H5H4, H5E7 – lecture 5
Synchronous Data Flow Graphs
Control Flow Models

I. Verbauwhede

Acknowledgement: H. De Man, P. Schaumont

2009

K.U.Leuven

Last class

- Presentation by Mr. R. Valvekens
  - Main message(s)?
- Models of computation
- Many representations of time:
  - Total order
  - Partial order
- For our purposes:
  - Synchronous data flow graphs
  - Control flow graphs
- Continue on this today

Today

- Control flow graphs
  - FSM
  - FSMD
  - Completeness
  - Hierarchy
  - Communication
  - Implementation issues

Reading

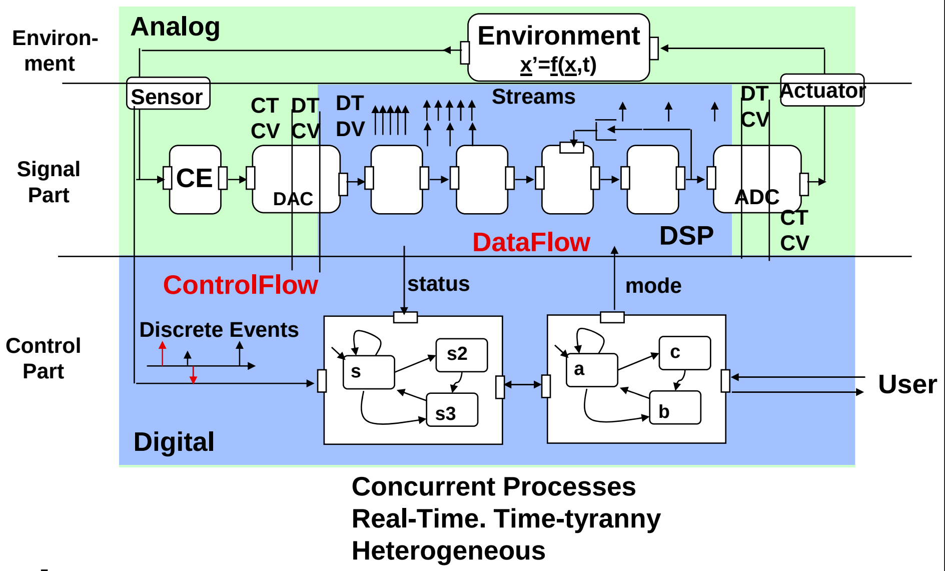
- P. Schaumont, GEZEL user's manual, http://rijndael.ece.vt.edu/gezel2/

Background reading:

- D. Harel, "StateCharts: A Visual formalism for complex systems," Science of computer programming 8 (1987), pages 231-274, Elsevier North Holland.
  - We will use StateCharts to illustrate control flow concepts in a visual manner.
  - In reality StateCharts is used for complex software systems and not really for hardware design.

- Davio, Deschamps, Thayse, "Digital systems with algorithm implementation," Wiley, 1983.

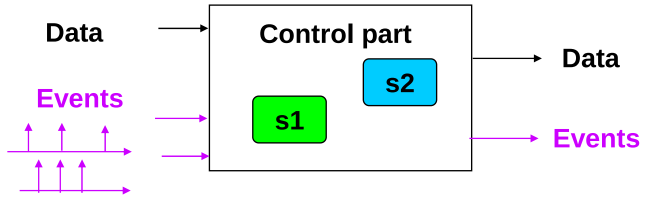
Remember: a typical embedded system



Concurrent Processes
Real-Time. Time-tyranny
Heterogeneous

Control Flow

- Systems reacting to external *discrete events*.
- Upon events enter a certain *mode* and perform some *actions* on data (in *synchronous* way)
- The mode reached depends on a *history* of input events
- The *same history* should produce the *same mode = "deterministic"*
- A mode corresponds to a *state*



•The prototype of a reactive system is a behavioral FSMD.

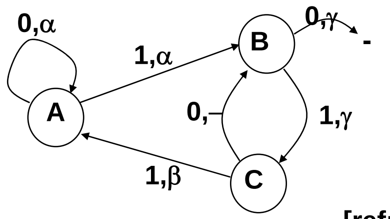
FSM (Finite State Machine)

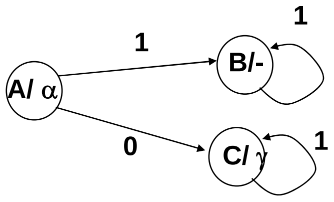
Mealy automaton:

FSM = (I, O, Q, M, N)
With I = input set
O= output set
Q = state set
M = set of functions QxI -> Q
N = set of functions QxI -> O

Moore automaton:

FSM = (I,O,Q,M,N)
I = same
O = same
Q = same
M = same
N = output function, Q -> O





[ref: Davio]

