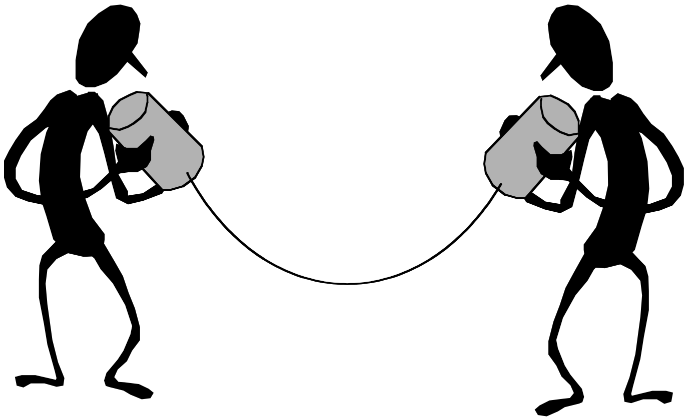
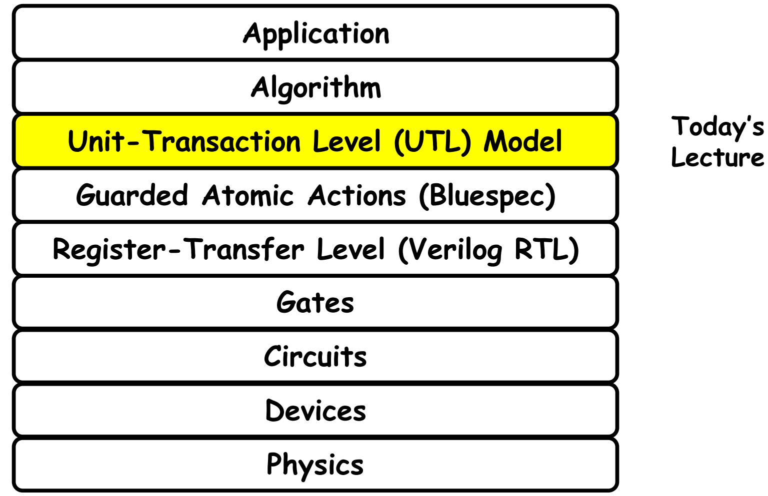
Transaction-Level Design



Hardware Design Abstraction Levels

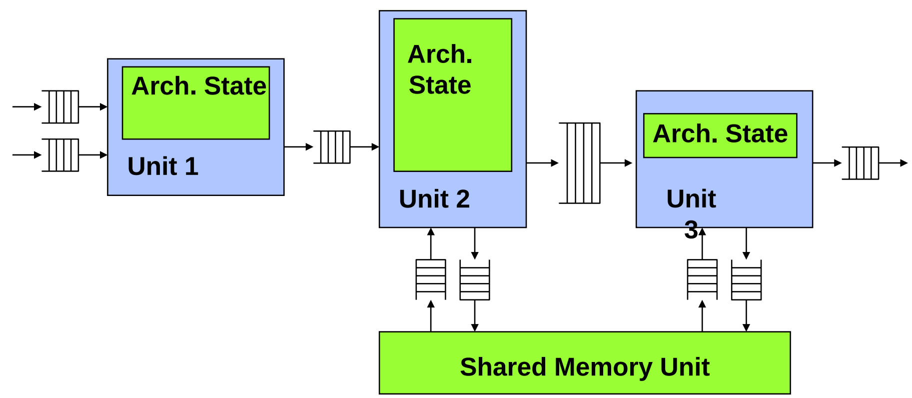


Application to RTL in One Step?

Modern hardware systems have complex functionality (graphics chips, video encoders, wireless communication channels), but sometimes designers try to map directly to an RTL cycle-level microarchitecture in one step

- Requires detailed cycle-level design of each sub-unit
  - Significant design effort required before clear if design will meet goals
- Interactions between units becomes unclear if arbitrary circuit connections allowed between units, with possible cycle-level timing dependencies
  - Increases complexity of unit specifications
- Removes degrees of freedom for unit designers
  - Reduces possible space for architecture exploration
- Difficult to document intended operation, therefore difficult to verify

