Discrete Event

- Explicit notion of time (global order...)
- DE simulator maintains a global event queue (Verilog and VHDL)

Drawbacks
- Global event queue => tight coordination between parts
- Simultaneous events => non-deterministic behavior

Some simulators use delta delay to prevent non-determinacy

Simultaneous Events in DE

Fire B or C?

B has 0 delay

B has delta delay

Fire C once? or twice?

Can be refined
E.g. introduce timing constraints
(minimum reaction time 0.1 s)

Fire C twice.

Still have problem with 0-delay (causality) loop
Outline

◆ Synchrony and asynchrony

◆ CFSM definitions
  • Signals & networks
  • Timing behavior
  • Functional behavior

◆ CFSM & process networks

◆ Example of CFSM behaviors
  • Equivalent classes

Codesign Finite State Machine

◆ Underlying MOC of Polis

◆ Combine aspects from several other MOCs

◆ Preserve formality and efficiency in implementation

◆ Mix
  • synchronicity
    ♦ zero and infinite time
  • asynchronicity
    ♦ non-zero, finite, and bounded time

◆ Embedded systems often contain both aspects
**Synchrony: Basic Operation**

- Synchrony is often implemented with clocks

- At clock ticks
  - Module reads inputs, computes, and produce output
  - All synchronous events happen simultaneously
  - Zero-delay computations

- Between clock ticks
  - Infinite amount of time passed

---

**Synchrony: Basic Operation (2)**

- Practical implementation of synchrony
  - Impossible to get zero or infinite delay
  - Require: computation time <<< clock period
  - Computation time = 0, w.r.t. reaction time of environment

- Feature of synchrony
  - Functional behavior independent of timing
    - Simplify verification
  - Cyclic dependencies may cause problem
    - Among (simultaneous) synchronous events
**Synchrony: Triggering and Ordering**

- All modules are triggered at each clock tick
- Simultaneous signals
  - No a priori ordering
  - Ordering may be imposed by dependencies
    - Implemented with delta steps

![Diagram showing ticks and delta steps](image)

**Synchrony: System Solution**

- System solution
  - Output reaction to a set of inputs
- Well-designed system:
  - Is completely specified and functional
  - Has an unique solution at each clock tick
  - Is equivalent to a single FSM
  - Allows efficient analysis and verification
- Well-design-ness
  - May need to be checked for each design (Esterel)
    - Cyclic dependency among simultaneous events
**Synchrony: Implementation Cost**

- Must verify synchronous assumption on final design
  - May be expensive

- Examples:
  - Hardware
    - Clock cycle > maximum computation time
      - Inefficient for average case
  - Software
    - Process must finish computation before
      - New input arrival
      - Another process needs to start computation

**Asynchrony: Basic Operation**

- Events are never simultaneous
  - No two events have the same tag

- Computation starts at a change of the input

- Delays are arbitrary, but bounded
Asynchrony: Triggering and Ordering

- Each module is triggered to run at a change of input
- No a priori ordering among triggered modules
  - May be imposed by scheduling at implementation

Asynchrony: System Solution

- Solution strongly dependent on input timing
- At implementation
  - Events may “appear” simultaneous
  - Difficult/expensive to maintain total ordering
    - Ordering at implementation decides behavior
    - Becomes DE, with the same pitfalls
Asynchrony: Implementation Cost

◆ Achieve low computation time (average)
  • Different parts of the system compute at different rates

◆ Analysis is difficult
  • Behavior depends on timing
  • Maybe be easier for designs that are insensitive to
    ✦ Internal delay
    ✦ External timing

Asynchrony vs. Synchrony in System Design

◆ They are different at least at
  • Event buffering
  • Timing of event read/write

◆ Asynchrony
  • Explicit buffering of events for each module
    ✦ Vary and unknown at start-time

◆ Synchrony
  • One global copy of event
    ✦ Same start time for all modules
Combining Synchrony and Asynchrony