Formal definition

An FSM is a 6-tuple *F*<*S*, *I*, *O*, *F*, *H*, s_0>

- ***S* is a set of all states {s_0, s_1, …, s_l}**
- ***I* is a set of inputs {i_0, i_1, …, i_m}**
- ***O* is a set of outputs {o_0, o_1, …, o_n}**
- ***F* is a next-state function (*S* x *I* → *S*)**
- ***H* is an output function (*S* → *O*)**
- **s_0 is an initial state**

Moore-type

- **Associates outputs with states (as given above, *H* maps *S* → *O*)**

Mealy-type

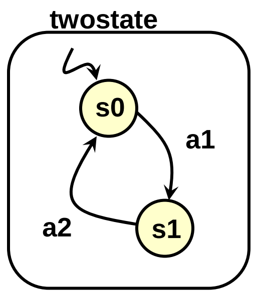
- **Associates outputs with transitions (*H* maps *S* x *I* → *O*)**

Shorthand notations to simplify descriptions

- **Implicitly assign 0 to all unassigned outputs in a state**
- **Implicitly AND every transition condition with clock edge (FSM is synchronous)**

[ref: Davio]

GEZEL: Finite State Machine

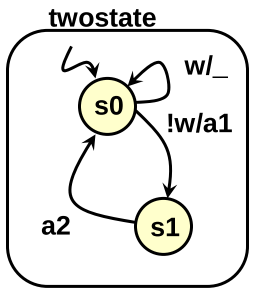


- Initial State: s0
- Normal State: s1
- State Transitions:
  s0 -> s1, s1 -> s0
- Actions: a1, a2

- Static sequencing

```
fsm twostate {
  initial s0;
  state   s1;
  @s0 (a1) -> s1;
  @s1 (a2) -> s0;
}
```

GEZEL: Conditional State Transitions



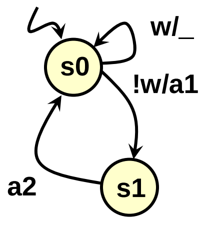
- State transition condition: w
- State transition from s0:
  if w then goto s0 else do a1 goto s1

- Dynamic sequencing
- Conditions need to be logically complete

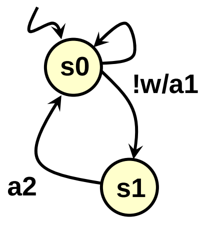
```
fsm twostate {
  initial s0;
  state   s1;
  @s0 if (w) then (idle) -> s0;
             else (a1)   -> s1;
  @s1 (a2) -> s0;
}
```

GEZEL: Conditions must be complete

• For any source state si, exactly 1 state transition needs to be enabled

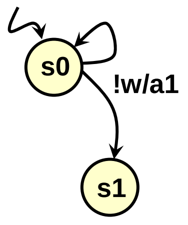


OK



not OK

s0 can have two enabled state transitions



not OK

s1 has never an enabled transition

Disadvantages of Single FSM

- Only in one state at a time
- State explosion in complex systems
- Need for:
  - **Behavioral hierarchy** `(state in state in state...)`
  - **Concurrency of states (prevents explosion)**
  - **Synchronization** `(shared memory, events, status)`
  - **Communication** `(shared memory(semaphores))`
  - **Exceptions** `(preemption,abortion, history=interrupt)`
  - **Programming constructs** `(state=sequential program execution)`
  - **Indeterminism** `(arbitrary choice)`

- Leads to “extended state machine” concept
- *Historical paper: “Statecharts: a visual formalism for complex systems”, Science of Computer programming Vol 8, 1987, pp. 231-275*
- Used in complex software programs

State Charts

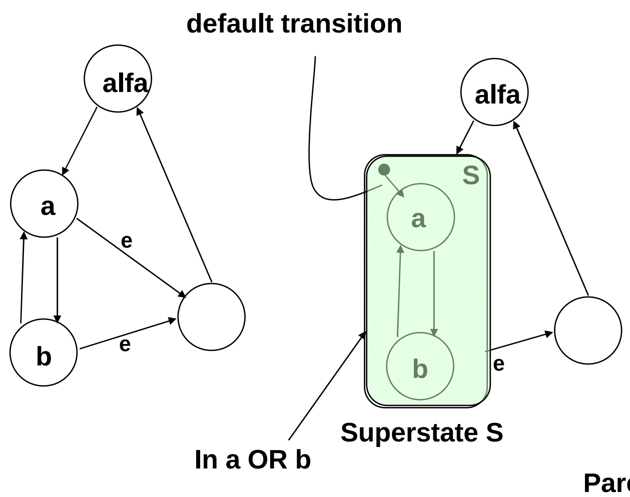
- State Charts is a *visual* programming language for control systems.
- commercial tools such as STATEMATE (I-Logix), ARGOS (Inria-Sofia Antipolis), SPECC (Gajski), STATEFLOW (Matlab).
- The basic concepts in the next slides are using syntax of STATEFLOW.
- Used to illustrate the concepts visually.

