Custom async circuit design from production rules to netlist

Benjamin Hill
benjamin.hill@intel.com

ASYNC Summer School 2022
Legal Information

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Your costs and results may vary.

Results have been estimated or simulated

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
Implementing custom circuit designs

Our task:
Given a gate-level system expressed as production rule set (PRS), generate netlist (SPICE) and physical implementation (layout)

Do as little full custom design as possible!

Stay tuned for later sessions describing more automated parts of the Yale physical implementation flow
Production rule basics

\[ \sim a \& \sim b \rightarrow \text{out}^+ \]

\[ a \& b \rightarrow \text{out}^- \]
Production rule sets form gates

\( \sim \text{in} \rightarrow \text{out}^+ \)
\( \text{in} \rightarrow \text{out}^- \)
Custom design flow

- **.act file (text entry)**
  - prs2sim

- **.sim and .al files**
  - digital simulation with irsim, cosmos

- **.prs file**
  - digital simulation, hazard identification using prsim

- **pre-layout .spice file**
  - analog simulation with xyce, ngspice

- **switch-level simulation report**
  - gate-level async simulation report

- **analog simulation report**

**Existing open-source tools:**
- toolname: ACT tools

https://avlsi.csl.yale.edu/act/doku.php?id=custom:start
CMOS transistors
**CMOS transistors**

- **PMOS**
  - Conducts current between source and drain (through P-type channel) when gate is **LOW**

- **NMOS**
  - Conducts current between source and drain (through N-type channel) when gate is **HIGH**

**Gate**

- **Source** ___________ **Drain**

**Switch**

- Voltage-controlled switch
PMOS

NMOS

gate

source \_\_\_\_\_\_ drain

drain \_\_\_\_\_\_ source

cross section

schematic symbol

layout (top down view)

p substrate

n well

p substrate

n well
CMOS transistors

PMOS

NMOS
Transistor operation: cutoff

PMOS
- G = 1
- S = 1
- D = 1
- B = 1

NMOS
- G = 0
- D = 0
- S = 0
Transistor operation: channel formation

PMOS

NMOS
Transistor operation: saturation

PMOS

\( G=0 \)

\( S=1 \)  \( D=0 \)  \( B=1 \)  \( p^+ \)

\( n \) well

Current flows through channel

NMOS

\( G=1 \)

\( D=1 \)  \( S=0 \)  \( B=0 \)  \( n^+ \)

\( p \) substrate
Transistor operation: cutoff

PMOS

G=1
S=1
D=0
B=1

no channel
no current

NMOS

G=0
S=0
D=1
B=0
CMOS transistors

PMOS

conducts current between source and drain (through P-type channel) when gate is LOW

gate

source __________ drain

NMOS

conducts current between source and drain (through N-type channel) when gate is HIGH

gate

drain __________ source

voltage-controlled switch
Layout and sizing
Layout example: inverter

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Design rules

set of geometric restrictions intended to yield high probability of correct fabrication, operation, and lifetime

Transistor sizing

Skywater 130 simplified design rules

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>75 nm*</td>
</tr>
<tr>
<td>$L$</td>
<td>$2 \lambda$</td>
</tr>
<tr>
<td>$W_N$</td>
<td>$6 \lambda$</td>
</tr>
<tr>
<td>$W_P$</td>
<td>$10 \lambda$</td>
</tr>
</tbody>
</table>

* minimum feature size for Skywater 130 is 150nm transistor gate length

https://github.com/asyncvlsi/sky130l
Sizing – how and why

• W, L, lambda
• ACT sizing body, prs body, default config
• Gates drive capacitance: think of simple R-C circuit
• Switching burns energy: charge and discharge capacitor
• Always trading off energy/area/performance
Transistor performance scaling intuition

Transistor device with width and length parameters

What happens as we vary them?
Analogy: resistors

Current through resistor (aka “drive strength”) is inversely proportional to effective resistance:

\[ I = \frac{V}{R} \]

- Decreased resistance, increased current

\[ R_{\text{parallel}} = \frac{1}{R_1} + \frac{1}{R_2} = \frac{R}{2} \]

\[ R_{\text{series}} = R_1 + R_2 = 2R \]
Transistor performance scaling intuition

Increasing transistor \( W \) increases drive strength.

Key performance metric: current through transistor, effective resistance, “drive strength.”

Increasing gate \( L \) decreases drive strength.