Transaction-Level Design

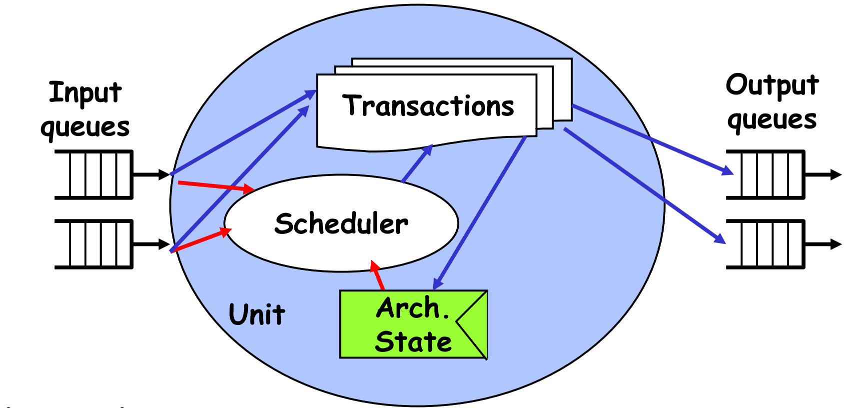


- Model design as messages flowing through FIFO buffers between units containing *architectural* state
- Each unit can independently perform an operation, or *transaction*, that may consume messages, update local state, and send further messages
- Transaction and/or communication might take many cycles (i.e., not necessarily a single Bluespec rule)

6.884 UTL Discipline

- Various forms of transaction-level model are becoming increasingly used in commercial designs
- UTL (Unit-Transaction Level) models are the variant we'll use in 6.884
- UTL forces a *discipline* on top-level design structure that will result in clean hardware designs that are easier to document and verify, and which should lead to better physical designs
  - A discipline *restricts* hardware designs, with the goal of avoiding bad choices
- UTL specs are not directly executable (yet), but could be easily implemented in C/C++/Java/SystemC to give a golden model for design verification
  - Bluespec will often, but not always, be sufficient for UTL model
- You're required to give an initial UTL description (in English text) of your project design by April 1 project milestone

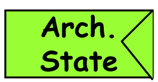
UTL Overview



Unit comprises:

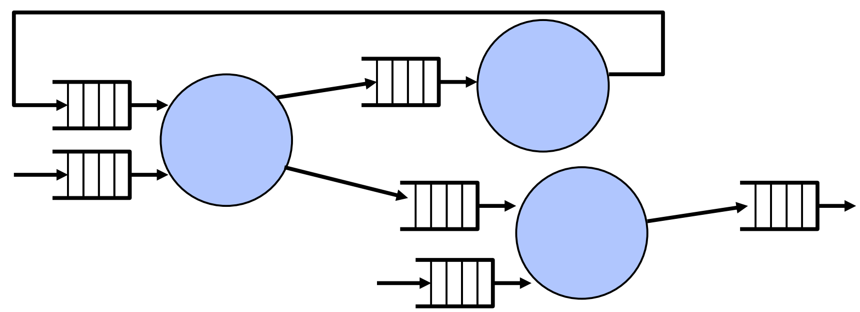
- Architectural state (registers + RAMs)
- Input queues and output queues connected to other units
- Transactions (atomic operations on state and queues)
- Scheduler (combinational function to pick next transaction to run)

Unit Architectural State



- Architectural state is any state that is visible to an external agent
  - i.e, architectural state can be observed by sending strings of packets into input queues and looking at values returned at outputs.
- High-level specification of a unit only refers to architectural state
- Detailed implementation of a unit may have additional *microarchitectural* state that is not visible externally
  - Intra-transaction sequencing logic
  - Pipeline registers
  - Caches/buffers

