What are Asynchronous Circuits?

- They are circuits that are not synchronous!

- Synchronous: Circuits that use a clock to separate consecutive system states from one another.

- Asynchronous: Circuits that define states in terms of input values and internal actions

Another Definition

- Synchronous: Time Domain
  - Assert signals at a specific time, and for a specific duration

- Asynchronous: Sequence Domain
  - Assert signals after some event, and retain until some other event

What Are They Good For?

- Top Ten List (From Al Davis, Async94)

  Asynchronous Advantages, Often Cited:

  -
  -
  -
**Top Ten List - Async Advantages**

1: Achieve Average Case Performance
   - Exploit data-dependent processing times
   - Best if difference between average and worst case is large
   - Be careful not to spend too much time on completion detection

2: Consume power only when needed
   - CMOS, in particular, consumes power only during transitions
   - Clocks make a lot of transitions, not all of them do useful work
   - Demonstrated ability for async circuits to consume power only on demand

3: Provide easy modular composition
   - LEGO™ approach
   - Allows incremental improvement
   - Object-oriented approach to hardware
   - Operating parameter robustness

4: Do not require clock alignment at interfaces
   - Synchronizing an incoming signal to a clock requires great care, and wastes time
   - Metastability can cause hard-to-find errors
   - Naturally adaptive to a variety of data rates

5: Metastability has time to resolve
   - Any bistable device can get caught in a metastable region for an unpredictable amount of time
   - Assuming fixed resolution time leaves possibility of errors
   - Arbiters can be used to ensure correctness

6: Avoid clock distribution problems
   - Major design time drain
   - Major power budget drain
   - Major chip area drain

7: Exploit concurrency more gracefully
   - Natural way to describe systems with lots of concurrency
   - Let concurrency happen rather than plan all interleavings
Top Ten List - Async Advantages

8: Provide intellectual Challenge
   • Lots of good puzzles
   • Informal reasoning is dangerous
   • Room for innovation
9: Exhibit intrinsic elegance
   • Provide direct mapping of sequence domain
   • Tangible target for theoretical work
   • Correct-by-construction design
   • Measurement vs. trust

Top Ten List - Async Advantages

10: Global synchrony does not exist anyway!
   • High clock speeds, large chips, and even larger systems
   • Global synchrony is a useful abstraction, but it’s not reality
   • May as well admit it, and figure out where async techniques can help solve problems

A Trio of Taxonomies

- Timing Models
- Signaling Protocols
- System Specification and Structure

Taxonomy #1: Timing Models

- Bounded Delays
  • Similar to synchronous circuits
  • Measure maximum delay of each circuit piece, or assume a range of delays
  • Model with extra delay (if required)

```
  Inputs                  Outputs
     >                   >
     >                   >
   Req       Delay
     >                   >
   Logic

  Ack
```
More Timing Models

- Speed Independent Circuits
  - Arbitrary delays in gates
  - Wires have no delay

- Delay Insensitive
  - Arbitrary delays on gates and wires
  - Very appealing model, but the class of circuits for which this really holds is small

Even More Timing Models

- Quasi Delay Insensitive
  - Delay insensitive, but with *isochronous forks*
  - Delay in isochronous forks assumed to be similar
  - In practice, very close to Speed Independent

Combination of Timing Models

- In practice, SI or DI within a range of delay possibilities seems useful

- Careful design of circuit modules can allow DI assumption at interface, SI, qDI, or Bounded Delay inside modules

- Use Bounded Delay for data path, some other model for control (i.e. Bundled Data)

Taxonomy #2: Signaling Protocols

- Layer a protocol on top of signal transitions
- Request/Acknowledge is a popular structure
- Usually referred to as *Self-Timed*
**Control Signaling**

- Four-Phase / Return to Zero / Level Signaling
- Specific protocol determines data release point

---

**Data Signaling**

- Bundled Data
  - “Normal” data wires, one per bit
  - Associated control that signals validity of data

---

**Control Signaling**

- Two-Phase / Non-Return to Zero / Transition Signaling

---

**Data Signaling**

- Dual-Rail Data
  - Two wires per bit, encoded to show validity
  - 00 = no data, 01 = 0, 10 = 1, 11 = error (4Φ)
  - Single acknowledge control wire
### Possible Combinations