◆ Wants to combine
  ◦ Flexibility of asynchrony
  ◦ Verifiability of synchrony

◆ Asynchrony
  ◦ Globally, a timing independent style of thinking

◆ Synchrony
  ◦ Local portion of design are often tightly synchronized

◆ Globally asynchronous, locally synchronous
  ◦ CFSM networks

CFSM Overview

◆ CFSM is FSM extended with
  ◦ Support for data handling
  ◦ Asynchronous communication

◆ CFSM has
  ◦ FSM part
    ◦ Inputs, outputs, states, transition and output relation
  ◦ Data computation part
    ◦ External, instantaneous functions
CFSM Overview (2)

◆ CFSM has:
  ♦ Locally synchronous behavior
    ♦ CFSM executes based on snap-shot input assignment
    ♦ Synchronous from its own perspective
  ♦ Globally asynchronous behavior
    ♦ CFSM executes in non-zero, finite amount of time
    ♦ Asynchronous from system perspective

◆ GALS model
  ♦ Globally: Scheduling mechanism
  ♦ Locally: CFSMs

Network of CFSMs: Depth-1 Buffers

◆ Globally Asynchronous, Locally Synchronous (GALS) model
Introducing a CFSM

◆ A Finite State Machine
◆ Input events, output events and state events
◆ Initial values (for state events)
◆ A transition function
  ➔ Transitions may involve complex, memory-less, instantaneous arithmetic and/or Boolean functions
  ➔ All the state of the system is under form of events
◆ Need rules that define the CFSM behavior

CFSM Rules: phases

◆ Four-phase cycle:
  ★ Idle
  ☰ Detect input events
  ☰ Execute one transition
  ☰ Emit output events
◆ Discrete time
  ▪ Sufficiently accurate for synchronous systems
  ▪ Feasible formal verification
◆ Model semantics: Timed Traces, i.e. sequences of events labeled by time of occurrence
**CFSM Rules: phases**

- Implicit *unbounded delay* between phases
- *Non-zero* reaction time  
  (avoid *inconsistencies* when interconnected)
- *Causal* model based on *partial order*  
  *(global asynchronicity)*
  - potential verification speed-up
- *Phases may not overlap*
- *Transitions always clear input buffers*  
  *(local synchronicity)*

**Communication Primitives**

- **Signals**
  - Carry information in the form of events and/or values
    - Event signals: present/absence
    - Data signals: arbitrary values  
      - Event, data may be paired
  - Communicate between two CFSMs  
    - 1 input buffer / signal / receiver
  - Emitted by a sender CFSM
  - Consumed by a receiver CFSM by setting buffer to 0
  - “Present” if emitted but not consumed
**Communication Primitives (2)**

- **Input assignment**
  - A set of values for the input signals of a CFSM
- **Captured input assignment**
  - A set of input values read by a CFSM at a particular time
- **Input stimulus**
  - Input assignment with at least one event present

**Signals and CFSM**

- **CFSM**
  - Initiates communication through events
  - Reacts only to input stimulus
    - except initial reaction
  - Writes data first, then emits associated event
  - Reads event first, then reads associated data
**CFSM networks**

- **Net**
  - A set of connections on the same signal
  - Associated with single sender and multiple receivers
  - An input buffer for each receiver on a net
    - Multi-cast communication

- **Network of CFSMs**
  - A set of CFSMs, nets, and a scheduling mechanism
  - Can be implemented as
    - A set of CFSMs in SW (program/compiler/OS/uC)
    - A set of CFSMs in HW (HDL/gate/clocking)
    - Interface (polling/interrupt/memory-mapped)

**Scheduling Mechanism**

- **At the specification level**
  - Should be as abstract as possible to allow optimization
  - Not fixed in any way by CFSM MOC

- **May be implemented as**
  - RTOS for single processor
  - Concurrent execution for HW
  - Set of RTOSs for multi-processor
  - Set of scheduling FSMs for HW
**Timing Behavior**

◆ **Scheduling Mechanism**
  - Globally controls the interaction of CFSMs
  - Continually deciding which CFSMs can be executed

◆ **CFSM can be**
  - Idle
    - Waiting for input events
    - Waiting to be executed by scheduler
  - Executing
    - Generate a single reaction
    - Reads its inputs, computes, writes outputs