Goal State charts

- Cluster states or super states
  - “In all airborne states, when yellow handle is pulled, eject chair
- Add independency or orthogonality
  - “gearbox change of state is independent of braking system
- Allow “general” transitions
  - ‘when selection button is pressed enter selected mode
- Capture refinement
  - “display mode consists of time-display, date-display and stop-watch display.

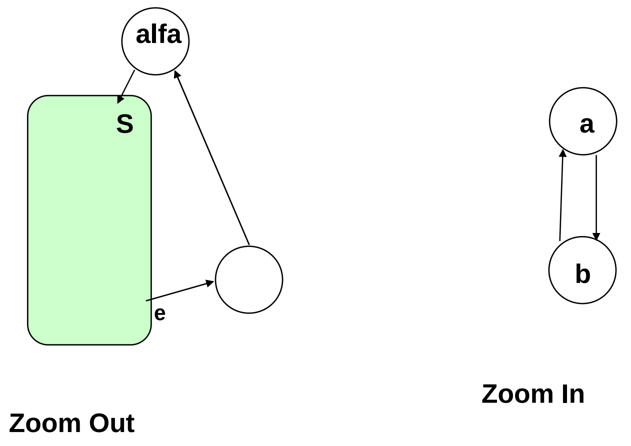
- All of this *visually* represented
- Statecharts = state-transition-diagrams + depth + orthogonality + broadcast-communication

Behavioral Hierarchy

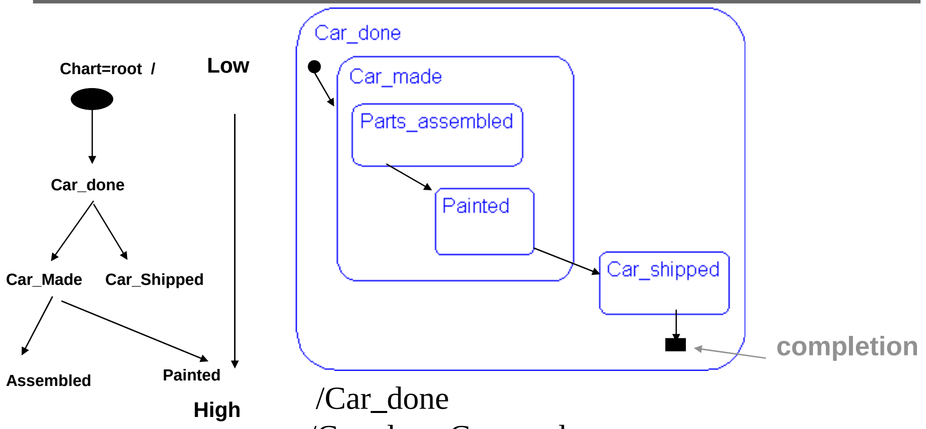


Parent S, activity
child FSM

Zoom In - Zoom Out



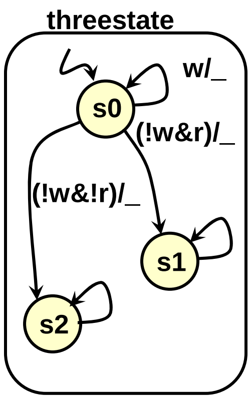
State Hierarchy



Parents
Children

/Car_done
/Car_done.Car_made
/Car_done.Car_shipped
/Car_done.Car_made.Parts_assembled
/Car_done.Car_made.Painted

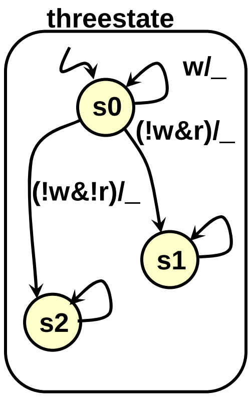
GEZEL: Nested Conditions



- **State transition condition: w and r**
- **Three possible transitions from s0:**
  if (w) then goto s0
  else if (r) then goto s1
  else goto s2
- **w has higher priority then r**

```
fsm threestate {
  initial s0;
  state   s1, s2;
  @s0 if (w) then (idle) -> s0;
      else if (r) then (idle) -> s1;
                  else (idle) -> s2;
  @s1 (idle) -> s1;
  @s2 (idle) -> s2;
}
```