Channel cross-section
Digital designer summary

Use NMOS in pull-down network, PMOS in pull-up
⇒ single stage logic is always inverting

Transistor drive strength $\propto \frac{Width}{Length}$
⇒ use minimum gate length for digital logic (usually)
State-holding gates
Combinational vs state-holding gates

**Combinational**
- Either **UP** or **DOWN** (but not both) is always conducting

**State-holding**
- At times neither **UP** nor **DOWN** is conducting
- **out** is undriven and maintains its previous value

**Interfering**
- Both **UP** and **DOWN** conducting simultaneously
- Causes short-circuit/crowbar current through gate, should not be more than transient
Example state-holding gate: Muller C-element

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>hold previous state</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>hold previous state</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

not CMOS implementable!
inverting C-element plus inverter

shorthand syntax
Problem: undriven dynamic nodes

Solution: staticizers
C-element PRS to SPICE example

```plaintext
defproc celem (bool? a,b; bool! out)
{
  bool _out;
  prs {
    a & b => _out-
    _out => out-
  }
}
```
Define new subcircuit (cell):
\[ \text{.subckt name ports} \]

Comments begin with *

Metadata generated by prs2net

MOSFET instances:
\[ M\text{name} D G S B \text{ type <param=val>} \]

End of inv subcircuit:

Instantiate subcircuits hierarchically:
\[ x\text{name} \text{ ports cellname} \]

*---- act defproc: inv<> -----
* raw ports: in out

\[ \text{.subckt inv in out} \]
\[ \text{.PININFO in:I out:O} \]
\[ \text{.POWER VDD Vdd} \]
\[ \text{.POWER GND GND} \]
\[ \text{.POWER NSUB GND} \]
\[ \text{.POWER PSUB Vdd} \]
* --- node flags ---
* out (combinational)
* --- end node flags ---
M0_ Vdd in out Vdd p W=1.5U L=0.6U
M1_ GND in out GND n W=0.9U L=0.6U

.ends

*---- end of process: inv<> -----

*---- act defproc: buf<> -----
* raw ports: in out

\[ \text{.subckt buf in out} \]
\[ x\text{stage1 in __out inv} \]
\[ x\text{stage2 __out out inv} \]
.ends

*---- end of process: buf<> -----
C-element PRS to SPICE example
C-element with weak keeper staticizer

```
defproc celem (bool? a,b; bool! out)
{
    bool _out;
    prs {
        a & b => _out-
        _out => out-
    }
}
```
C-element with combinational feedback

```bash
defproc celem_comb (bool? a, b; bool! out)
{
    bool _out;
    prs {
        [comb=1] a & b => _out-
        _out => out-
    }
}
```
van Berkel C-element

```c
defproc celem_H (bool? a,b; bool! out)
{
    bool _out;
    bool nmid[2], pmid[2];
    prs {
        // N-stack
        [keeper=0] a -> nmid[0]-
        [keeper=0] b -> nmid[1]-
        passn (out, nmid[0], nmid[0])
        passn (b, nmid[0], _out)
        passn (a, nmid[1], _out)

        // Symmetric P-stack, out inverter
        ...
    }
}
```
## Simulation options

<table>
<thead>
<tr>
<th></th>
<th>Gate level</th>
<th>Switch level</th>
<th>Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>simulator</td>
<td>prsim, actsim</td>
<td>irsim</td>
<td>Xyce</td>
</tr>
<tr>
<td>input</td>
<td>ACT PRS</td>
<td>.sim</td>
<td>SPICE</td>
</tr>
<tr>
<td>to generate</td>
<td>write directly or use e.g. chp2prs</td>
<td>prs2sim</td>
<td>prs2net</td>
</tr>
<tr>
<td>model</td>
<td>unit delay</td>
<td>RC delay</td>
<td>full analog</td>
</tr>
<tr>
<td>fidelity</td>
<td>lowest</td>
<td>medium</td>
<td>highest</td>
</tr>
<tr>
<td>speed</td>
<td>fastest</td>
<td>fast</td>
<td>slow</td>
</tr>
</tbody>
</table>
**Full custom flow example**

**Design specification**

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

CHP, dataflow, link-joint, etc

**Production rules**

```plaintext
defproc inv (bool? in; bool! out)
{
    prs { in => out- }
}
```

**Netlist (SPICE)**

```plaintext
*---- act defproc: inv<> -----
.subckt inv in out
*.PININFO in:I out:O
*.POWER VDD Vdd
*.POWER GND GND
*.POWER NSUB GND
*.POWER PSUB Vdd
* --- node flags ---
* out (combinational)
M0_ Vdd in out Vdd p W=1.5U L=0.6U
M1_ GND in out GND n W=0.9U L=0.6U
.ends
*---- end of process: inv<> -----
```

**Schematic**

**Layout**

write or compile

prs2net

prs2sim

full custom or semi-automated

extracted parasitics