**Timing Behavior: Mathematical Model**

◆ **Transition Point**
  - Point in time a CFSM starts executing

◆ **For each execution**
  - Input signals are read and cleared
  - Partial order between input and output
  - Event is read before data
  - Data is written before event emission
**Timing Behavior: Transition Point**

◆ A transition point \( t_i \)
  - Input may be read between \( t_i \) and \( t_{i+1} \)
  - Event that is read may have occurred between \( t_{i-1} \) and \( t_{i+1} \)
  - Data that is read may have occurred between \( t_0 \) and \( t_{i+1} \)
  - Outputs are written between \( t_i \) and \( t_{i+1} \)

◆ CFSM allow loose synchronization of event & data
  - Less restrictive implementation
  - May lead to non intuitive behavior

---

**Event/Data Separation**

- Value \( v_1 \) is lost even though
  - It is sent with an event
  - Event may not be lost

◆ Need atomicity
Atomicity

◆ Group of actions considered as a single entity
◆ May be costly to implement
◆ Only atomicity requirement of CFSM
  ♦ Input events are read atomically
    ♦ Can be enforced in SW (bit vector) HW (buffer)
    ♦ CFSM is guaranteed to see a snapshot of input events
◆ Non-atomicity of event and data
  ♦ May lead to undesirable behavior
  ♦ Atomicized as an implementation trade-off decision

Non Atomic Data Value Reading

◆ Receiver R1 gets (X=4, Y=5), R2 gets (X=5 Y=4)
◆ X=4 Y=5 never occurs
◆ Can be remedied if values are sent with events
  ♦ still suffers from separation of data and event
**Atomicity of Event Reading**

Each sees a snapshot of events in time

Different captured input assignment
- Because of scheduling and delay

**Functional Behavior**

- Transition and output relations
  - input, present_state, next_state, output

- At each execution, a CFSM
  - Reads a captured input assignment
  - If there is a match in transition relation
    - consume inputs, transition to next_state, write outputs
  - Otherwise
    - consume no inputs, no transition, no outputs
**Functional Behavior (2)**

◆ Empty Transition
  - No matching transition is found

◆ Trivial Transition
  - A transition that has no output and no state changes
  - Effectively throw away inputs

◆ Initial transition
  - Transition to the init (reset) state
  - No input event needed for this transition

---

**CFSM and Process Networks**

◆ CFSM
  - An asynchronous extended FSM model
  - Communication via bounded non-blocking buffers
    ◆ Versus CSP and CCS (rendezvous)
    ◆ Versus SDL (unbounded queue & variable topology)
  - Not continuous in Kahn’s sense
    ◆ Different event ordering may change behavior
      ◆ Versus dataflow (ordering insensitive)
**CFSM Networks**

- Defined based on a global notion of time
  - Total order of events
  - Synchronous with relaxed timing
    - Global consistent state of signals is required
    - Input and output are in partial order

**Buffer Overwrite**

- CFSM Network has
  - Finite Buffering
  - Non-blocking write
    - Events can be overwritten
      - if the sender is “faster” than receiver

- To ensure no overwrite
  - Explicit handshaking mechanism
  - Scheduling
Example of CFSM Behaviors

- A and B produce i1 and i2 at every i
- C produce err or o at every i1,i2
- Delay (i to o) for normal operation is nr, err operation 2nr
- Minimum input interval is ni
- Intuitive “correct” behavior
  - No events are lost

Equivalent Classes of CFSM Behavior

- Assume parallel execution (HW, 1 CFSM/processor)
- Equivalent classes of behaviors are:
  - Zero Delay
    - nr= 0
  - Input buffer overwrite
    - ni<nr
  - Time critical operation
    - ni/2<nr≤ni
  - Normal operation
    - nr<ni/2
Equivalent Classes of CFSM Behavior (2)

◆ Zero delay: \( n_r = 0 \)
  - If C emits an error on some input
    - A, B can react instantaneously & output differently
  - May be logically inconsistent

◆ Input buffers overwrite: \( n_i < n_r \)
  - Execution delay of A, B is larger than arrival interval
    - always loss of event
    - requirements not satisfied