Queues

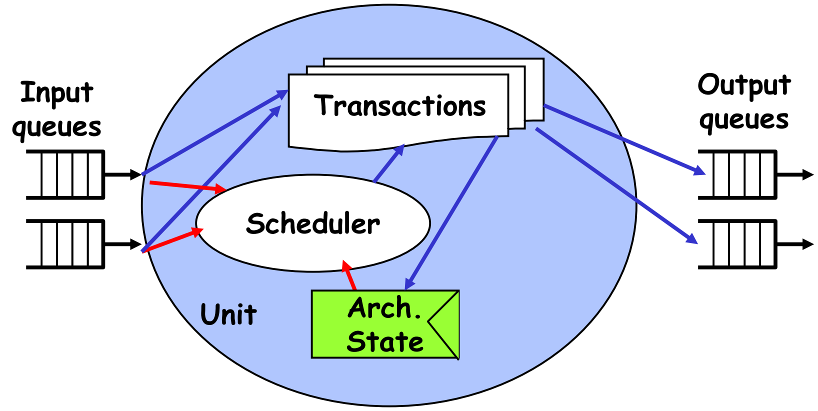


- Queues expose communication latency and decouple units' execution
- Queues are point-to-point channels only
  - No fanout, a unit must replicate messages on multiple queues
  - No buses in a UTL design (though implementation may use them)
- Transactions can only pop head of input queues and push at most one element onto each output queue
  - Avoids exposing size of buffers in queues
  - Also avoids synchronization inherent in waiting for multiple elements

Transactions

- Transaction is a guarded atomic action on local state and input and output queues
  - Similar to Bluespec rule except a transaction might take a variable number of cycles
- Guard is a predicate that specifies when transaction can execute
  - Predicate is over architectural state and heads of input queues
  - Implicit conditions on input queues (data available) and output queues (space available) that transaction accesses
- Transaction can only pop up to one record from an input queue and push up to one record on each output queue

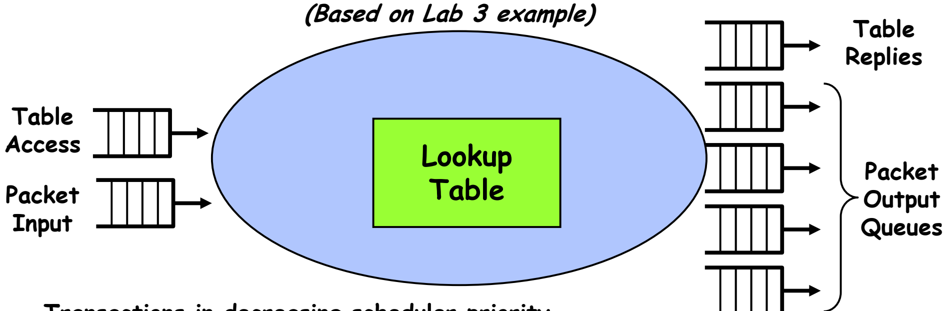
Scheduler



- Scheduling function decides on transaction priority based on local state and state of input queues
  - Simplest scheduler picks arbitrarily among ready transactions
- Transactions may have additional predicates which indicate when they can fire
  - E.g., implicit condition on all necessary output queues being ready

UTL Example: IP Lookup

(Based on Lab 3 example)



Transactions in decreasing scheduler priority

- Table_Write (request on table access queue)
  - Writes a given 12-bit value to a given 12-bit address
- Table_Read (request on table access queue)
  - Reads a 12-bit value given a 12-bit address, puts response on reply queue
- Packet_Process (request on packet input queue)
  - Looks up header in table and places routed packet on correct output queue

This level of detail is all the information we really need to understand what the unit is supposed to do! Everything else is implementation.

UTL & Architectural-Level Verification

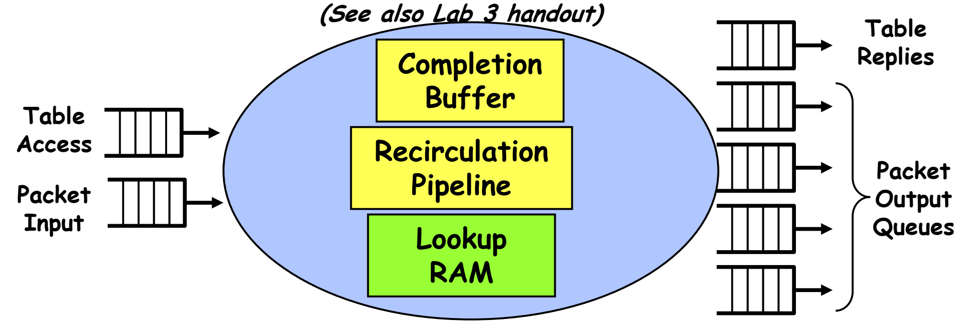
- Can easily develop a *sequential* golden model of a UTL description (pick a unit with a ready transaction and execute that sequentially)
- This is not straightforward if design does not obey UTL discipline
  - Much more difficult if units not decoupled by point-to-point queues, or semantics of multiple operations depends on which other operations run concurrently
- Golden model is important component in verification strategy
  - e.g., can generate random tests and compare candidate design's output against architectural golden model's output

UTL Helps Physical Design

- Restricting inter-unit communication to point-to-point queues simplifies physical layout of units
  - Can add latency on link to accommodate wire delay without changing control logic
- Queues also decouple control logic
  - No interaction between schedulers in different units except via queue full/empty status
  - Bluespec methods can cause arbitrarily deep chain of control logic if units not decoupled correctly
- Units can run at different rates
  - E.g., use more time-multiplexing in unit with lower throughput requirements or use different clock

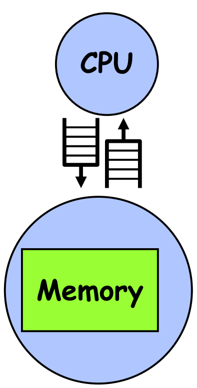
Refining IP Lookup to RTL

(See also Lab 3 handout)



- The recirculation pipeline registers and the completion buffer are microarchitectural state that should be invisible to external units.
- Implementation must ensure atomicity of transactions:
  - Completion buffer ensures packets flow through unit in order
  - Must also ensure table write doesn't appear to happen in middle of packet lookup, e.g., wait for pipeline to drain before performing write

Non-Blocking Cache Example



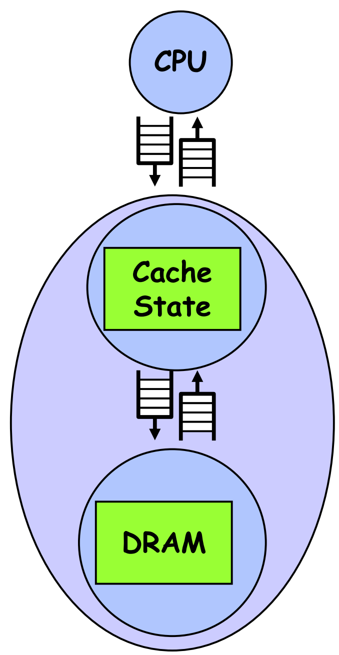
Memory unit transactions:

`Load<address, tag>`
`returns Reply<tag, data>`

`Store<address,data>` modifies memory

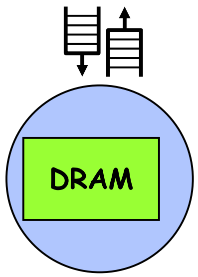
Load replies can be out-of-order
- Spec should strictly split load transaction in two and include additional architectural state in memory unit as otherwise no way for loads to get reordered. Omitted here for clarity.

Refining UTL Design



- Memory unit implemented as two communicating units, Cache and DRAM
- CPU's view of Memory unit unchanged
  - i.e., the cache state should not be visible to the CPU

A DRAM Unit

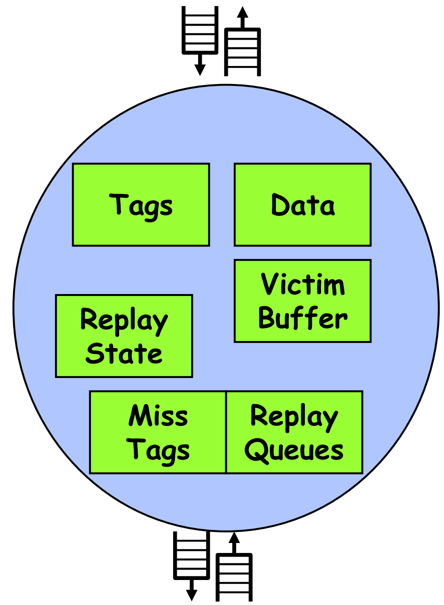


DRAM Unit reads and writes whole cache lines (four words) in order

Transactions:

- LoadLine<addr> returns RepLine<dataline> from DRAM
- StoreLine<addr,dataline> updates DRAM

Non-Blocking Cache Unit



Victim Buffer holds evicted dirty line awaiting writeback to DRAM (writeback cache)

Miss Tags hold address of all cache miss requests pending in DRAM unit

Replay Queues hold secondary misses for each miss tag already requested from DRAM

Replay State holds state of any active replay of a returned cache line

CPU Load Transaction

Load<addr,tag> (if miss tag and replay queue free)
- if (cache hit on addr) then
  - update replacement policy state bits
  - return Reply<tag,data> to CPU
- else
  - if (hit in miss tags) then
    - append request <R,tag,addr[1:0]> to associated Replay Queue
  - else
    - allocate new miss tag and append <R,tag,addr[1:0]> to Replay Queue
    - send LoadLine<addr> to DRAM unit
    - select victim line according to replacement policy
    - if victim dirty then copy to victim buffer
    - invalidate victim's in-cache tag

Replay Queue holds entries with tag and offset of requested word within cache line (addr<1:0>)

CPU Store Transaction

Store<addr,data> (if miss tag and replay queue free)
- if (cache hit on addr) then
  - update replacement policy state bits
  - update cache data and set dirty bit on line
- else
  - if (hit in miss tags) then
    - append request <W,addr[1:0],data> to associated Replay Queue
  - else
    - allocate new miss tag and append <W,addr[1:0],data> to Replay Queue
    - send LoadLine<addr> to DRAM unit
    - select victim line according to replacement policy
    - if victim dirty then copy to victim buffer
    - invalidate victim's in-cache tag

Victim Writeback Transaction

(if buffered victim)
  send StoreLine<victim.addr,victim.dataline> to DRAM unit
  clear victim buffer

DRAM Response Transactions

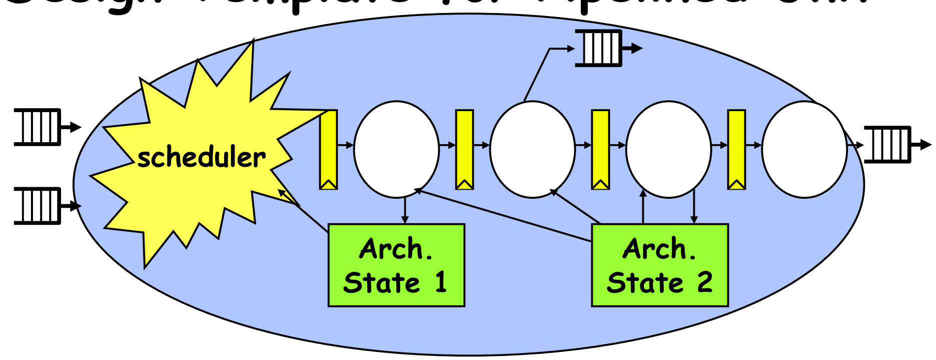
RepLine <dataline> /* Receive DRAM Response Transaction */
  locate associated miss tag (allocated in circular order)
  locate invalid line in destination cache set
  overwrite victim tag and data with new line
  initialize replay state with new line and replay queue

(if replay state valid) /* Replay Transaction */
  read next replay queue entry
  if <R,addr,tag>, read from line and send Reply<tag,data> to CPU
  if <W,addr,data> write data to line and set its dirty bit
  if no more reply queue entries then
    clear replay state
    deallocate miss tags and replay queue (circular buffer)

Cache Scheduler

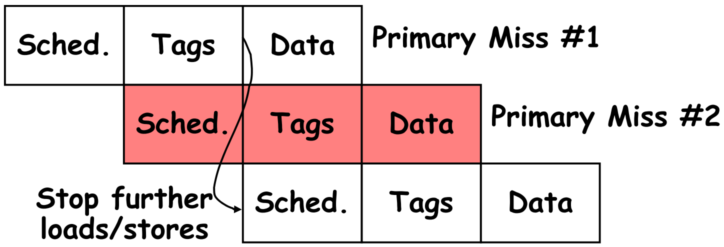
Descending Priority
- Replay
- DRAM Response
- Victim Writeback
- CPU Load or Store

Design Template for Pipelined Unit



- Scheduler only fires transaction when it can complete without stalls
  - Avoids driving heavily loaded stall signals
- Architectural state (and outputs) only written in one stage of pipeline, only read in same or earlier stages
  - Simplifies hazard detection/prevention
- Have different transaction types access expensive units (RAM read ports, shifters, multiply units) in same pipeline stage to reduce area

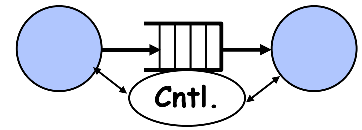
Skid Buffering



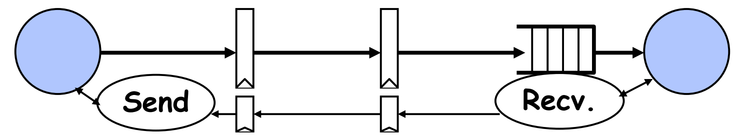
- Consider non-blocking cache implemented as a three stage pipeline: (scheduler, tag access, data access)
- CPU Load/Store not admitted into pipeline unless miss tag, reply queue,and victim buffer available in case of miss
- If hit/miss determined at end of Tags stage, then second miss could enter pipeline
- Solutions?
  - Could only allow one load/store every two cycles => low throughput
  - Skid buffering: Add additional victim buffer, miss tags, and replay queues to complete following transaction if miss. Stall scheduler whenever there is not enough space for *two* misses.

Implementing Communication Queues

- Queue can be implemented as centralized FIFO with single control FSM if both ends are close to each other and directly connected:

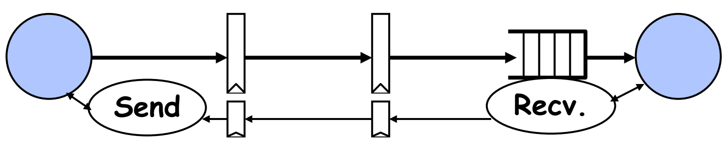


- In large designs, there may be several cycles of communication latency from one end to other. This introduces delay both in forward data propagation and in reverse flow control



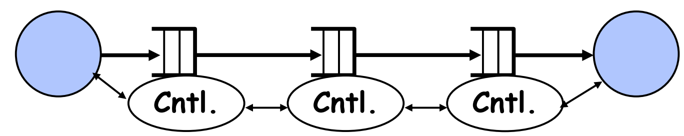
- Control split into send and receive portions. A *credit-based flow control* scheme is often used to tell sender how many units of data it can send before overflowing receivers buffer.

End-End Credit-Based Flow Control



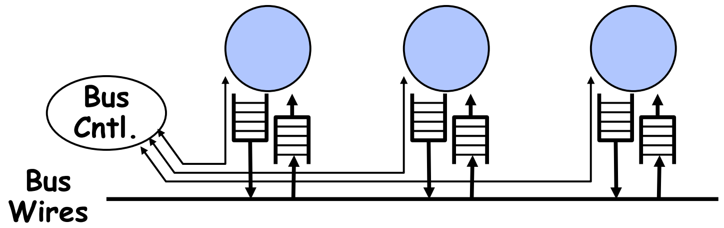
- For one-way latency of N cycles, need 2*N buffers at receiver
  - Will take at least 2N cycles before sender can be informed that first unit sent was consumed (or not) by receiver
- If receive buffer fills up and stalls communication, will take N cycles before first credit flows back to sender to restart flow

Distributed Flow Control



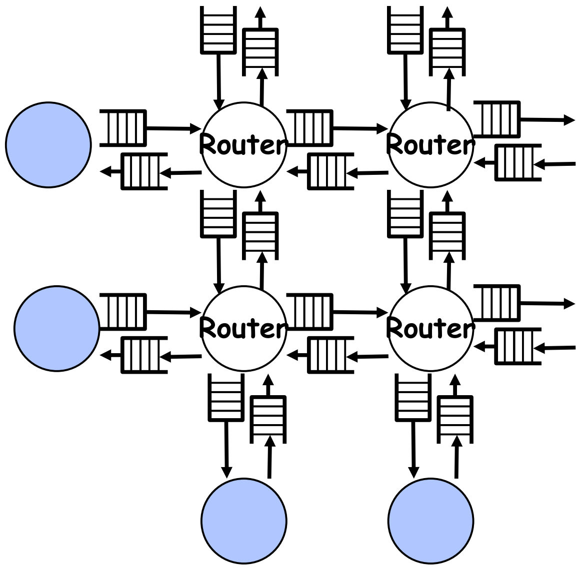
- An alternative to end-end control is distributed flow control (chain of FIFOs)
- Lower restart latency after stalls
- Can require more circuitry and can increase end-end latency

Buses



- Buses were popular board-level option for implementing communication as they saved pins and wires
- Less attractive on-chip as wires are plentiful and buses are slow and cumbersome with central control
- Often used on-chip when shrinking existing legacy system design onto single chip
- Newer designs moving to either dedicated point-point unit communications or an on-chip network

On-Chip Network



- On-chip network multiplexes long range wires to reduce cost
- Routers use distributed flow control to transmit packets
- Units usually need end-end credit flow control in addition because intermediate buffering in network is shared by all units