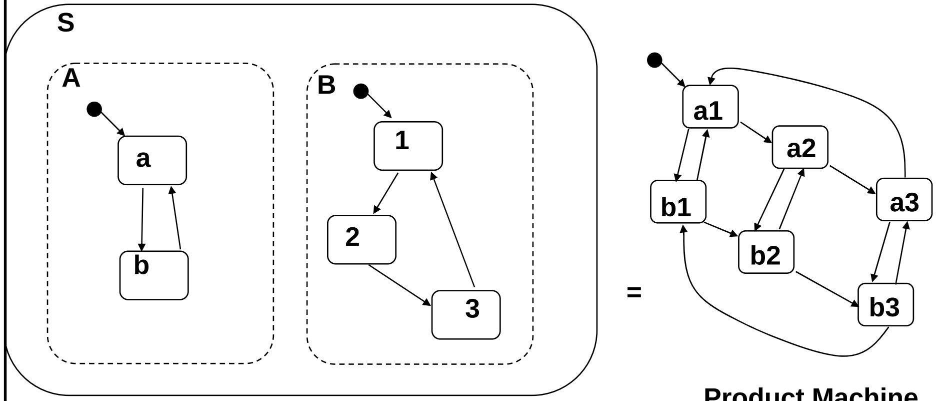
GEZEL: (Un)Nested Conditions



- Explicit priority

```
fsm threestate {
  initial s0;
  state   s1, s2;
  @s0 if (w) then (idle) -> s0;
      else if (~w&r) then (idle)->s1;
      else (idle) -> s2;
  @s1 (idle) -> s1;
  @s2 (idle) -> s2;
}
```

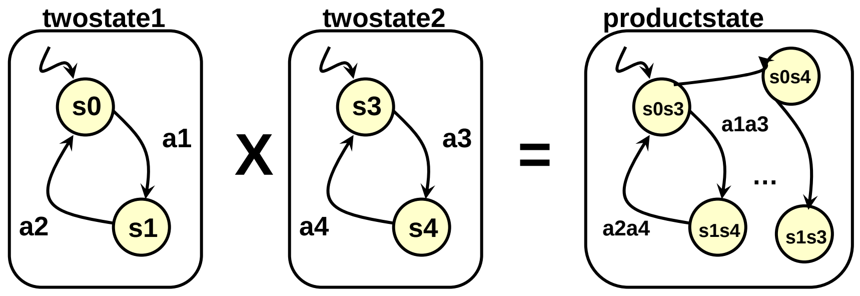
Concurrency



Product Machine
#States = #A*#B

S: superstate, S active-> S.A, S.B BOTH active
A and B entered concurrently-> S.A.a,S.B.1
Prevents state explosion

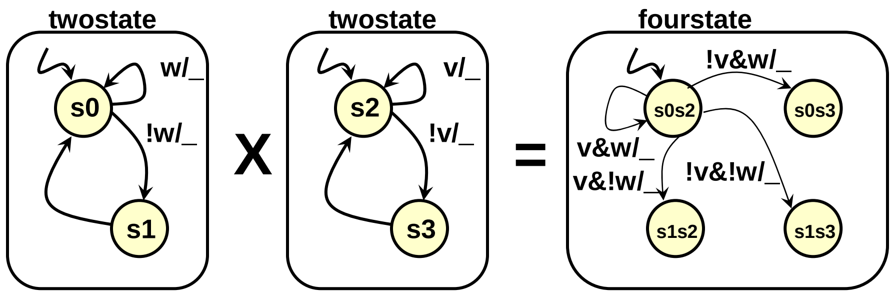
Concurrency



[all combinations covered?]

- Product state machine:
  state space is obtained as the product of composing state machines

Concurrency



(some transitions skipped)

- Product state machine may explode in complexity
  (M-state combined with N-state can lead to M.N states)
- State explosion prevents modeling of many concurrency problems
  as a combined state machine
- Concurrency modeling - express unrelated state machines separately

FSMD

FSMD = (I, O, Q, M, N, V)

With I = input set

O= output set

Q = state set

V = variable set (from data path)

M = set of functions QxIxV -> Q

N = set of functions QxIxV -> O

Finite-state machine with datapath model (FSMD)

FSMD extends FSM: complex data types and variables for storing data

- **FSMs use only Boolean data types and operations, no variables**

FSMD: 7-tuple <*S*, *I* , *O*, <u>*V*</u>, *F*, *H*, s_0>

- ***S* is a set of states {s_0, s_1, …, s_l}**
- ***I* is a set of inputs {i_0, i_1, …, i_m}**
- ***O* is a set of outputs {o_0, o_1, …, o_n}**
- <u>*V* is a set of variables {v_0, v_1, …, v_n}</u>
- ***F* is a next-state function ($S \times I \times V \rightarrow S$)**
- ***H* is an <u>action</u> function ($S \rightarrow O + V$)**
- **s_0 is an initial state**

***I*,*O*,*V* may represent complex data types (i.e., integers, floating point, etc.)**

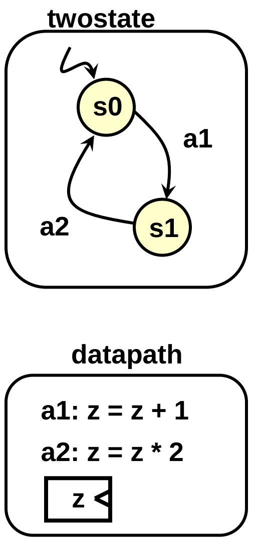
***F*,*H* may include arithmetic operations**

***H* is an action function, not just an output function**

- **Describes variable updates as well as outputs**

Complete system state now consists of current state, s_i, and values of all variables

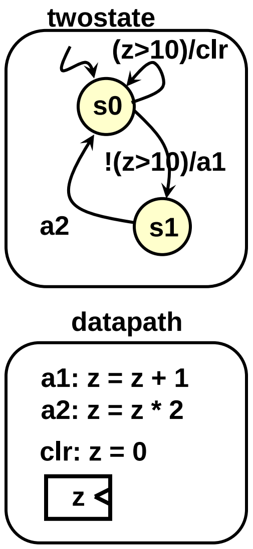
GEZEL: Finite State Machine + Datapath



• **datapath implements actions a1, a2**

```
dp datapath {
  reg z : ns(20);
  sfg a1 {z = z + 1;}
  sfg a2 {z = z << 1;}
}

fsm twostate(datapath) {
  initial s0;
  state   s1;
  @s0 (a1) -> s1;
  @s1 (a2) -> s0;
}
```

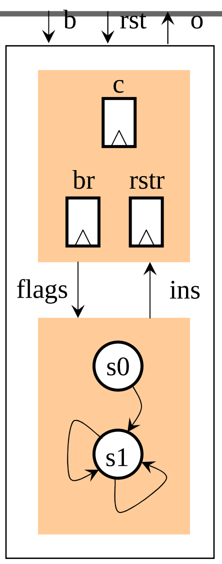
GEZEL: Datapath conditions



• **Datapath implements state transitions conditions**
• **State transition conditions are always directly dependent on registers**

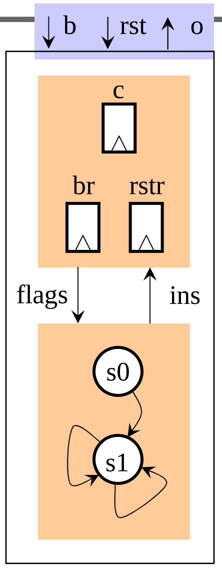
```
dp datapath {
  reg z : ns(20);
  sfg a1  {z = z + 1;}
  sfg a2  {z = z << 1;}
  sfg clr {z = 0;}
}

fsm twostate(datapath) {
  initial s0;
  state   s1;
  @s0 if (z>10) then (clr) -> s0;
      else (a1) -> s1;
  @s1 (a2) -> s0;
}
```

Ones counter FSMD in GEZEL



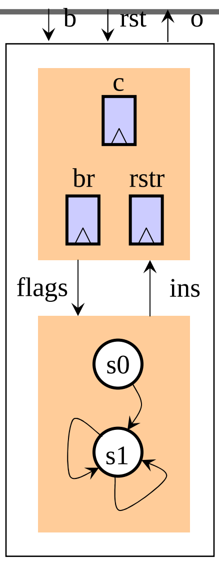
```
dp ones(in b, rst : ns(1); out o : ns(8)) {
  reg c  : ns(8);
  reg br, rstr : ns(1);
  sfg reset { c = 0; o = 0; br = b;
              rstr = 0;}
  sfg inc   { c = c + 1; o = c;
              br = b; rstr = rst; }
  sfg hold  { br = b; rstr = rst;
              o = c;}
}
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) -> s1;
      else if (br) then (inc) -> s1;
      else (hold) -> s1;
}
```

Ones counter FSMD in GEZEL



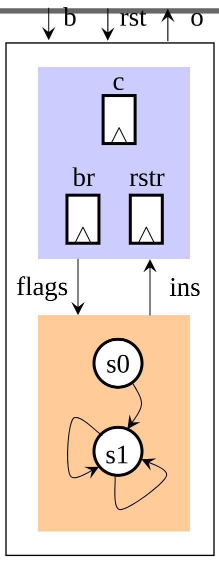
```
dp ones(in b, rst : ns(1); out o : ns(8)) {
  reg c  : ns(8);
  reg br, rstr : ns(1);
  sfg reset { c = 0; o = 0; br = b;
              rstr = 0;}
  sfg inc   { c = c + 1; o = c;
              br = b; rstr = rst; }
  sfg hold  { br = b; rstr = rst;
              o = c;}
}
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) -> s1;
      else if (br) then (inc) -> s1;
      else (hold) -> s1;
}
```