Equivalent Classes of CFSM Behavior (3)

◆ Time critical operation: \( n_i/2 < n_r \leq n_i \)
  - Normal operation results in no loss of event
  - Error operation may cause lost input

◆ Normal operation: \( n_r < n_i/2 \)
  - No events are lost
  - May be expensive to implement

◆ If error is infrequent
  - Designer may accept also time critical operation
    - Can result in lower-cost implementation
Equivalent Classes of CFSM Behavior (4)

◆ Implementation on a single processor
  • Loss of Event may be caused by
    ◆ Timing constraints
      • \( n < 3n_r \)
    ◆ Incorrect scheduling
      • If empty transition also takes \( n_r \)
        • ACBC round robin will miss event
        • ABC round robin will not

Some Possibility of Equivalent Classes

◆ Given 2 arbitrary implementations, 1 input stream:
  • Dataflow equivalence
    ◆ Output streams are the same ordering
  • Petri net equivalence
    ◆ Output streams satisfied some partial order
  • Golden model equivalence
    ◆ Output streams are the same ordering
      ◆ Except reordering of concurrent events
    ◆ One of the implementations is a reference specification
  • Filtered equivalence
    ◆ Output streams are the same after filtered by observer
Conclusion

◆ CFMS
  • Extension: ACFSM: Initially unbounded FIFO buffers
    ◆ Bounds on buffers are imposed by refinement to yield ECFSM
  • Delay is also refined by implementation
  • Local synchrony
    ◆ Relatively large atomic synchronous entities
  • Global asynchrony
    ◆ Break synchrony, no compositional problem
    ◆ Allow efficient mapping to heterogeneous architectures

Data-flow networks

◆ A bit of history
◆ Syntax and semantics
  • actors, tokens and firings
◆ Scheduling of Static Data-flow
  • static scheduling
  • code generation
  • buffer sizing
◆ Other Data-flow models
  • Boolean Data-flow
  • Dynamic Data-flow
Data-flow networks

- Powerful formalism for data-dominated system specification
- Partially-ordered model (no over-specification)
- Deterministic execution independent of scheduling
- Used for
  - simulation
  - scheduling
  - memory allocation
  - code generation
  for Digital Signal Processors (HW and SW)

A bit of history

- Karp computation graphs (‘66): seminal work
- Kahn process networks (‘58): formal model
- Dennis Data-flow networks (‘75): programming language for MIT DF machine
- Several recent implementations
  - graphical:
    - Ptolemy (UCB), Khoros (U. New Mexico), Grape (U. Leuven)
    - SPW (Cadence), COSSAP (Synopsys)
  - textual:
    - Silage (UCB, Mentor)
    - Lucid, Haskell
**Data-flow network**

- A Data-flow network is a collection of **functional nodes** which are connected and communicate over **unbounded FIFO queues**
- Nodes are commonly called **actors**
- The bits of information that are communicated over the queues are commonly called **tokens**

**Intuitive semantics**

- (Often stateless) actors perform computation
- Unbounded FIFOs perform communication via **sequences of tokens** carrying values
  - integer, float, fixed point
  - matrix of integer, float, fixed point
  - image of pixels
- State implemented as self-loop
- **Determinacy:**
  - unique output sequences given unique input sequences
  - Sufficient condition: **blocking read**
    (process cannot test input queues for emptiness)
**Intuitive semantics**

- At each time, one actor is **fired**
- When firing, actors **consume** input tokens and **produce** output tokens
- Actors can be fired only if there are enough tokens in the input queues

**Example: FIR filter**
- single input sequence $i(n)$
- single output sequence $o(n)$
- $o(n) = c_1 i(n) + c_2 i(n-1)$
Intuitive semantics

Example: FIR filter
- single input sequence $i(n)$
- single output sequence $o(n)$
- $o(n) = c_1 i(n) + c_2 i(n-1)$

\[ o(n) = c_1 i(n) + c_2 i(n-1) \]

