Interconnection Networks

- How do we move data between processors?

Design Options:
- Topology
- Routing
- Switching (circuit or packet)
- Flow control
- Deadlock

Crossbar

- Crossbar Switching Networks
  - Non-blocking: No connections block any connections between other processors and memory units; Performance wise scalable
  - Low latency and high throughput
  - Number of switching elements (cost): $O(P^2)$
  - Not scalable (cost)
  - Cray Y-MP

Bus

- Bus based Networks
  - Processors and memory units are connected through a “bus”
  - Simple, cost-effective for small-scale multiprocessors
  - Bus bandwidth limits the number of processors
    - Not scalable (performance)
Multistage
- An intermediate class of networks which lies between crossbar and bus-based networks
  - Performance: more scalable than bus
  - Cost: more scalable than crossbar
- Built from small (e.g., 2x2 crossbar) switch nodes, with a regular interconnection pattern
- Also used in message passing systems (e.g., IBM SP2)

Omega Network
- All stages are same, logP stages
- Uses 2x2 crossbar in each switch
- Cost: O(P log P) switching elements
  - better in comparison with O(P^2) for crossbars
- Single path from source to destination
- Can add extra stages and pathways to minimize collisions and increase fault tolerance
- Blocking network

Omega Network (Cont’d)
Each switch is a 2x2 crossbar
(Broadcast possible too)

Stage i
if source and destination differ in ith bit, “cross”
Otherwise, “straight”
src(010) Æ dest(110)
100 Æ Cross Straight Straight

Butterfly Network (Cont’d)
Destination a2a1a0
In stage i switch
- Send to upper port if ai=0
- Send to lower port if ai=1
Destination 101 Æ Lower Upper Lower

Evaluation Criteria
- Latency/Bandwidth (small and large messages)
- Bisection Width and Bisection Bandwidth
  - Minimum links/volume of communication allowed between any two halves of network with equal number of nodes
- Node degree
  - the number of links connected to a node (processor)
- Diameter
  - Maximum distance between any two processors (maximum latency)
- Connectivity and Partitionability
  - Arc connectivity: Minimum number of arcs that must be removed from network to break it into two disconnected networks
- Cost and scalability
- Symmetry and Homogeneity
- Fault tolerance

Fully Connected
- Not scalable
- Equivalent of crossbar (direct vs. indirect networks)
- Deg = k-1, Diameter = 1, Bisect = k*k/4, Links = k*(k-1)/2
Linear Array / Ring

- Cheap: Cost is $O(N)$
- High overall bandwidth
- High latency ($O(N)$)
- Examples: KSR machine, Hector
- Linear array
  - Diameter = $N$, Degree = 2, Bisection width = 1, Bandwidth = $N-1$, Mean latency = $N/2$, Asymmetric, Heterogeneous
- Ring
  - Diameter = $N$, Degree = 2, Bisection = $N$, Latency = $N/2$, Symmetric, Homogeneous

2-D Mesh / Torus

- Degree = 4, Diameter = $\sqrt{2\log(N)}$, Bisection = $\sqrt{N}$, Easy to build, scalable
- With wraparound links called Torus

Trees

- Cheap: Cost is $O(N)$
- Latency is $O(\log N)$
- Degree = 1, 2, 3, Diameter = $2\log N$, Bisection = 1, Asymmetric
- Easy to layout as planar graphs
- For random permutations, root can become bottleneck.

Fat Trees

- To avoid root being bottleneck, notion of Fat-Trees (used in CM-5)
- Expand bandwidth at each higher level, increases bisection

Hypercubes

- Also called binary $n$-cubes
- Number of nodes $N = 2^n$
- Latency: $O(\log N)$
- Minimizes hops
  - Degree = $n$, Diameter = $n$, Bisection = $2^{(n-1)}$, Nodes = $2^n$, Links = $n \times 2^{(n-1)}$
- Good bisection BW but tough to layout in 3D space
- Popular in early message-passing computers (e.g., intel iPSC, NCUBE)
- Other topologies can be embedded in hypercubes (tree, mesh)

k-ary n-cubes

- Generalization of hypercubes: $k$ nodes (rather than just 2 nodes) in a string
- Total number of nodes $N = k^N$
- Allows for wider channels but requires more hops
Switching Alternatives

- Circuit Switching
- Packet Switching
- Store-and-forward
- Cut-through
  - Virtual cut-through
  - Wormhole