Ones counter FSMD in GEZEL



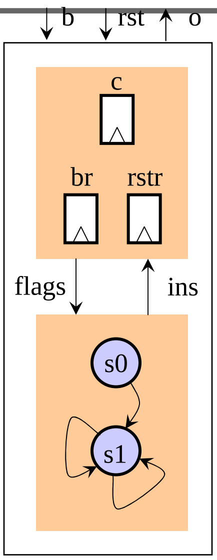
```
dp ones(in b, rst : ns(1); out o : ns(8)) {
  reg c  : ns(8);
  reg br, rstr : ns(1);
  sfg reset { c = 0; o = 0; br = b;
              rstr = 0;}
  sfg inc   { c = c + 1; o = c;
              br = b; rstr = rst; }
  sfg hold  { br = b; rstr = rst;
              o = c;}
}
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) -> s1;
      else if (br) then (inc) -> s1;
      else (hold) -> s1;
}
```

Ones counter FSMD in GEZEL



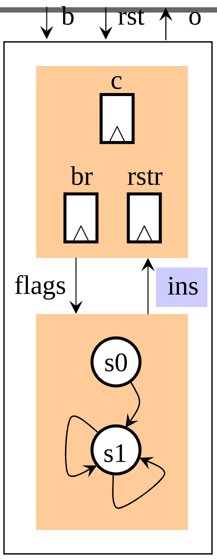
```
dp ones(in b, rst : ns(1); out o : ns(8)) {
  reg c  : ns(8);
  reg br, rstr : ns(1);
  sfg reset { c = 0; o = 0; br = b;
              rstr = 0;}
  sfg inc   { c = c + 1; o = c;
              br = b; rstr = rst; }
  sfg hold  { br = b; rstr = rst;
              o = c;}
}
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) -> s1;
      else if (br) then (inc) -> s1;
      else (hold) -> s1;
}
```

Ones counter FSMD in GEZEL



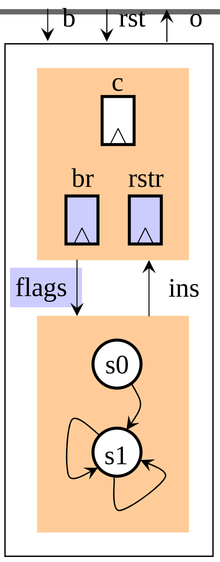
```
dp ones(in b, rst : ns(1); out o : ns(8)) {
  reg c  : ns(8);
  reg br, rstr : ns(1);
  sfg reset { c = 0; o = 0; br = b;
              rstr = 0;}
  sfg inc   { c = c + 1; o = c;
              br = b; rstr = rst; }
  sfg hold  { br = b; rstr = rst;
              o = c;}
}
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) -> s1;
      else if (br) then (inc) -> s1;
      else (hold) -> s1;
}
```

Ones counter FSMD in GEZEL



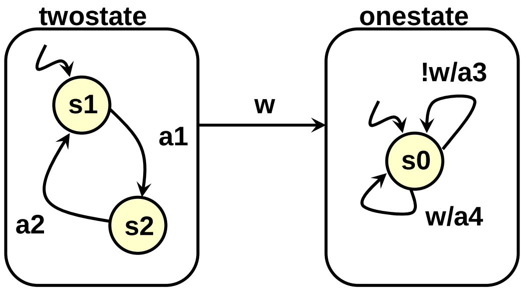
```
dp ones(in b, rst : ns(1); out o : ns(8)) {
  reg c  : ns(8);
  reg br, rstr : ns(1);
  sfg reset { c = 0; o = 0; br = b;
              rstr = 0;}
  sfg inc   { c = c + 1; o = c;
              br = b; rstr = rst; }
  sfg hold  { br = b; rstr = rst;
              o = c;}
}
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) -> s1;
      else if (br) then (inc) -> s1;
      else (hold) -> s1;
}
```

Ones counter FSMD in GEZEL



```
dp ones(in b, rst : ns(1); out o : ns(8)) {
  reg c  : ns(8);
  reg br, rstr : ns(1);
  sfg reset { c = 0; o = 0; br = b;
              rstr = 0;}
  sfg inc   { c = c + 1; o = c;
              br = b; rstr = rst; }
  sfg hold  { br = b; rstr = rst;
              o = c;}
}
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) -> s1;
      else if (br) then (inc) -> s1;
      else (hold) -> s1;
}
```

Communication

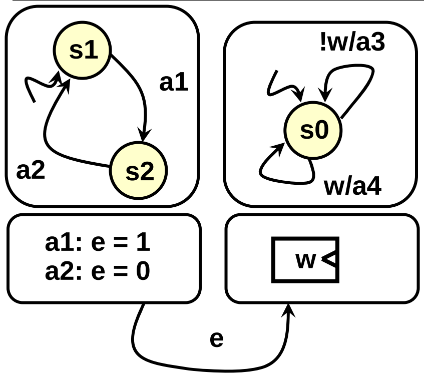


a1: w = 0
a2: w = 1

a3 executes with a1
a4 executes with a2

- FSM(D) can exchange events
- Communication is global and instantaneous
  (nice for math and proofs, but not physically realizable!)

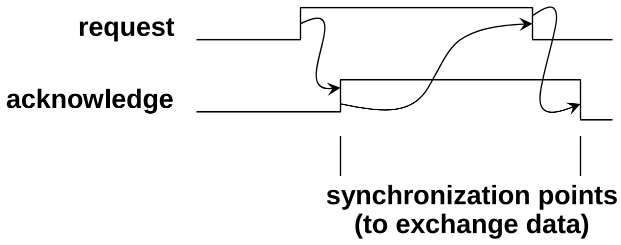
GEZEL: Physical Communication

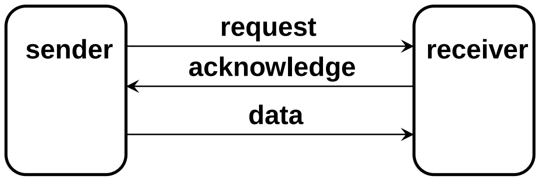


```
dp datapath(in e : ns(1)) {
  reg w : ns(20);
  always  {w = e;}
  sfg a3  {...}
  sfg a4  {...}
}
fsm onestate(datapath) {
  initial s0;
  @s0 if (w) then (a3) -> s0;
             else (a4) -> s0;
}
```

• Datapaths have ports to write & read events
• Data is captured in register and used in state transition condition. This causes a cycle delay; physical communication always has delay

Synchronization





Synchronized Communication

Requirements for sender/receiver synchronization depending on various communication primitives

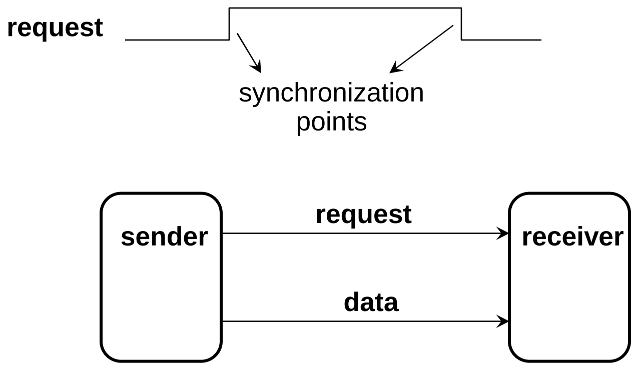
Sender/Receiver communication			**Sender/Receiver synchronization**
blocking-read blocking-write			Two-phase handshake
nonblocking-read blocking-write	**or**	blocking-read nonblocking-write	Single-phase handshake
nonblocking-read nonblocking-write			Perfect synchrony

Two-phase handshake - sender

```
dp sender(out req : ns(1);
          in  ack : ns(1);
          out d   : ns(8)) {
  reg rack : ns(1);
  always     {rack = ack;}
  sfg reqhi  {req = 1;}
  sfg reqlo  {req = 0;}
  sfg send   {d   = 10;}
  sfg idle   {d   = anything;}
}
fsm onestate(datapath) {
  initial s0, s1;
  @s0 (reqhi) -> s1;
  @s1 if (rack) then (send, reqlo) -> s2;
      else (reqhi, idle) -> s1;
  @s2 if (!rack) then (reqhi, idle) -> s1;
      else (reqlo, idle) -> s2;
}
```

synchronization points

Slow sender, fast receiver



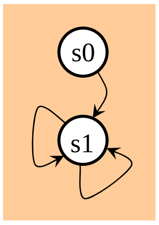
• A similar case also exists with a fast sender, slow receiver (only acknowledge signal)

Single-side handshake - slow sender

```
dp sender(out req : ns(1);
          out d   : ns(8)) {
  sfg reqhi  {req = 1;}
  sfg reqlo  {req = 0;}
  sfg send   {d   = 10;}
  sfg idle   {d   = anything;}
}
fsm onestate(datapath) {
  initial s0, s1;
  @s0 (reqhi, send) -> s1;
  @s1 (reqlo, idle) -> s0;
}
```

From FSM to gates

```
fsm ctl_ones(ones) {
  initial s0;
  state s1;
  @s0 (reset) -> s1;
  @s1 if (rstr) then (reset) ->
s1;
      else if (br) then (inc) -
> s1;
      else (hold) -> s1;
}
```