Intuitive semantics

Example: FIR filter
- single input sequence $i(n)$
- single output sequence $o(n)$
- $o(n) = c_1 i(n) + c_2 i(n-1)$

\[ o(n) = c_1 i(n) + c_2 i(n-1) \]
Intuitive semantics

- Example: FIR filter
  - single input sequence $i(n)$
  - single output sequence $o(n)$
  - $o(n) = c_1 i(n) + c_2 i(n-1)$
Intuitive semantics

◆ Example: FIR filter
  • single input sequence \( i(n) \)
  • single output sequence \( o(n) \)
  • \( o(n) = c_1 i(n) + c_2 i(n-1) \)
Intuitive semantics

◆ Example: FIR filter
  - single input sequence i(n)
  - single output sequence o(n)
  - o(n) = c1 i(n) + c2 i(n-1)
**Intuitive semantics**

- Example: FIR filter
  - single input sequence $i(n)$
  - single output sequence $o(n)$
  - $o(n) = c_1 i(n) + c_2 i(n-1)$

**Questions**

- Does the order in which actors are fired affect the final result?
- Does it affect the “operation” of the network in any way?
- Go to Radio Shack and ask for an unbounded queue!!
**Formal semantics: sequences**

- Actors operate from a sequence of input tokens to a sequence of output tokens.
- Let tokens be noted by \( x_1, x_2, x_3, \) etc...
- A sequence of tokens is defined as
  \[
  X = [ x_1, x_2, x_3, \ldots ]
  \]
- Over the execution of the network, each queue will grow a particular sequence of tokens.
- In general, we consider the actors mathematically as functions from sequences to sequences (not from tokens to tokens).

**Ordering of sequences**

- Let \( X_1 \) and \( X_2 \) be two sequences of tokens.
- We say that \( X_1 \) is less than \( X_2 \) if and only if (by definition)
  \( X_1 \) is an initial segment of \( X_2 \)
- Homework: prove that the relation so defined is a partial order (reflexive, antisymmetric and transitive)
- This is also called the prefix order.
- Example: \([ x_1, x_2 ] \leq [ x_1, x_2, x_3 ]\)
- Example: \([ x_1, x_2 ]\) and \([ x_1, x_3, x_4 ]\) are incomparable.
Chains of sequences

◆ Consider the set $S$ of all finite and infinite sequences of tokens
◆ This set is partially ordered by the prefix order
◆ A subset $C$ of $S$ is called a chain iff all pairs of elements of $C$ are comparable
◆ If $C$ is a chain, then it must be a linear order inside $S$ (otherwise, why call it chain?)
◆ Example: $\{ [x_1], [x_1, x_2], [x_1, x_2, x_3], \ldots \}$ is a chain
◆ Example: $\{ [x_1], [x_1, x_2], [x_1, x_3], \ldots \}$ is not a chain

(Least) Upper Bound

◆ Given a subset $Y$ of $S$, an upper bound of $Y$ is an element $z$ of $S$ such that $z$ is larger than all elements of $Y$
◆ Consider now the set $Z$ (subset of $S$) of all the upper bounds of $Y$
◆ If $Z$ has a least element $u$, then $u$ is called the least upper bound (lub) of $Y$
◆ The least upper bound, if it exists, is unique
◆ Note: $u$ might not be in $Y$ (if it is, then it is the largest value of $Y$)
**Complete Partial Order**

- Every chain in S has a least upper bound
- Because of this property, S is called a **Complete Partial Order**
- Notation: if C is a chain, we indicate the least upper bound of C by \( \text{lub}(C) \)
- Note: the least upper bound may be thought of as the limit of the chain

**Processes**

- Process: function from a p-tuple of sequences to a q-tuple of sequences
  
  \[ F : S^p \rightarrow S^q \]

- Tuples have the induced point-wise order:
  
  \[ Y = (y_1, \ldots, y_p), \ Y' = (y'_1, \ldots, y'_p) \text{ in } S^p : Y \leq Y' \text{ iff } y_i \leq y'_i \text{ for all } 1 \leq i \leq p \]

- Given a chain C in \( S^p \), \( F(C) \) may or may not be a chain in \( S^q \)
- We are interested in conditions that make that true
**Continuity and Monotonicity**

