO THE / RESULT, AND ITERATION 2 IS FAST?

COMPLICATION #2:TIME



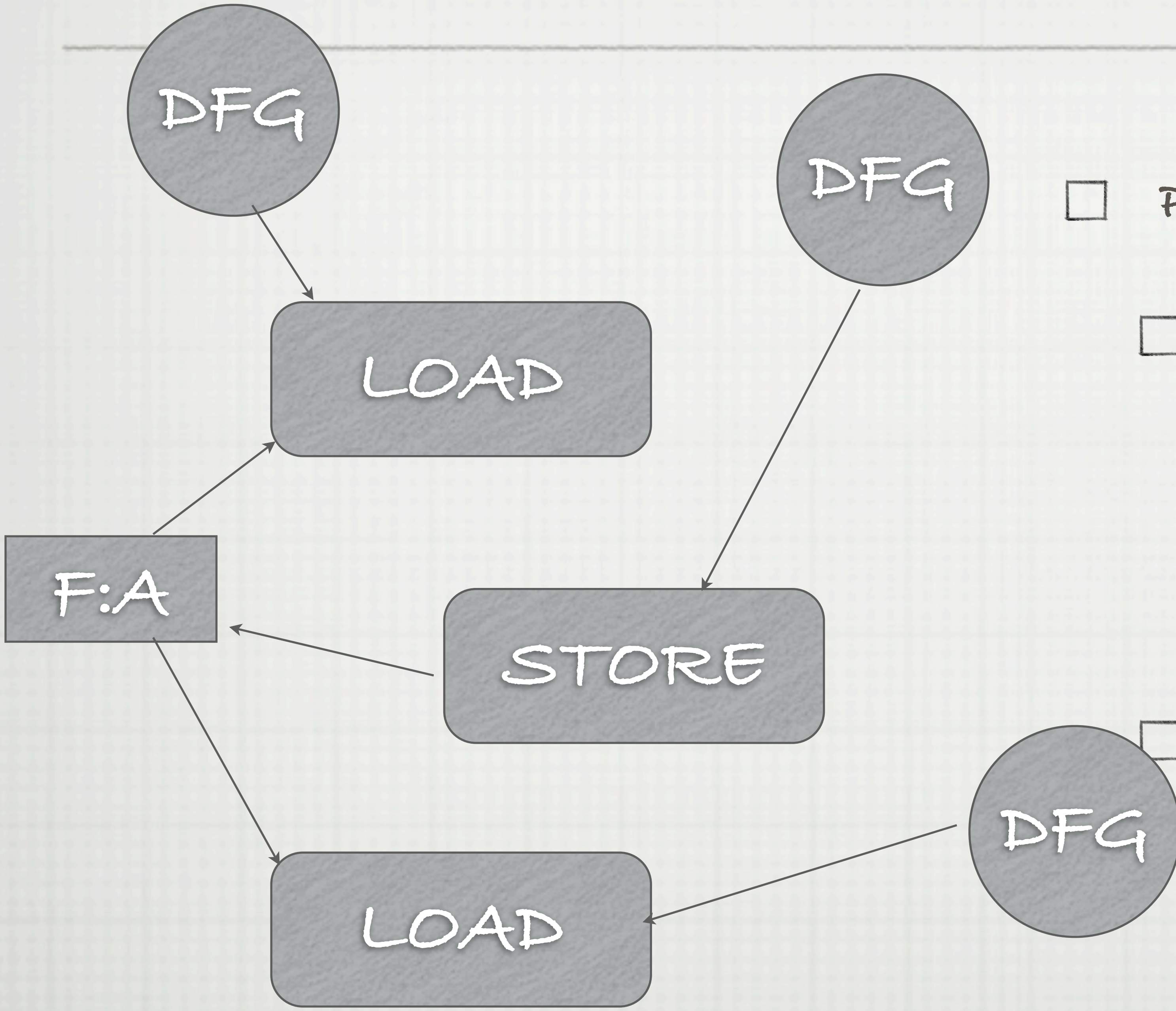
- SOLUTIONS OPEN UP PANDORAS BOX:
  - ORDERED NETWORKS (STATIC DATAFLOW)
  - NAMED VALUES (TAGGED TOKEN DATAFLOW)

COMPLICATION #3: MEMORY



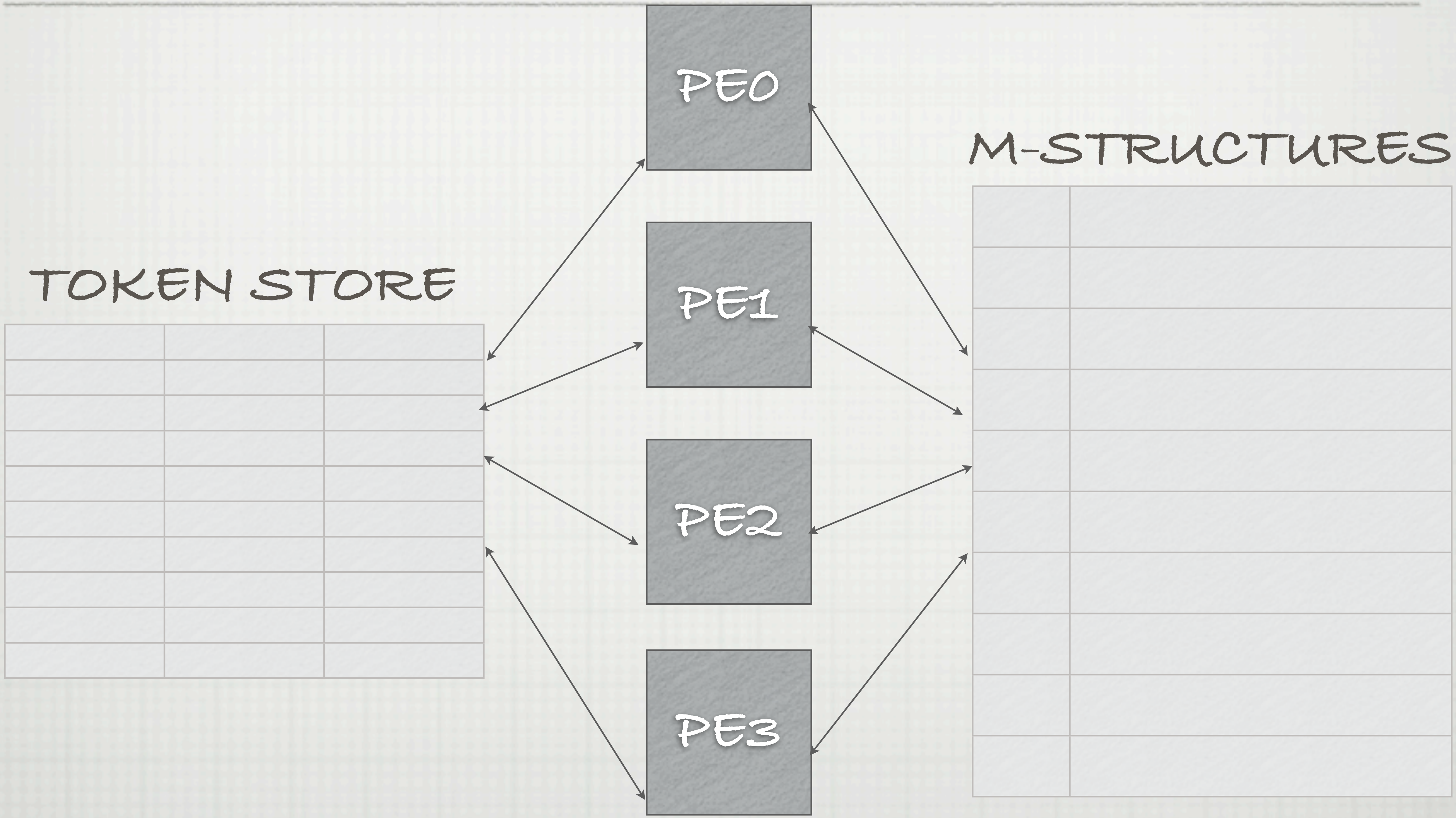
- PROGRAMMERS LIKE THEIR SIDE-EFFECTS

COMPLICATION #3: MEMORY

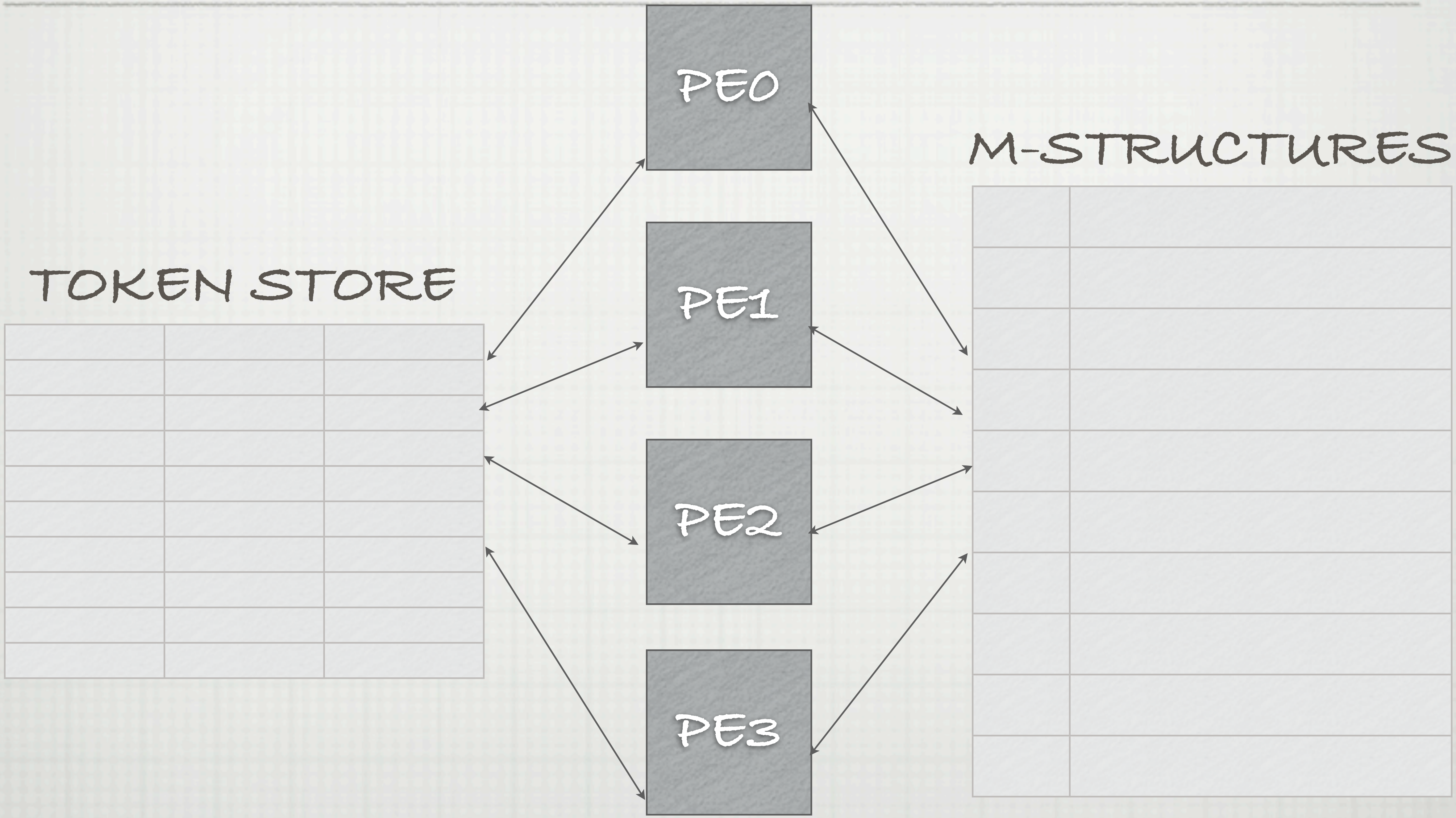


- PAST SOLUTIONS:
  - WRITE-ONCE MEMORY (VALUE-ORIENTATED LANGUAGES)
    - I-STRUCTURES
  - READ-LOCK MEMORY
    - FULL/EMPTY BITS
    - M-STRUCTURES

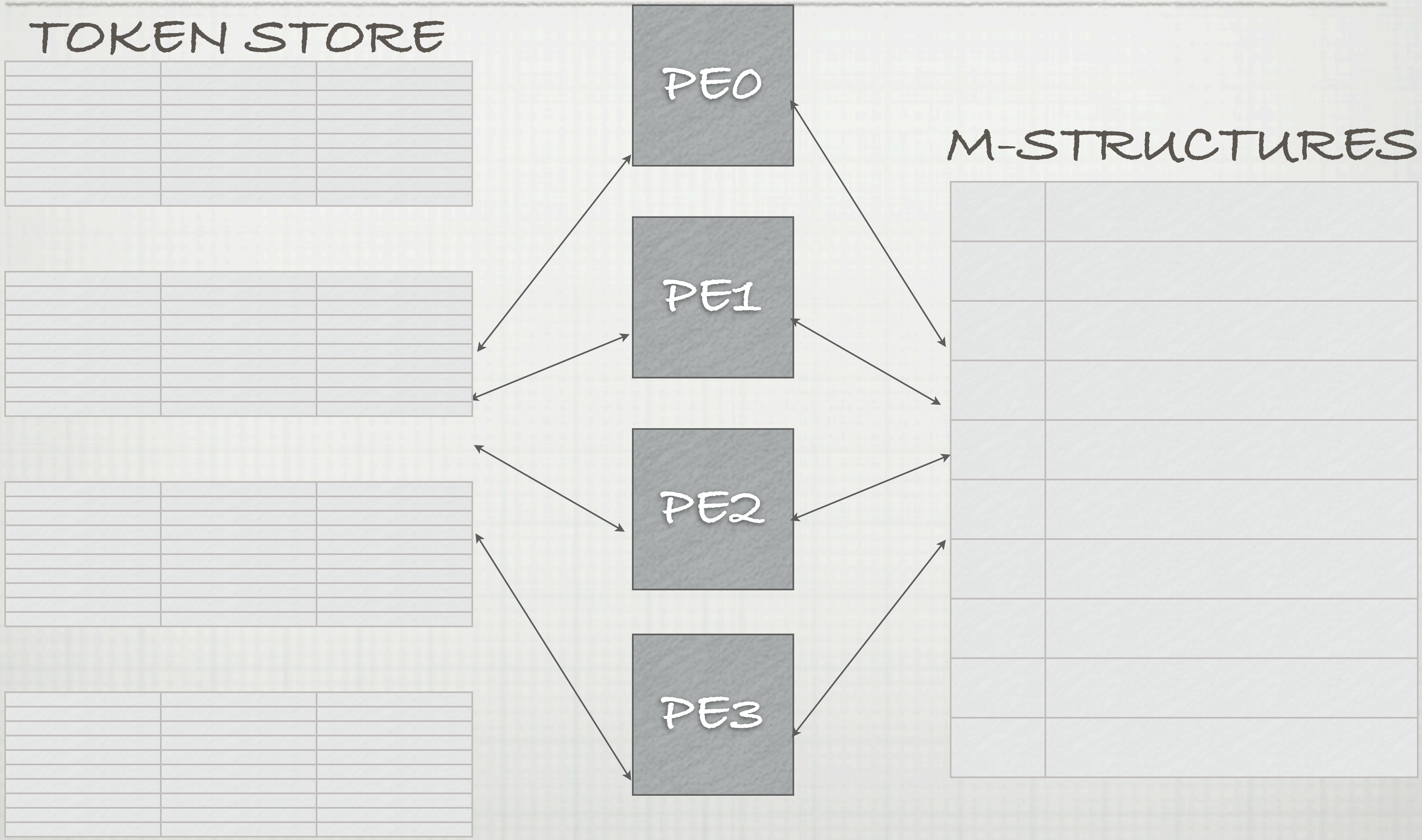
CANONICAL DATAFLOW MACHINE



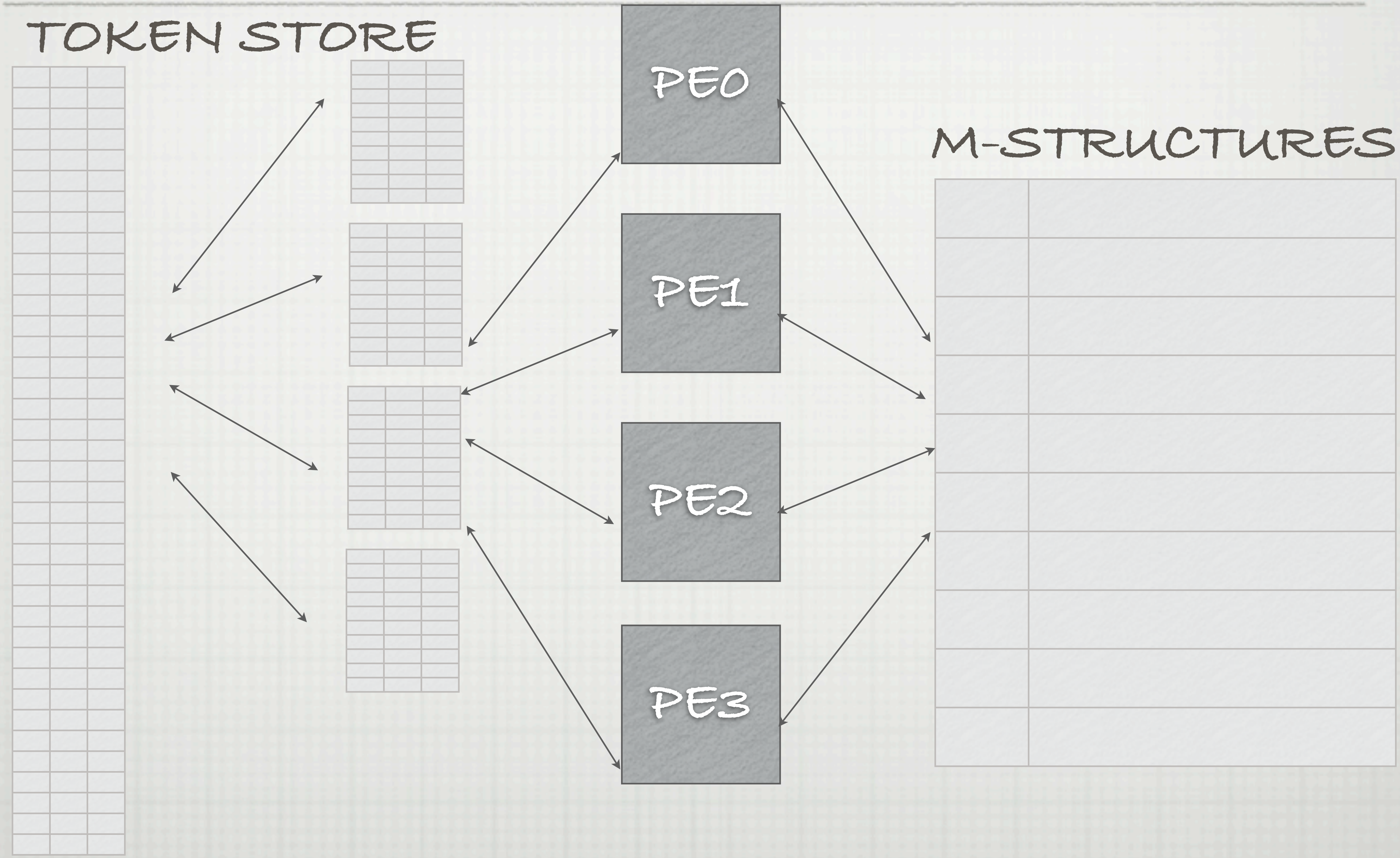
IMPLEMENTATION PROBLEM #1: THE MEMORY WALL



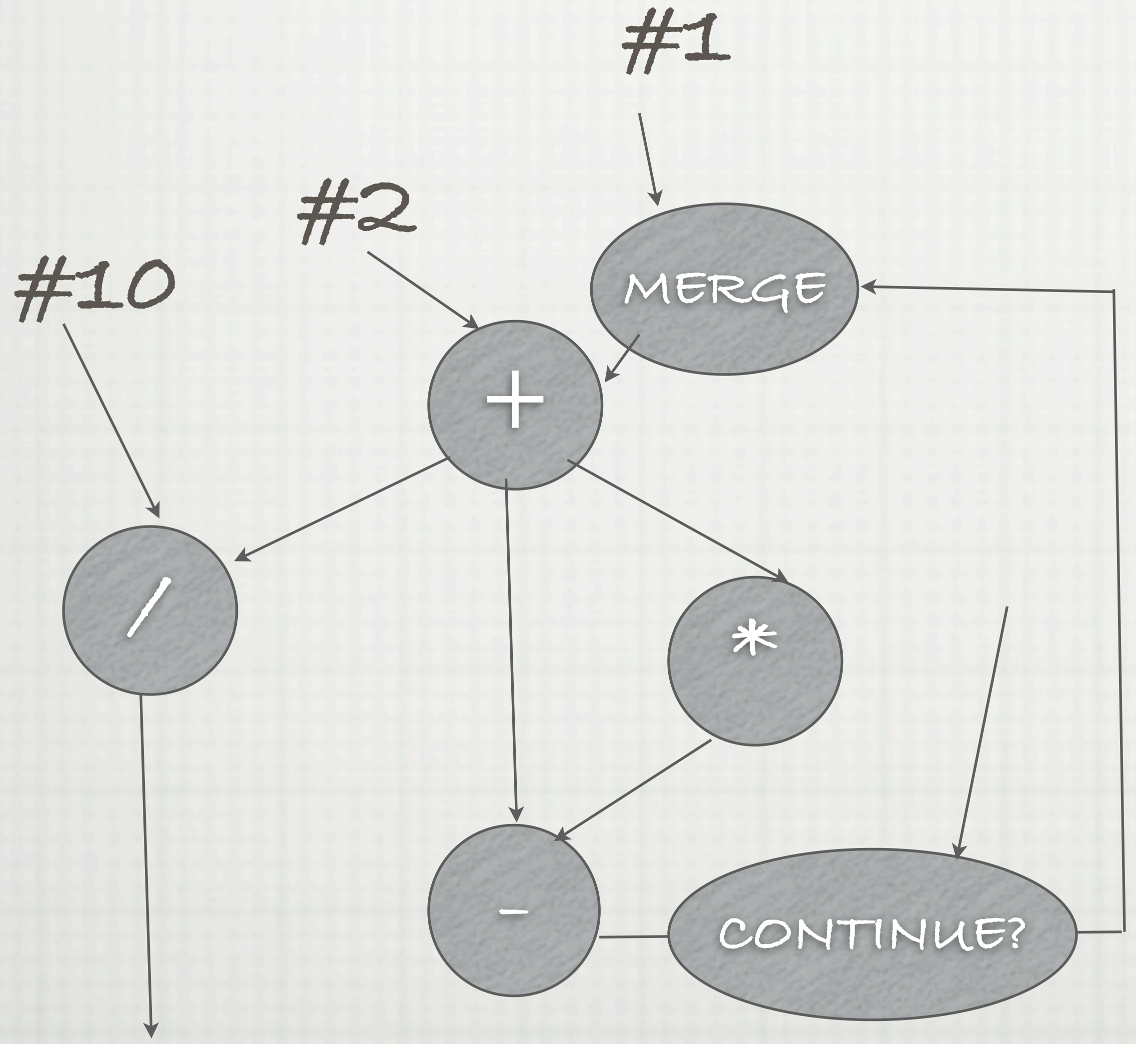
IMPLEMENTATION PROBLEM #1: THE MEMORY WALL



IMPLEMENTATION PROBLEM #1: THE MEMORY WALL

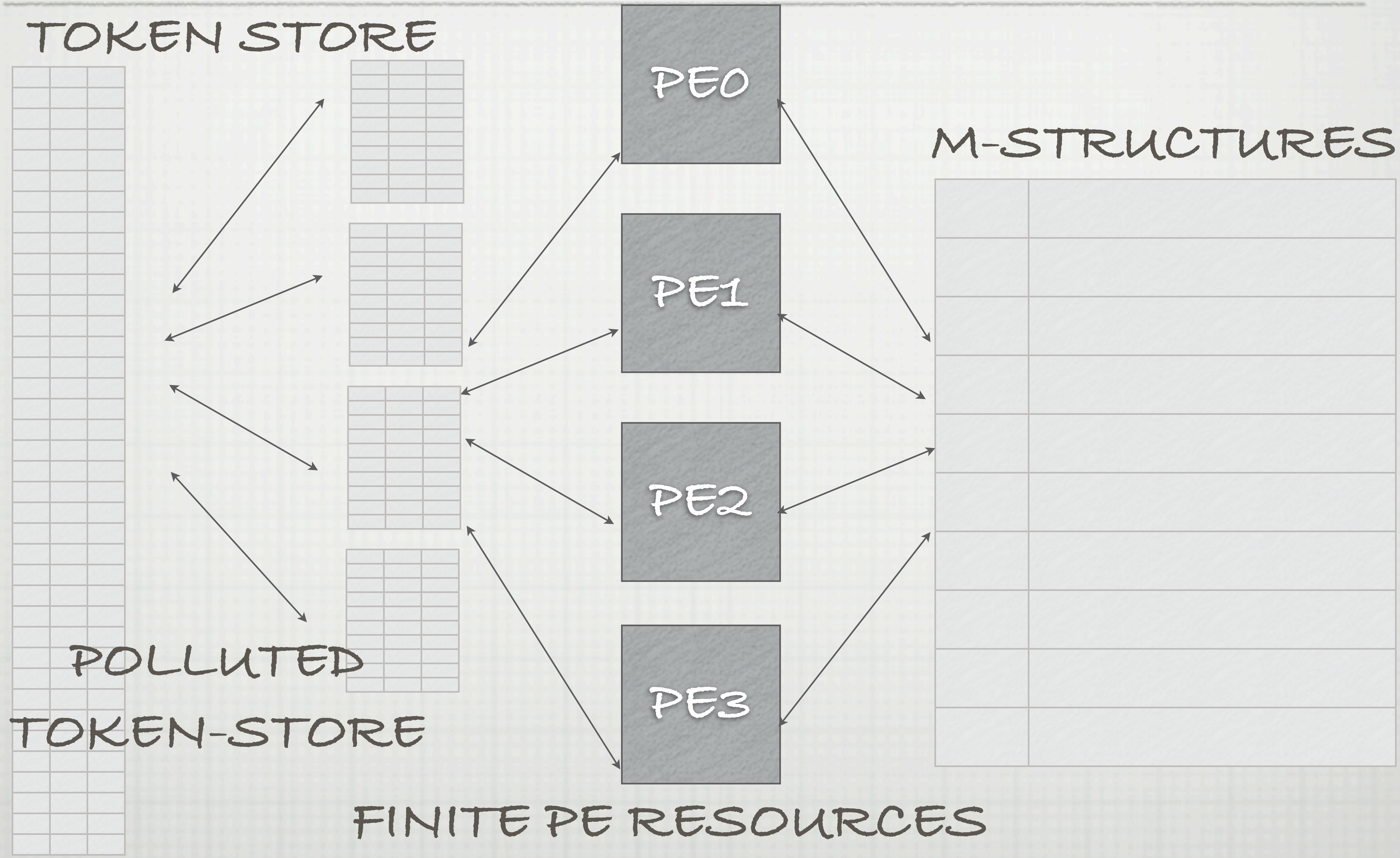


IMPLEMENTATION PROBLEM #1: THE THE SCHEDULING PROBLEM

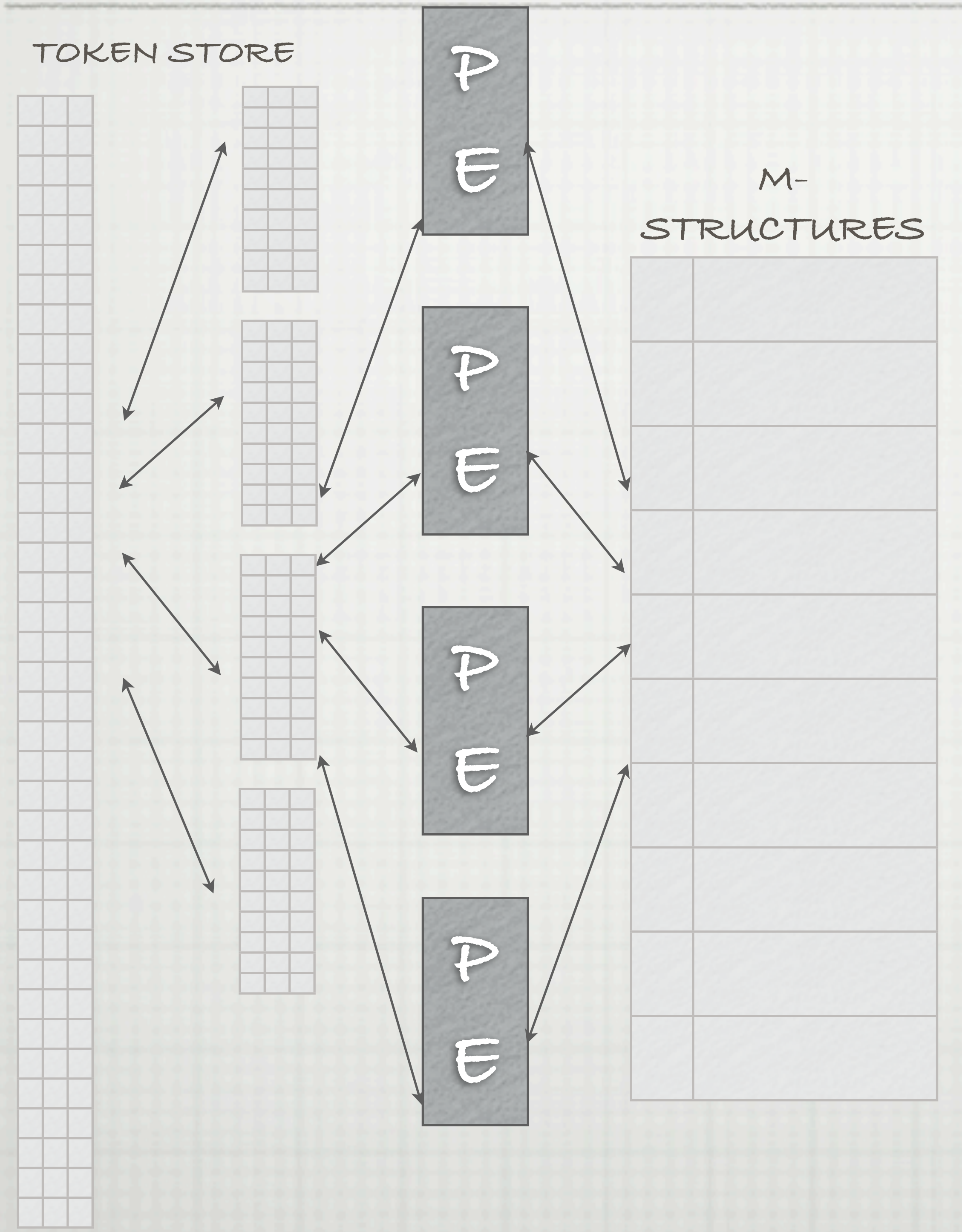


- DO I ITERATE THE LOOP OR FOLLOW THE / PATH?
- HOW DO I KNOW WHICH COMPUTATION IS ON THE CRITICAL PATH?

IMPLEMENTATION PROBLEM #2: THE SCHEDULING PROBLEM



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;

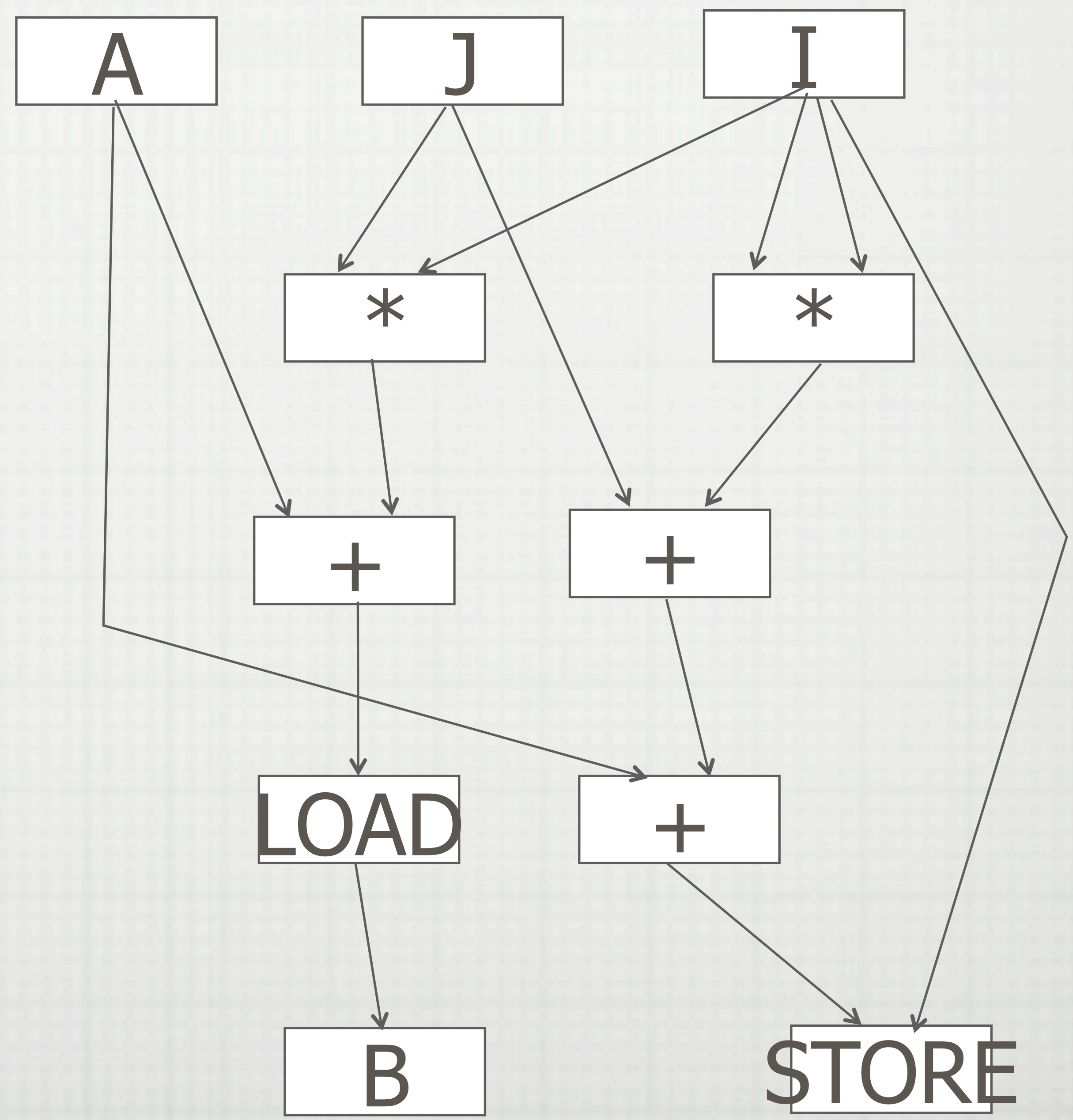
B = A[I*J];

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

DATAFLOW EXAMPLE

A[J + I*I] = I;

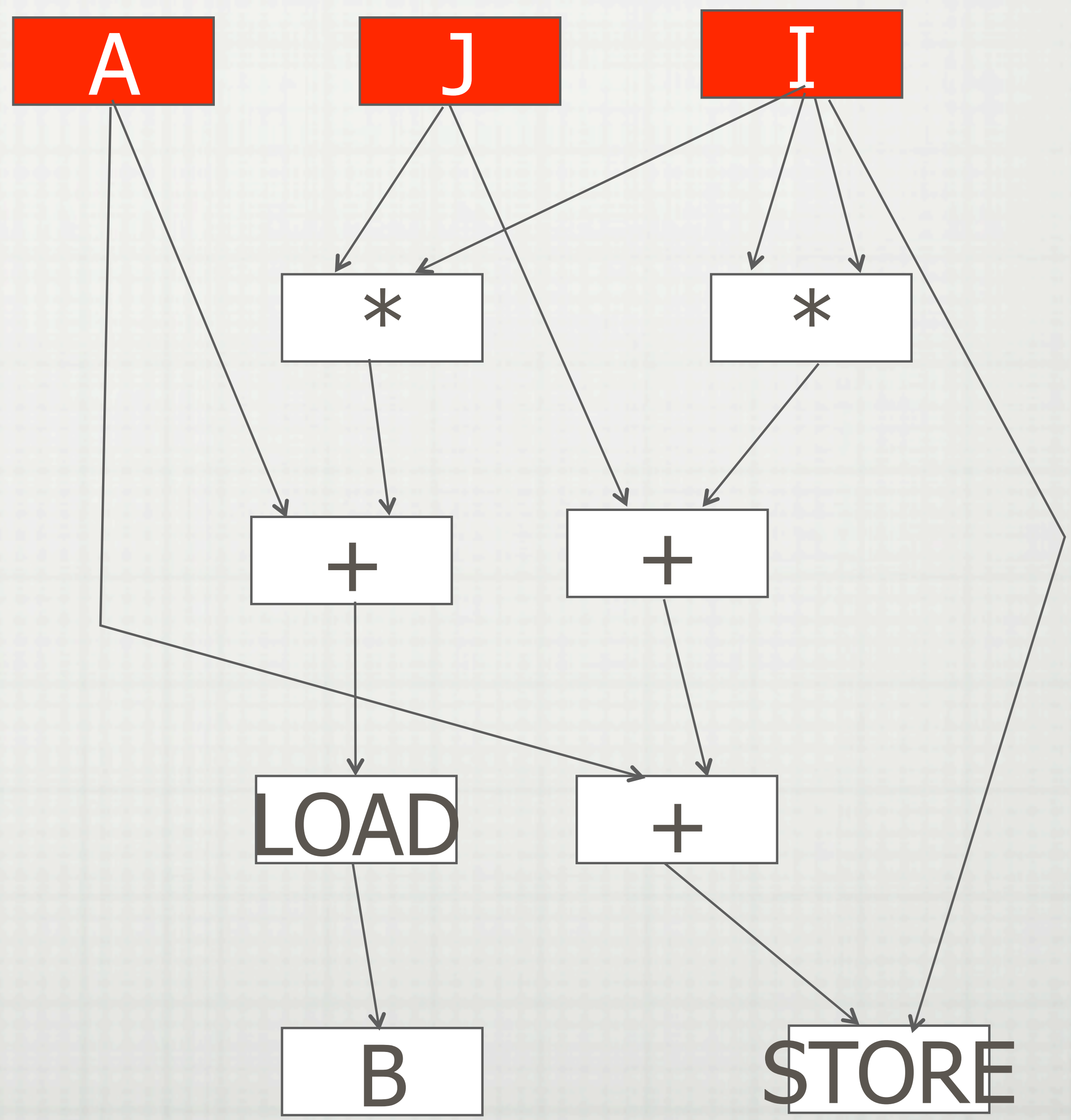
B = A[I*J];



DATAFLOW EXAMPLE

A[J + I*I] = I;

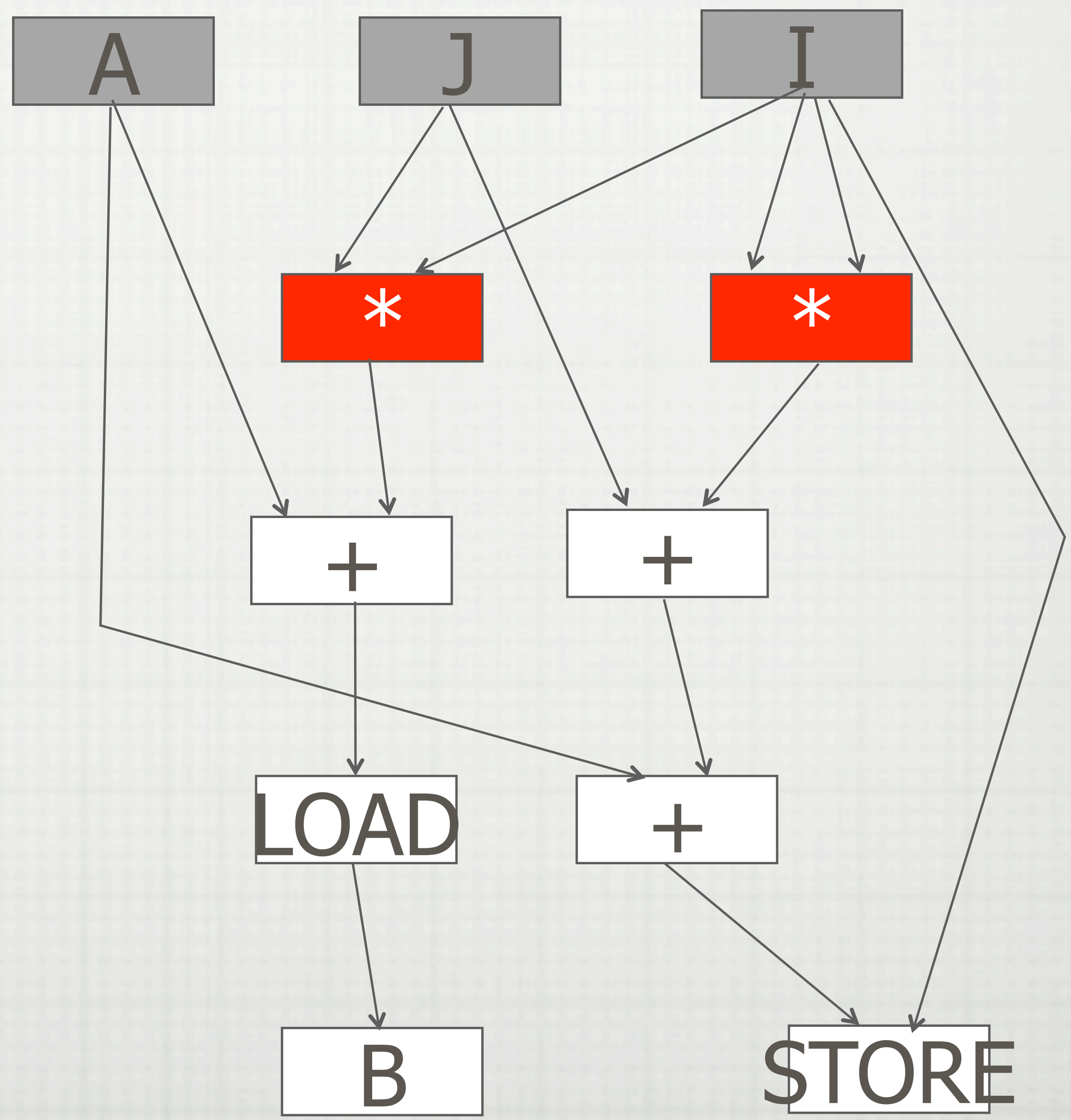
B = A[I*J];



DATAFLOW EXAMPLE

A[J + I*I] = I;

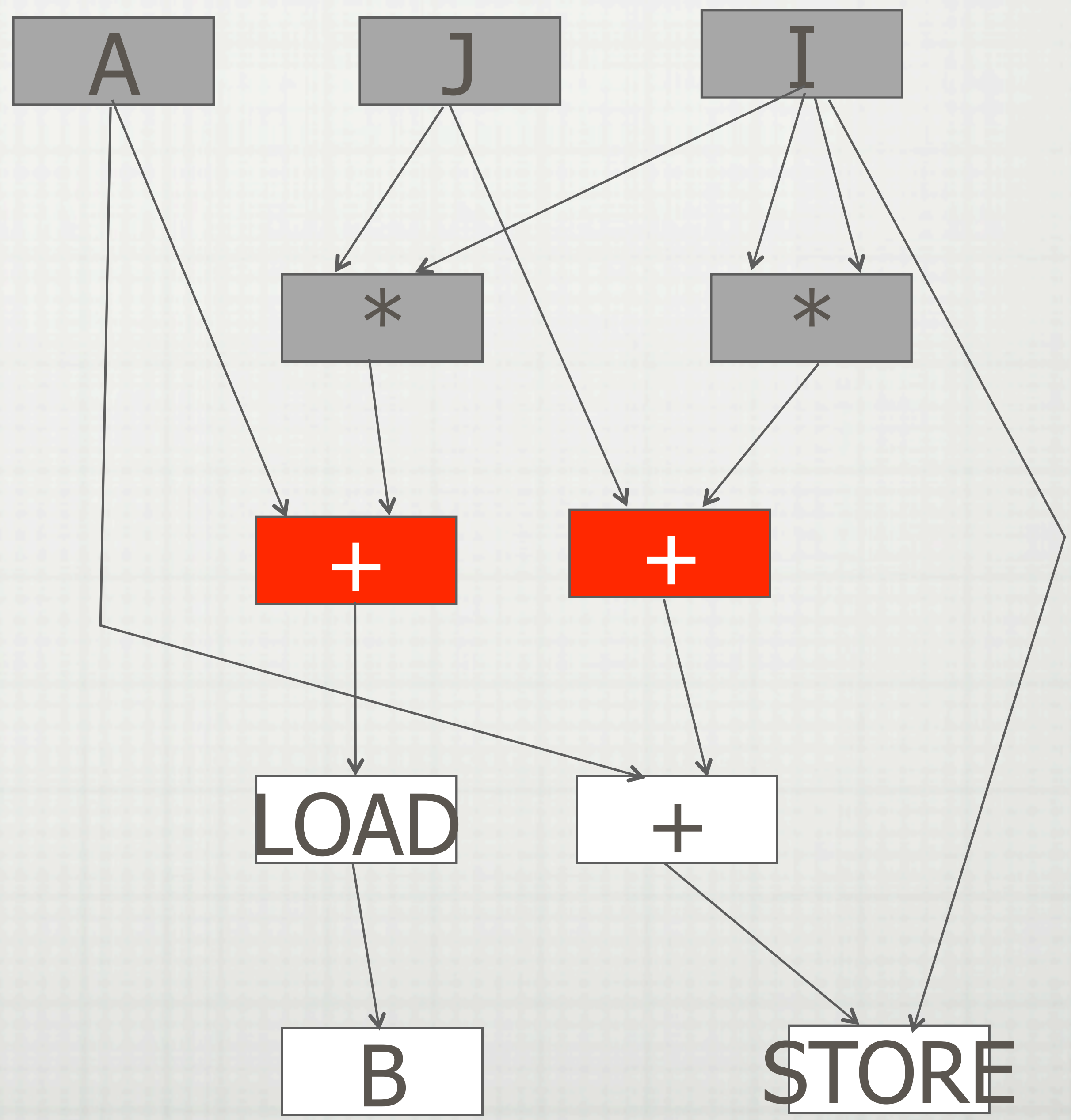
B = A[I*J];



DATAFLOW EXAMPLE

A[J + I*I] = I;

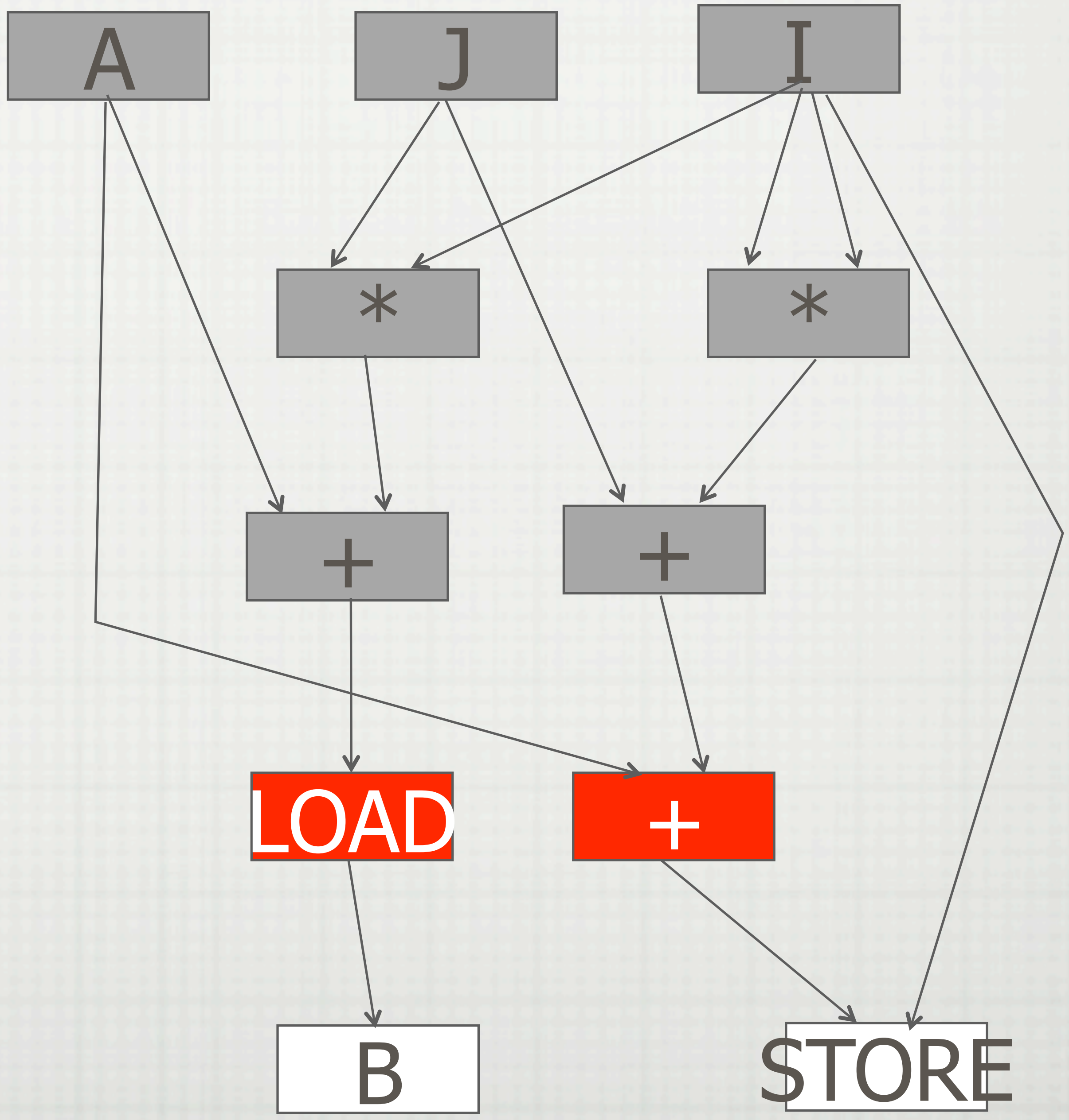
B = A[I*J];



DATAFLOW EXAMPLE

A[J + I*I] = I;