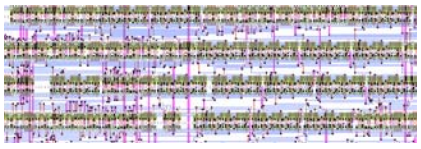
2 states = 1 bit
2 inputs = 2 bits
3 instructions = 2 bits

rstr	br	cs	ns	out[0][1]
X	X	0	1	00
1	X	1	1	00
0	1	1	1	01
0	0	1	1	10

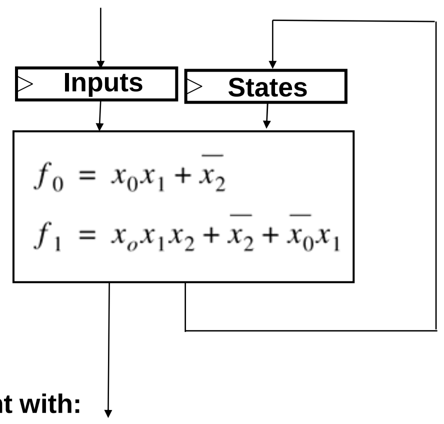
FSM to Gates

Fixed program
- **FSM to standard cells: perfect fit**
- **FSM to PLA (Programmable Logic Array): perfect fit**

With programmability:
- **Micro-programming**
- **Program counter**
- **Control unit of processor**


layout

From FSM to gates

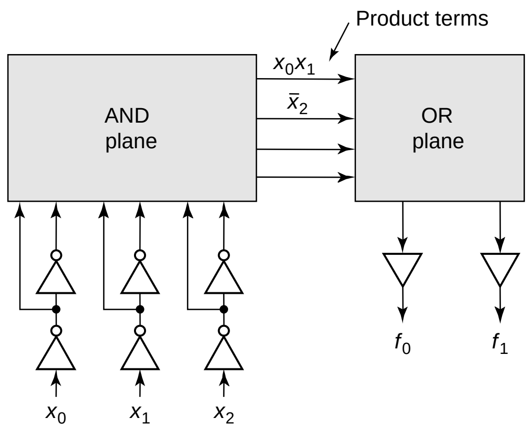


$$f_0 = x_0 x_1 + \overline{x_2}$$

$$f_1 = x_o x_1 x_2 + \overline{x_2} + \overline{x_0} x_1$$

Implement with:
-PLA
- Logic synthesis

A Historical Perspective: the PLA



[Rabaey 2nd edition]

Two-Level Logic

$$f_0 = x_0 x_1 + \overline{x_2}$$

$$f_1 = x_o x_1 x_2 + \overline{x_2} + \overline{x_0} x_1$$

minterm

Every logic function can be expressed in sum-of-products format (AND-OR)

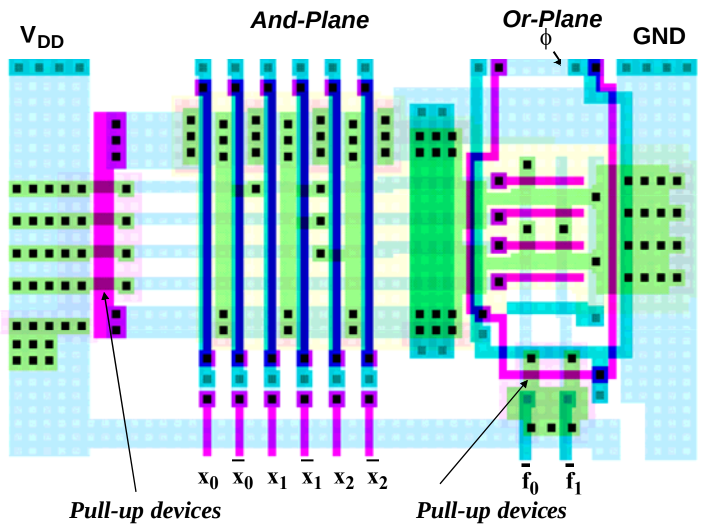
Inverting format (NOR-NOR) more effective

$$\overline{f_0} = \overline{\overline{(\overline{x_0} + \overline{x_1})} + \overline{x_2}}$$

$$\overline{f_1} = \overline{\overline{(\overline{x_0} + \overline{x_1} + \overline{x_2})} + \overline{x_2} + \overline{(x_0 + \overline{x_1})}}$$

[Rabaey 2nd edition]

PLA Layout – Exploiting Regularity


V_{DD}
And-Plane
Or-Plane
ϕ
GND
x_0 $\overline{x_0}$ x_1 $\overline{x_1}$ x_2 $\overline{x_2}$
$\overline{f_0}$ $\overline{f_1}$
Pull-up devices
Pull-up devices

[Rabaey 2nd edition]

Other alternatives

When program large & complex:

- Critical path & pipelining
- Micro programming
- Program Counter
- Control unit of processor

Conclusion

- Synchronous Data flow for stream processing
- Control flow for condition handling, events, special cases
- In reality: mixture of the two, always going to be some part not modeled easily.