- **Continuity**: $F$ is continuous iff (by definition) for all chains $C$, $\text{lub}(F(C))$ exists and
  \[ F(\text{lub}(C)) = \text{lub}(F(C)) \]
- **Similar to continuity in analysis using limits**
- **Monotonicity**: $F$ is monotonic iff (by definition) for all pairs $X, X'$
  \[ X \leq X' \Rightarrow F(X) \leq F(X') \]
- **Continuity implies monotonicity**
  - intuitively, outputs cannot be “withdrawn” once they have been produced
  - timeless causality. $F$ transforms chains into chains

**Least Fixed Point semantics**

- **Let $X$ be the set of all sequences**
- **A network is a mapping $F$ from the sequences to the sequences**
  \[ X = F(X, I) \]
- **The behavior of the network is defined as the unique least fixed point of the equation**
- **If $F$ is continuous then the least fixed point exists**
  \[ \text{LFP} = \text{LUB}(\{ F^n(\bot, I) : n \geq 0 \}) \]
From Kahn networks to Data Flow networks

- Each process becomes an actor: set of pairs of
  - firing rule
    (number of required tokens on inputs)
  - function
    (including number of consumed and produced tokens)

- Formally shown to be equivalent, but actors with firing are more intuitive
- Mutually exclusive firing rules imply monotonicity
- Generally simplified to blocking read

Examples of Data Flow actors

- SDF: Synchronous (or, better, Static) Data Flow
  - fixed input and output tokens

- BDF: Boolean Data Flow
  - control token determines consumed and produced tokens
Static scheduling of DF

- Key property of DF networks: output sequences do not depend on time of firing of actors
- SDF networks can be statically scheduled at compile-time
  - execute an actor when it is known to be fireable
  - no overhead due to sequencing of concurrency
  - static buffer sizing
- Different schedules yield different
  - code size
  - buffer size
  - pipeline utilization

Static scheduling of SDF

- Based only on process graph (ignores functionality)
- Network state: number of tokens in FIFOs
- Objective: find schedule that is valid, i.e.:
  - admissible
    - (only fires actors when fireable)
  - periodic
    - (brings network back to initial state firing each actor at least once)
- Optimize cost function over admissible schedules
**Balance equations**

- Number of produced tokens must equal number of consumed tokens on every edge

```
   A   n_p    n_c   B
```

- Repetitions (or firing) vector $v_S$ of schedule $S$: number of firings of each actor in $S$

  - $v_S(A) \cdot n_p = v_S(B) \cdot n_c$
  - must be satisfied for each edge

**Balance equations**

- Balance for each edge:
  - $3 \cdot v_S(A) - v_S(B) = 0$
  - $v_S(B) - v_S(C) = 0$
  - $2 \cdot v_S(A) - v_S(C) = 0$
  - $2 \cdot v_S(A) - v_S(C) = 0$
**Balance equations**

\[
\begin{bmatrix}
3 & -1 & 0 \\
0 & 1 & -1 \\
2 & 0 & -1 \\
2 & 0 & -1
\end{bmatrix}
\]

\( M v_S = 0 \)

iff \( S \) is periodic

\( M \) is Full rank (as in this case)
- no non-zero solution
- no periodic schedule

(too many tokens accumulate on A->B or B->C)

\[M = \begin{bmatrix}
3 & -1 & 0 \\
0 & 1 & -1 \\
2 & 0 & -1 \\
2 & 0 & -1
\end{bmatrix}\]

**Balance equations**

\[
\begin{bmatrix}
2 & -1 & 0 \\
0 & 1 & -1 \\
2 & 0 & -1 \\
2 & 0 & -1
\end{bmatrix}
\]

\( M = \begin{bmatrix}
2 & -1 & 0 \\
0 & 1 & -1 \\
2 & 0 & -1 \\
2 & 0 & -1
\end{bmatrix}\)

- Non-full rank
  - infinite solutions exist (linear space of dimension 1)
- Any multiple of \( q = [1 \ 2 \ 2]^T \) satisfies the balance equations
- ABCBC and ABBCC are minimal valid schedules
- ABABBCBCCC is non-minimal valid schedule
Static SDF scheduling