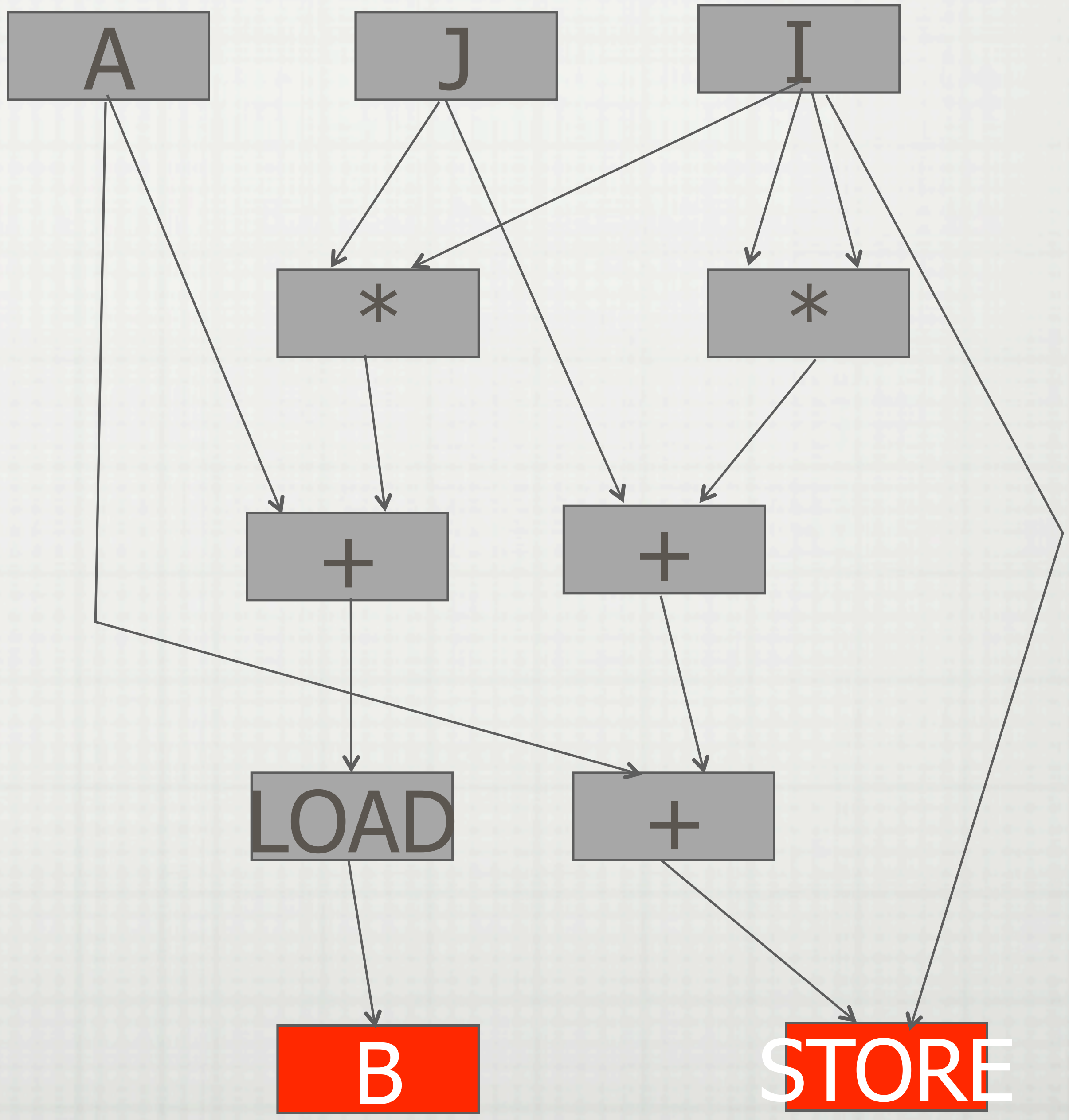
B = A[I*J];



DATAFLOW EXAMPLE

A[J + I*I] = I;

B = A[I*J];



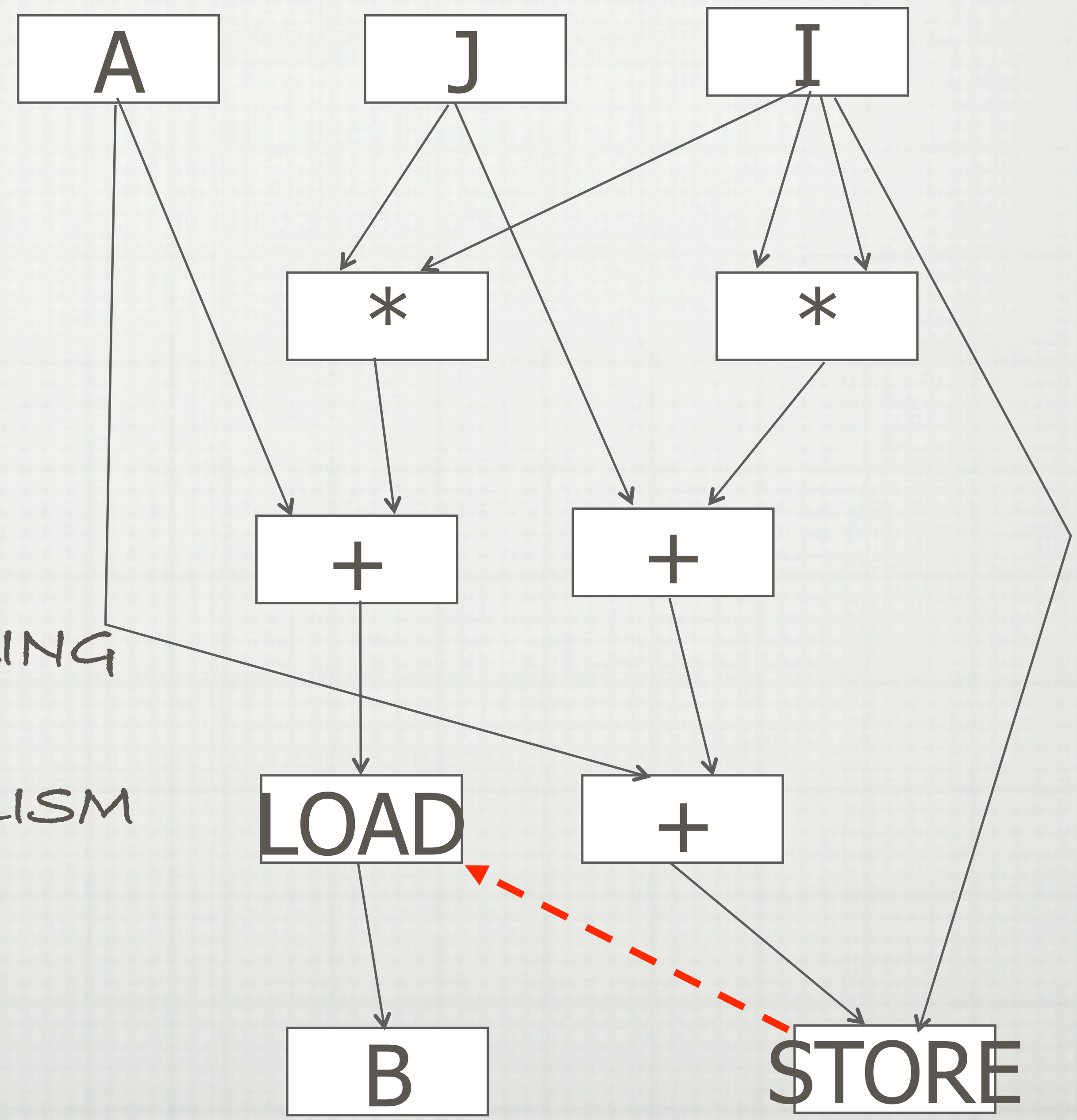
DATAFLOW'S ACHILLES' HEEL

- 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



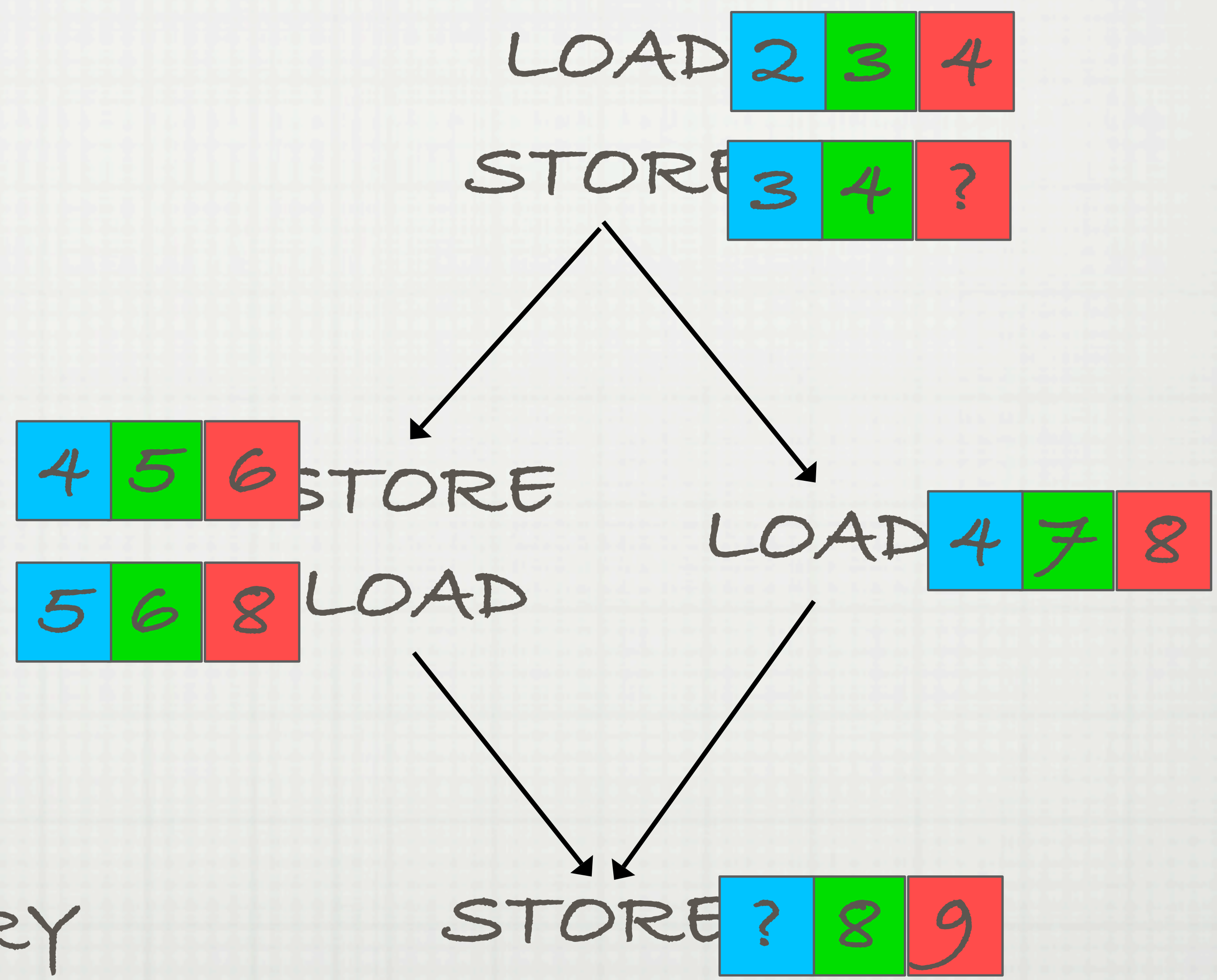
WAVE-ORDERED MEMORY

- COMPILER ANNOTATES MEMORY OPERATIONS

SEQUENCE #

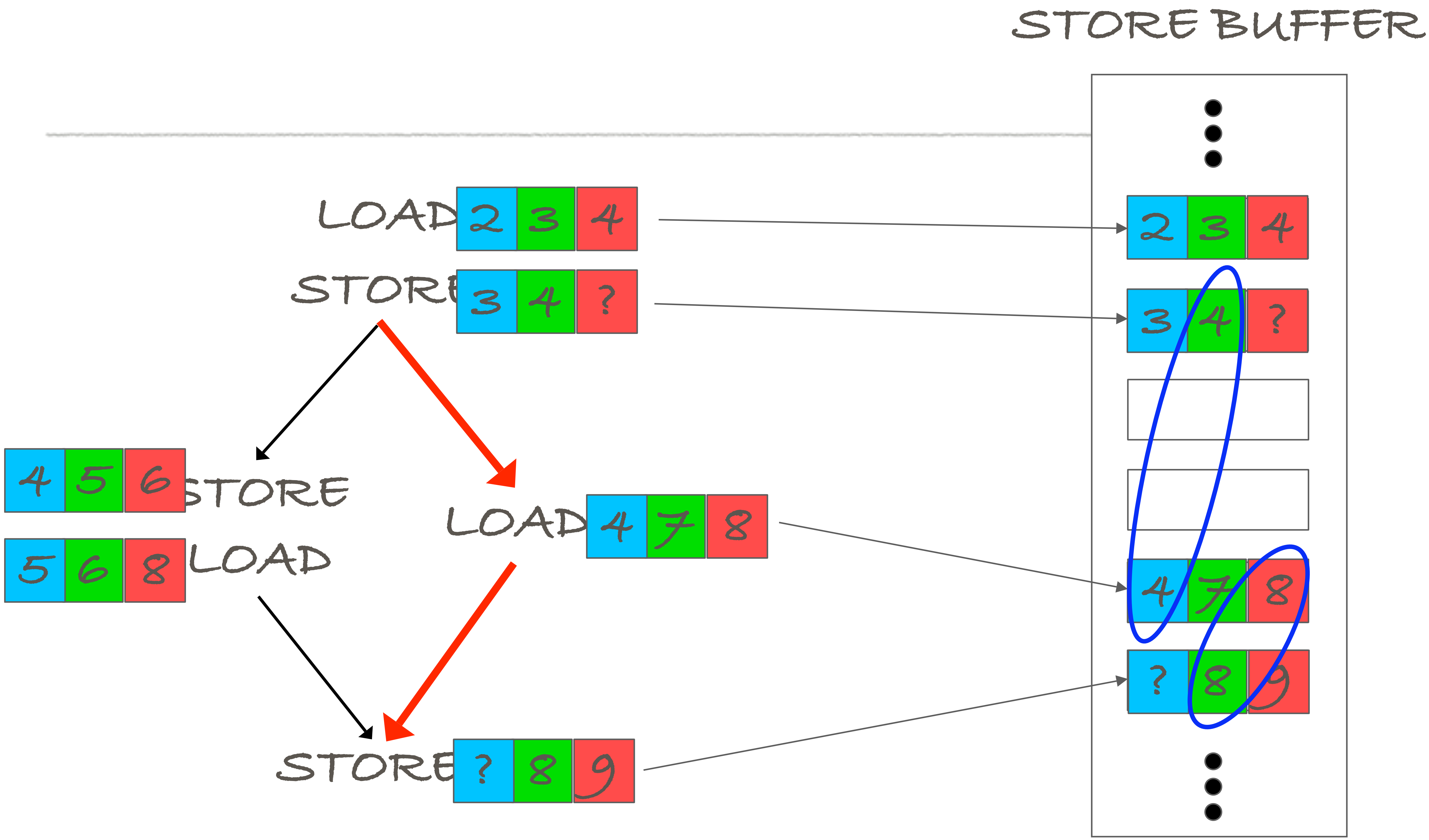
SUCCESSOR

PREDECESSOR



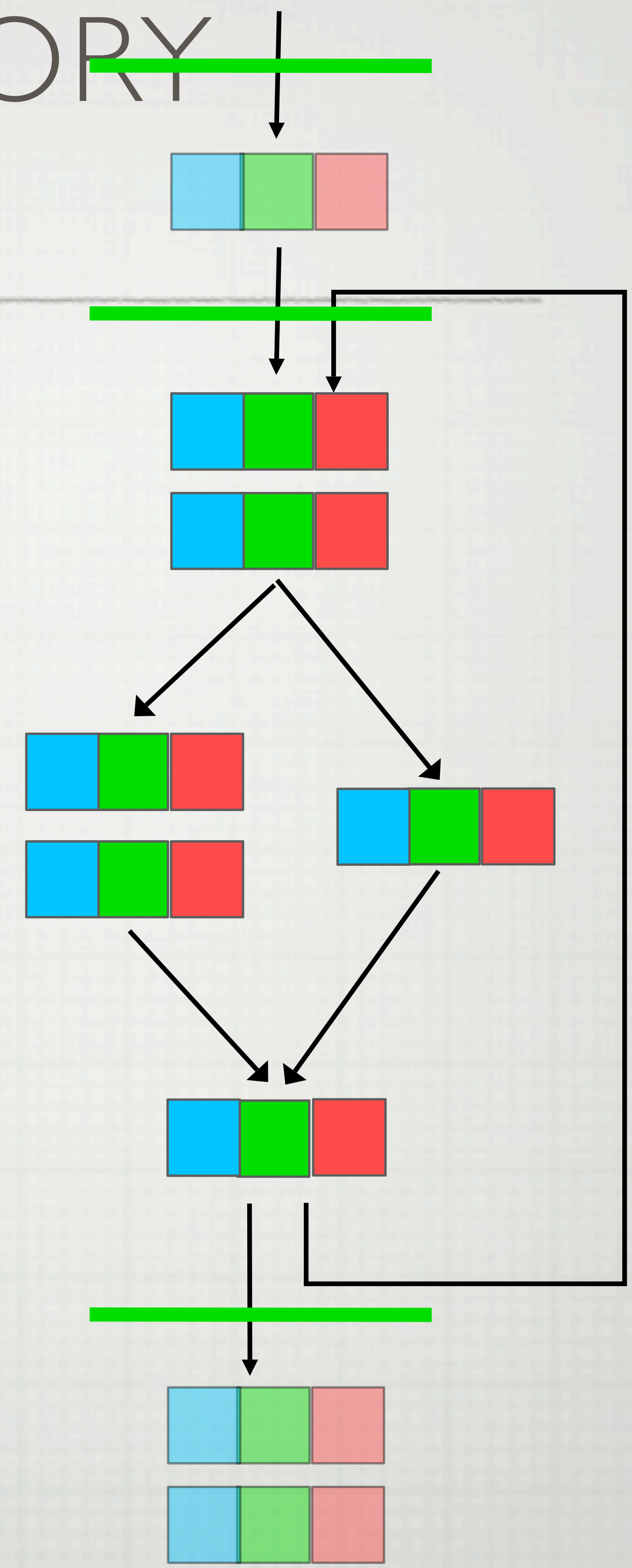
- SEND MEMORY REQUESTS IN ANY ORDER

WAVE-ORDERING EXAMPLE



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