Store and Forward

- Message passes from node to node
- Each node stores the entire message
- After examining the message header, the node forwards it on the appropriate link
- If a blockage appears, messages are held until it clears (multiple messages may accumulate)

Virtual Cut Through

- Messages are passed as a train of packets through a series of nodes
- Only get buffered if they are blocked; accumulate in node at location of lead packet
- Saves intermediate stores and sends, and cuts down on buffer space
- Wormhole Routing Similar to virtual cut-through
  - When message is blocked, trailing packets (flits) are stored at their current node.
  - Limits buffer size to a single packet in each direction

Routing

- For sending a message from a source node to a destination node, routing algorithms determine which path is taken
- Various properties/classifications:
  - Minimal vs. Non-minimal
    - Minimal: always select shortest path
    - Non-minimal: may route the message along a longer path (for example to avoid congestion)

Deterministic vs. Adaptive Routing

- Deterministic: a unique path is determined solely based on source and destination
- Adaptive: Current state of the network is also used to determine the route

Dimension Ordered Routing

- Based on numbering scheme determined by dimension of channel
- Deterministic
- Routes can be quickly determined
- Called XY-routing for 2D meshes
  - Message is first sent along X dimension until reaches the column of destination
  - Message is then sent along Y dimension until reaches destination
- For hypercubes dimension ordered routing is called E-cube routing
**Deadlock**

- How can it arise?
  - necessary conditions:
    - shared resource
    - incrementally allocated
    - non-preemptible
- Think of a channel as a shared resource that is acquired incrementally
  - source buffer then dest. Buffer
  - channels along a route
- **Deadlock free**
  - No traffic pattern can lead to a situation where no packets move forward

**Deadlock Example**

**Deadlock Free**

- How do you avoid it?
  - constrain how channel resources are allocated
    - ex: dimension order
  - XY-routing
    - Removing one of the turns is enough
  - Another approach: add virtual channels
    - Improve the performance too
  - Show that there are no cycles in Channel Dependence Graph

**Routing Design Summary**

- Routing Algorithms restrict the set of routes within the topology
  - simple mechanism selects turn at each hop
  - Virtual cut through and Wormhole routings
- Deadlock-free if channel dependence graph is acyclic
  - limit turns to eliminate dependences
  - add separate channel resources to break dependences
  - combination of topology, algorithm, and switch design
- Deterministic vs. adaptive routing

**General**

- Three parts
  - 90% of the work is the first part
  - After getting that part done, second part is easy
    - Essentially adding while loops in different parts of the code to work on multiple instructions
  - Third part is the easiest
    - Requiring simple change to the Fetch stage
- Pipeline stages
  - Fetch (IF)
  - Decode (ID1)
  - Issue (ID2)
  - EXE (if cycle) EXE (2nd cycle)...
  - LSQ Refresh (LQSR)
  - Writeback (WB)
  - Release FU (RFU)
  - Commit (CM)

**Lab Assignment 4**
Fetch (IF)

- As before (lab 3)
- Do multiple fetches with `mult_fetch` option enabled

Dispatch (ID1)

- Get the first instruction from the Fetch/Dispatch Queue
- Perform functional simulation
- Setup dependency links (if requires output of other units make a note)
  - Make an entry in the odep_list of the instruction producing data
- If ready, put in ready queue, otherwise move on
  - The instruction will be put in ready queue later on (WB/LSQ Refresh) when the dependencies are satisfied
- If ndyn_sched option enabled, do the above for all entries in the Fetch/Dispatch queue

Issue (ID2)

With “ndyn_sche” option enabled
- Inspect all entries in ready queue
  - Mark as complete if Store
  - Assign to FU if requires one and there is one available
  - If can’t process put the entry aside
- Make all the entries which have been put aside the new ready list
- Without the option simply look at the first entry

LSQ Refresh (LSQR)

- Walk through lsq
- Terminate search after first unresolved store
- Stores: if store address known: record it (say in an array)
- Loads: check to see if address conflicts with any stores in lsq
  - If no conflicts enqueue (readyq_enqueue)

Write Back (WB)

- If operation has completed, check out all its outputs
- for all of them
  - update the create vector
  - walk output list (odep_list)
    - Are all the register operands of target instruction ready
      - If yes enqueue non-memory operations
      - If yes enqueue if not a load (loads issued when we check there is no memory conflicts in lsq_refresh)
- Repeat the above with ndyn_sched
- Release FU (RFU) and Commit (CM) similar to those in Lab3