<table>
<thead>
<tr>
<th>Bundled</th>
<th>Dual-Rail</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Four-Phase</strong></td>
<td>Amulet 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Two-Phase</strong></td>
<td>Micropipelines</td>
</tr>
<tr>
<td></td>
<td>Amulet 1</td>
</tr>
</tbody>
</table>

### Taxonomy #3: System Specification and Structure

- **Finite State Machine**
- Petri-Net Based
- Macromodules
- Syntax-Directed Program Translation

### Finite State Machines

- Classical asynchronous technique
- Huffman-style state machine

### Asynchronous FSM Models (AFSM)

- **Fundamental mode operation**
  - After an input change, AFSM must settle into new stable state before the next input change
  - Similar to setup and hold restrictions in synchronous machines

- **Different conditions on input changes**
  - Single input change (SIC)
  - Multiple input change (MIC)
  - Unrestricted input change (UIC)
AFSM Design Method

- Generate primitive flow table
- Minimize states => reduced flow table
- Do state assignment
- Generate logic
  - Logic must be hazard-free for every input transition (under some input model and timing model)

Extending AFSM to Burst Mode

- Problem: SIC is too slow, MIC is too hard, UIC is much too hard.
  - SIC forces too much sequencing

- Solution: Operate in fundamental mode, but on bursts of inputs rather than single inputs
  - Allow inputs to change in any order inside the burst
  - A burst of outputs may be required
  - Outputs must be allowed to settle before another input burst is allowed

Burst Mode AFSM

Burst Mode Properties

- Bursts follow a set of rules:
  - Inputs in burst may arrive in any order and at arbitrary times
  - Each state has a unique entry point
  - No input burst may be a subset of another in a given state

- Various techniques to build the circuits: locally clocked, unclocked, 3-D machines
**Extended Burst Mode**

Add a couple of features to burst mode specifications:

- Directed don’t-cares
  - Input edges that may or may not occur
  - Terminated by a definitive transition

- Level condition signals
  - Level signals that are sampled to determine state change direction
  - Must be distinct from transition signals

---

**Taxonomy #3: System Specification and Structure**

- Finite State Machine
- *Petri-Net Based*
- Macromodules
- Syntax-Directed Program Translation

---

**Extended Burst Mode Notation**

---

**Techniques Based on Petri Nets**

- These techniques are based on signal transitions (i.e. traces), rather than on system states

- Traces are a representation of the interface behavior of a circuit

- Petri-net based methods includes I-nets, Signal Transition Graphs (STG), Change Diagrams, Commands, etc...
**I-nets (Interface Nets)**

- Petri nets with transitions labeled with interface signal names

![Diagram of I-nets](image)

**I-net Descriptions**

- The I-net describes the allowed interface behavior
- Note that it may impose restrictions on the environment
- Can be translated into a state machine for implementation (I-net => Interface State Graph (ISG))

**ISG Model**

- Execute the I-net to generate ISG

![Diagram of ISG Model](image)

**Encoded ISG (EISG)**

- Encode states of the ISG

![Diagram of EISG](image)

State Vector is ABX
I-net Generality

- Very general specification, can describe choice

![Diagram of I-net](image)

Signal Transition Graphs

Like I-nets, STGs focus on interface signal transitions, but:

- Use graph theory to reason about properties represented in the STG
- Restrict the allowable forms of STGs such that they are implementable
- Signal transitions are annotated with directions

Example STG

- Underlined signals are inputs
- Balls are like Petri-net tokens

![Diagram of STG](image)

Properties of STGs

- Can be checked for liveness (deadlock free), persistency (hazard free)
- Requires only syntactic check of the STG
- A live, persistent STG can be implemented as a speed-independent circuit
- STGs have been extended to handle various forms of choice, and to check for many more properties that influence behavior and circuits
Taxonomy #3: System Specification and Structure

- Finite State Machine
- Petri-Net Based
- Macromodules
- Syntax-Directed Program Translation

Macromodules

- Like standard cells (the LEGO™ approach)
- A set of building blocks that can be assembled into asynchronous systems
- Usually considered delay-insensitive at the module interface
- Module internals designed using one of the previously defined techniques

