Week 2: Gate-level design
<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 AM</td>
<td>Recap of models; handshake protocols</td>
</tr>
<tr>
<td>9:50 AM</td>
<td><em>break</em></td>
</tr>
<tr>
<td>10:00 AM</td>
<td>Gates and gate-level simulation</td>
</tr>
<tr>
<td>10:25 AM</td>
<td><em>break</em></td>
</tr>
<tr>
<td>10:30 AM</td>
<td>Pipeline example</td>
</tr>
<tr>
<td>11:00 AM</td>
<td><em>break</em></td>
</tr>
<tr>
<td>11:10 AM</td>
<td>Syntax-directed translation to cells</td>
</tr>
<tr>
<td>12:15 PM</td>
<td>Non-determinism</td>
</tr>
<tr>
<td>12:45 PM</td>
<td>Q&amp;A</td>
</tr>
<tr>
<td>1:00 PM</td>
<td>End of Day 2</td>
</tr>
</tbody>
</table>
Recap of previous week: message-passing abstraction

```plaintext
defproc buffer(chan?(int) L; chan!(int) R)
{
  int x;  // local state

  chp {
    *[ L?x; R!x ]
  }
}

defproc gcd(chan?(int) X, Y; chan!(int) O)
{
  int x, y;  // local state

  chp {
    *[ X?x, Y?y;
        *[ x > y -> x := x - y
        ][ y > x -> y := y - x
    ];
    O!x
  }
}

defproc alu(chan?(int<2>) cmd; chan?(int) X, Y; chan!(int) O)
{
  int x, y;  // local state
  int<2> c;

  chp {
    *[ X?x, Y?y, cmd?c;
        [c=0 -> O!(x + y)
        ][c=1 -> O!(x - y)
        ][c=2 -> O!(x & y)
        ][c=3 -> O!(x | y)
    ]
  }
}

CHP = Communicating Hardware Processes
```
Recap of previous week: dataflow abstraction

- Dataflow model of computation
  - Tokens flowing through pipelines
  - Each component operates in parallel
- Each dataflow component can be written as a CHP program
From message-passing to signals

- Variables
  - Integers can be implemented as an array of signals (bool)

- What about channels?

```plaintext
defproc alu(chan?(int<2>) cmd;
  chan?(int) X, Y; chan!(int) O)
{
  int x, y; // local state
  int<2> c;

  chp {
    *[ X?x, Y?y, cmd?c;
      [c=0] -> O!(x + y)
      [c=1] -> O!(x - y)
      [c=2] -> O!(x & y)
      [c=3] -> O!(x | y)
    ]
  }
}
```
Channels can be implemented in a number of ways:

- Each channel requires a set of wires.
- Sender and receiver must follow a communication protocol.

channel

signals / wires

[ ]
Dataless channels: two-phase protocol

- Two-phase protocol
  - Also called transition signaling protocol
  - Two wires to implement channel
  - Two signal changes ("phases") in sequence
- One end of the channel **initiates** the communication
  - Called the “**active**” end of the channel (other end is **passive**)

Either end of the channel can initiate the communication!
Dataless channels: four-phase protocol

- Four-phase protocol
  - Two wires to implement channel
  - Four signal changes ("phases") in sequence
  - Sometimes called "return to zero" protocol
- One end of the channel **initiates** the communication
  - Called the "**active**" end of the channel (other end is **passive**)

Either end of the channel can initiate the communication!
Dataless channels: single track protocol

- Two-phase protocol
  - One wire to implement channel
  - Two signal changes ("phases") in sequence
- One end of the channel **initiates** the communication
  - Called the "active" end of the channel (other end is **passive**)

Either end of the channel can initiate the communication!
Encoding data: bundled data communication

- Protocol on request/acknowledge protocol can be *any of the ones seen earlier!*
  - Two wires (or one) for the control
  - N data wires for N-bit data communication
  - **Timing** requirement (“bundled data timing requirement”)
Encoding data: bundled data communication

- Protocol on request/acknowledge protocol can be *any of the ones seen earlier!*
  - Two wires (or one) for the control
  - N data wires for N-bit data communication
  - **Timing** requirement (“bundled data timing requirement”)
Encoding data: delay-insensitive encoding

- Four-phase communication with dual-rail data encoding
  - Two wires for one bit
  - Four-phase handshake on (data 0, acknowledge) or (data 1, acknowledge)
Delay-insensitive encoding

- 1-of-N encoding
  - $N$ wires to send $\log(N)$ bits of information
  - Common choices: $N=2$ or $N=4$

- k-of-N encoding
  - Maximum value occurs for $k = \text{floor}(N/2)$
    - Extra wires: $\sim O(\log(N))$
    - These are called Sperner codes

- Mix-and-match
  - $N/2$ copies of a 1-of-4 code ($2N$ wires for $N$ bits)
How do I know that data has arrived?

- 1-of-N encoding
  - OR gate

- k-of-N encoding
  - ... a bit more complicated!

How do I check all bits have arrived?
- Check each individual code
- Combine checks using a completion tree

Standard gate: C-element

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>hold state</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>hold state</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multi-bit delay-insensitive communication

- In this example, the “request” is **embedded** in the data encoding
  - Data bits are valid is interpreted as a **phase** in the 2-phase/4-phase communication
    - Replaces request going high (or acknowledge going high), for example

---

Replace with wire encoding!
Encoding data: two-phase delay-insensitive encoding

- Two popular approaches
  - Toggle data wire to send the appropriate bit
  - Four-state encoding (popularly called level-encoded dual rail or LEDR)
    - One of the wires is the data bit
    - The second wire is toggled when next data bit is unchanged
Channels in ACT

- Example
  - Bundled-data four-phase channels
  - Defined in the ACT standard library

```cpp
import std::channel;

/* This defines std::channel::bd<M> as an implementation of chan(int<M>) */
```

defproc alu(chan?(int<2>) cmd;
    chan?(int) X, Y; chan!(int) O)
{
    int x, y;  // local state
    int<2> c;

    chp {
        *[ X?x, Y?y, cmd?c;
            [c=0 -> O!(x + y)
            []c=1 -> O!(x - y)
            []c=2 -> O!(x & y)
            []c=3 -> O!(x | y)
        ]
    }
}
```