- Main SDF scheduling theorem (Lee ’86):
  - A connected SDF graph with \( n \) actors has a periodic schedule iff its topology matrix \( M \) has rank \( n-1 \)
  - If \( M \) has rank \( n-1 \) then there exists a unique smallest integer solution \( q \) to \( Mq = 0 \)
  - Rank must be at least \( n-1 \) because we need at least \( n-1 \) edges (connected-ness), providing each a linearly independent row
  - Admissibility is not guaranteed, and depends on initial tokens on cycles

Admissibility of schedules

- No admissible schedule:
  BACBA, then deadlock...
- Adding one token (delay) on A->C makes BACBACBA valid
- Making a periodic schedule admissible is always possible, but changes specification...
Admissibility of schedules

- Adding initial token changes FIR order

From repetition vector to schedule

- Repeatedly schedule fireable actors up to number of times in repetition vector
  \[ q = [1 \ 2 \ 2]^T \]

- Can find either ABCBC or ABBCC
- If deadlock before original state, no valid schedule exists (Lee ‘86)
From schedule to implementation

◆ Static scheduling used for:
  • behavioral simulation of DF (extremely efficient)
  • code generation for DSP
  • HW synthesis (Cathedral by IMEC, Lager by UCB, …)

◆ Issues in code generation
  • execution speed (pipelining, vectorization)
  • code size minimization
  • data memory size minimization (allocation to FIFOs)
  • processor or functional unit allocation

Compilation optimization

◆ Assumption: code stitching
  (chaining custom code for each actor)

◆ More efficient than C compiler for DSP

◆ Comparable to hand-coding in some cases

◆ Explicit parallelism, no artificial control dependencies

◆ Main problem: memory and processor/FU allocation depends on scheduling, and vice-versa
**Code size minimization**

- **Assumptions (based on DSP architecture):**
  - subroutine calls expensive
  - fixed iteration loops are cheap
    ("zero-overhead loops")
- **Absolute optimum: single appearance schedule**
  - e.g. ABCBC → A (2BC), ABBCC → A (2B) (2C)
    - may or may not exist for an SDF graph...
    - buffer minimization relative to single appearance schedules
      (Bhattacharyya '94, Lauwereins '96, Murthy '97)

**Buffer size minimization**

- **Assumption: no buffer sharing**
- **Example:**

  ![Graph diagram](image)

  - q = | 100 100 10 1 |^T
- **Valid SAS: (100 A) (100 B) (10 C) D**
  - requires 210 units of buffer area
- **Better (factored) SAS: (10 (10 A) (10 B) C) D**
  - requires 30 units of buffer areas, but...
  - requires 21 loop initiations per period (instead of 3)
**Dynamic scheduling of DF**

- SDF is limited in modeling power
  - no run-time choice
    - cannot implement Gaussian elimination with pivoting

- More general DF is too powerful
  - non-Static DF is Turing-complete (Buck ‘93)
    - bounded-memory scheduling is not always possible

- BDF: semi-static scheduling of special “patterns”
  - if-then-else
  - repeat-until, do-while

- General case: thread-based dynamic scheduling
  (Parks ‘96: may not terminate, but never fails if feasible)

---

**Example of Boolean DF**

- Compute absolute value of average of $n$ samples

---
### Example of general DF

- Merge streams of multiples of 2 and 3 in order (removing duplicates)

```
a = get (A)
b = get (B)
forever {
  if (a > b) {
    put (O, a)
a = get (A)
  } else if (a < b) {
    put (O, b)b = get (B)
  } else {
    put (O, a)
a = get (A)
b = get (B)
  }
}
```

- Deterministic merge
  (no “peeking”)

### Summary of DF networks

#### Advantages:

- Easy to use (graphical languages)
- Powerful algorithms for
  - verification (fast behavioral simulation)
  - synthesis (scheduling and allocation)
- Explicit concurrency

#### Disadvantages:

- Efficient synthesis only for restricted models
  - (no input or output choice)
- Cannot describe reactive control (blocking read)