Hazards in von Neumann Architectures

- **Pipeline hazards limit performance**
  - Structural hazards
  - Data hazards due to
    - True dependences
    - Name (false) dependences: anti and output dependences
  - Control hazards

- **Name dependences** can be removed by:
  - compiler (register) renaming
  - renaming hardware $\rightarrow$ advanced superscalars
  - single-assignment rule $\rightarrow$ dataflow computers

- **Data hazards due to true dependences and control hazards** can be avoided if succeeding instructions in the pipeline come from different contexts
  - $\rightarrow$ e.g., multithreaded processors, dataflow machines
Dataflow vs. von Neumann

**von Neumann or control flow computing model:**
- Program: A series of addressable instructions, each of which either
  - Specifies an operation and memory locations of the operands OR
  - Specifies (un)conditional transfer of control to some other instruction
- The next instruction to be executed depends on what happened during the execution of the current instruction
- The next instruction to be executed is pointed to and triggered by the Program Counter (PC)
- The instruction is executed even if some of its operands are not available yet (e.g. uninitialized)

**Dataflow model:** Execution is driven only by the availability of operand
- No PC and global updateable store
- The two features of von Neumann model that become bottlenecks in exploiting parallelism are missing

**Comparison:** Veen Paper figure 2
Main characteristic: The *single-assignment rule*
- A variable may appear on the left side of an assignment only once within the area of the program in which it is active

A dataflow program is compiled into a dataflow graph
- A directed graph consisting of named nodes, which represent instructions, and arcs, which represent data dependences among instructions
- The dataflow graph is similar to a dependence graph used in intermediate representations of compilers

During the execution of the program, data propagate along the arcs in data packets, called *tokens*

This flow of tokens enables some of the nodes (instructions) and fires them
Dataflow Terminology

- **Node**: instruction
- **Token**: data item
- **Arc**: connection between nodes
- **Firing**: execution of a node
- **Enabling rule**: conditions that need to be met in order for a node to fire (enabled)
- **Ports (input, output)**: point where an arc enters or leaves a node
- Example program before and after data flow graphs: Veen paper figure 3
Nodes and Program Structures

- Functional: (+, -, *, /, ^, …)
- Conditional: Veen paper figure 4a
- Merge: Veen paper figure 4b
- Conditional Expressions: Veen paper figure 5
- Loops: Veen paper figure 8
Node Communication and Synchronization

- **Static**
  - Locks (compound branch and merge nodes)
    - Nodes only fire when all inputs are ready
    - Loss of concurrency
  - Acknowledging (control flow protocol)
    - Extra arcs from consumer to producer
    - Increases resources needed

- **Dynamic**
  - Each iteration is executed in a separate instance of the graph
  - Code copying
    - New instance of subgraph is created per iteration
    - Need to direct tokens from earlier iterations to inputs of new iteration
  - Tagged tokens (Veen paper figure 10, 11)
    - Attach a tag to each token, associating it with an iteration
    - Fire when input tokens have all the same tag
Issues with Tagged Tokens

- How to manage tags
  - Size
  - Distribution

- Storage overhead
  - Tags have to be stored with tokens
  - Tokens that cannot be consumed at the moment may need to be stored for later use

- Too much parallelism
  - Storage overflow
  - Deadlocks
Processing Element Architecture

- Dataflow machine contains several processing elements (PEs) that communicate with each other
- Functional diagram of a processing element: Veen paper figure 12
- Processing element operation
  - Enabling unit receives token
  - Enabling unit stores token at addressed node
  - If node is now enabled, send node to functional unit
  - Functional unit processes node
  - Output + destination address are sent back to enabling unit
Tagged Architectures

- Functional diagram of a processing element in a tagged-token machine: Veen paper figure 13

- Processing element operation:
  - Matching unit receives token
  - Check memory: If all other inputs with same tag are there, send all tokens to fetching unit
  - Fetching unit retrieves node from memory
  - Fetching unit assembles an executable packet and sends it to functional unit
  - Functional unit executes node with inputs provided by packet
  - Output is sent back to matching unit
Dataflow Multiprocessors

- **One-level architecture: Veen paper figure 14a**
  - Instructions are executed in the PEs, and results are used in the same PE or communicated to enabling unit of the correct PE

- **Two-level architecture: Veen paper figure 14b**
  - Each functional unit consists of several functional elements that can process packets in parallel
  - An executable packet is allocated to any idle PE

- **Two-stage architecture: Veen paper figure 14c**
  - PEs are split into two stages with a communication medium between the two stages
  - Each enabling unit can send executable packets to any functional unit
  - Suitable for heterogeneous functional units (when some functional elements have specialized capabilities)
Implementing a Tagged-Token Architecture

Tagged-token overview
- Dynamically schedule operations when operands become available
- Attach a tag to each token, associating it with each token
- Fire when inputs tokens all have the same tag

Implementation Issues
- Matching operation involves considerable complexity on the critical path of instruction scheduler
- Failure to match “implicitly” allocates buffer resources
- Inability to simplify resource management
- Managing tags: Size and distribution
- Storage overhead: Tags need to be stored with tokens, tokens that cannot be used currently need to be stored for later use
- Storage overflow
- Potential deadlocks
Explicit Token-Store (ETS) Architecture

- Key differences from tagged token architecture
  - Removes need for associative matching
  - Token storage is explicit
  - Meeting point for operands is determined by simple address calculation (compared to complex hash and match logic)
  - Techniques employed in a von Neumann architecture can be used

- ETS Features
  - Storage of tokens is dynamically allocated
  - When a function is invoked, an activation frame is allocated explicitly (this provides storage for tokens used in the function)
  - Arcs in the graph are mapped to slots in the frame
  - Token = value + IP + FP
  - Each frame slot has “presence bits” indicating the status of the slot
  - 3 atomic operations (r/w/x) are defined on the value part
ETS instruction

- **1-address form**
  - One operand is the value
  - Second is contents of the effective addr (e.g., FP+r)
  - value = acc; IP = PC; FP = index reg

- **Can specify synchronization operation**
  - State transition on the presence bits associated with the memory operand

- **Can specify multiple successors**

- **One-to-one correspondence between tagged-token operations and ETS instructions**
  - But scheduling is simpler in ETS since it doesn’t need complex hashing and matching logic

- **ETS representation of an executing dataflow program:** Papadopoulos paper figure 1
Monsoon

- Prototype built at MIT lab, full scale microprocessor built in conjunction with Motorola

- Monsoon features
  - Contains pipelined PEs connected via a multistage switching network to each other and to a set of interleaved I-structured (IS) memory modules
  - Communication through tokens – no distinction between inter- and intra-processor communication
  - Activation frame created local to PE, resides entirely on one PE
  - A code block is bound (at invocation) to a specific PE on which it executes to completion
  - Concurrent loop iterations are assigned separate activation frames, may execute on separate PEs
    - Reduces inter-PE traffic on network
  - Parallelism within activation frames keeps pipeline full
  - Tag segmented by PE: TAG = PE : (FP.IP)
Monsoon Pipeline

- Each PE uses an eight-stage pipeline (Papadopoulos paper figure 2)
  - Instruction fetch
    - Precedes token matching (unlike associative matching units in dynamic dataflow processors)
  - Effective address generation: explicit token address is computed from the frame address and operand offset
  - Presence bit operation: A presence bit is accessed to find out if the first operand has already arrived
    - Not arrived → presence bit set and the current token is stored into the frame slot of the frame memory
    - Arrived → presence bit is reset and the operand can be retrieved from the slot of the frame memory in next stage
  - Frame operation stage: Operand storing or retrieving.
  - Next 3 stages: execution stages, next tag computed concurrently.
  - Form-token stage (last stage)
    - Forms one or two new tokens that are sent to the network, stored in a user token queue, a system token queue, or directly recirculated to the instruction fetch stage of the pipeline
Reading Assignment

- Karthikeyan Sankaralingam et al., “Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture,” ISCA, 2003 (Skim)
- Steven Swanson et al., “Wavescalar,” MICRO, 2003 (Skim)
- Brucek Khailany et al., “Imagine: Media Processing with Streams,” IEEE Micro, 2001 (Skim)
- Reminder: Project reports and presentations due next Wednesday