Sutherland’s Micropipelines

- Popular macromodule library
- Two-phase transition signaling

Basic Micropipeline

- Looks like a FIFO with processing
More Complicated Circuits

- Branching and Merging FIFOs

![FIFO Toggle Branch Diagram]

FIFO Toggle Branch

- IN[7:0]
- ROUT0
- AOUT0
- AOUT1
- ROUT1
- OUT[7:0]

FIFO Toggle Merge

Taxonomy #3: System Specification and Structure

- Finite State Machine
- Petri-Net Based
- Macromodules
- Syntax-Directed Program Translation
Syntax Directed Program Translation

- Start with program description
- Translate to asynchronous circuit automatically
- Program notation should be capable of describing concurrency
- Basic communication structure is a channel
- CSP, OCCAM, and Tangram are popular starting points

Example: FIFO Buffer

```
(PROC buffer ((CHAN A B)) ; define a buffer
 (WHILE TRUE ; repeat forever
  (SEQ ((VAR temp<8>)) ; do in sequence
   (? A temp) ; input from A
   (! B temp)))) ; output to B

; define a parallel composition of two buffers

(PAR ((CHAN input mid output))
 (buffer input mid)
 (buffer mid output))
```

Overview of One Method

- Write an OCCAM program
- Use syntax-directed translation from that program to a collection of macromodules
- Perform peephole optimization of that circuit
- Place and route resulting circuit

FIFO Buffer Translation 1

```
Start

True

Req

Ack

A Channel

Body of WHILE is a Sequential construct

Seq

While

Ack

B Channel
```
FIFO Buffer Translation 2

Body processes of SEQ are Input and Output Primitives.

FIFO Buffer Translation 3

FIFO Buffer Translation 4

Peephole Optimization

- Identify circuit structure that can be improved
- Substitute "better" circuit
Recall the Program Text

(PROC buffer ((CHAN A B)) ; define a buffer
  (WHILE TRUE ; repeat forever
    (SEQ ((VAR temp<8>)) ; do in sequence
      (? A temp) ; input from A
      (! B temp))) ; output to B
  ; define a parallel composition of two buffers

(PAR ((CHAN input mid output))
  (buffer input mid)
  (buffer mid output))

Parallel Composition
Program Transformation

- Higher level description
- Correct by compilation circuits
- Lots of variations on this scheme
  - Different specification languages
  - Different libraries
  - Different compilation strategies

More Conclusions

- Asynchronous circuits and systems seem to have some compelling advantages
- So far, most are just potential, not demonstrated
- Don’t be misled by the Top Ten List. There are plenty of significant problems left to solve!

Conclusions (at last!)

- Asynchronous circuits research area is becoming too large to talk about in an hour!
- Topics that I have not talked about
  - Formal methods
  - Verification
  - Testing
  - Circuit techniques
  - Arbitration
  - Synchronous-Asynchronous interfacing
  - Datapath design
  - Low power circuits
  - Large case studies and landmark results

Required Reading List

- C. Seitz. System Timing, Chapter 7 of Introduction to VLSI Systems by Mead and Conway, 1980
- I. Sutherland. Micropipelines, Communications of the ACM, June 1989
Supplemental Reading List

- Finite State Machines
  - S. M. Nowick and D. L. Dill, Automated synthesis of locally-clocked asynchronous state machines, *ICCAD-91*
  - S. M. Nowick and D. L. Dill, Exact two-level minimization of hazard-free logic with multiple-input changes, *ICCAD-92*
  - K. Y. Yun and D. L. Dill, Unifying synchronous/asynchronous state machine synthesis, *ICCAD-93*

- Petri-Net Based Approaches
  - C. Molnar, T. Fang, and F. Rosenberger, Synthesis of delay-insensitive modules. In *Chapel Hill Conference on VLSI*, 1985

- Macromodules

- Program Translation Approaches
  - K. van Berkel and R. Saeijs. Compilation of communicating processes into delay-insensitive circuits. *ICCD-88*
  - E. Brunvand and R. Sproull. Translating concurrent programs into delay-insensitive circuits, *ICCAD-89*
Further Pointers

- This list only scratches the surface. For further pointers, try: