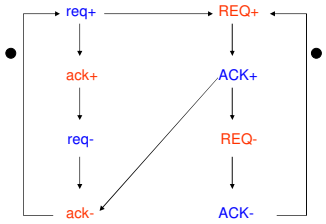
HY 590.20

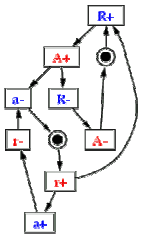
Χρονισμός Ψηφιακών Συστημάτων
Χρήστος Σωτηρίου
8

A faster STG?

- Does it need an extra variable?



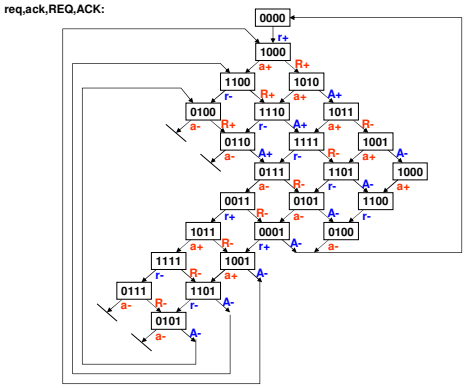
Drawn by `draw_astg`



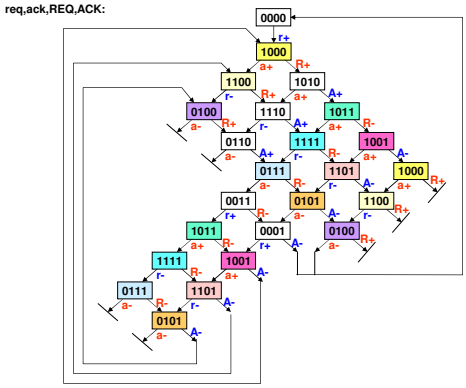
INPUTS: r,A
OUTPUTS: a,R

Note reverse colors

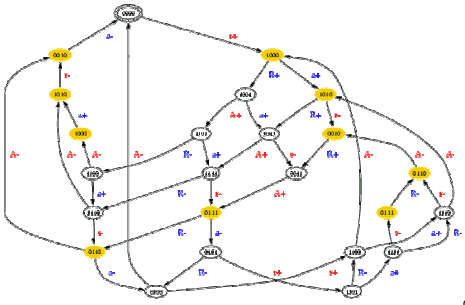
The SG



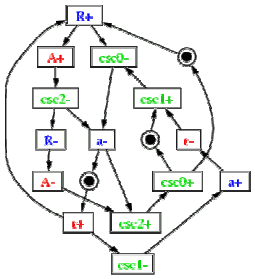
The SG



Drawn by `write_sg` & `draw_astg`

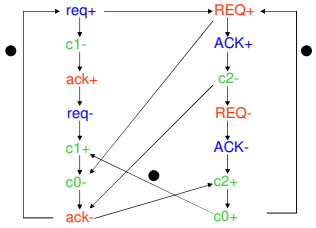


Extra states inserted by `petrify`



7

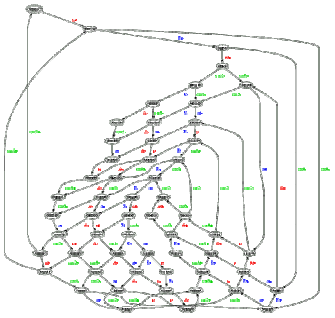
Rearranged STG



Initial Internal State: c0=c1=c2=1

8

The new State Graph…



9

The Synthesized Complex Gates Circuit

```
INORDER = r A a R csc0 csc1 csc2;
OUTORDER = [a] [R] [csc0] [csc1] [csc2];
[a] = a (csc2 + csc0) + csc1';
[R] = csc2 (csc0 (a + r) + R);
[csc0] = csc0 (csc1' + a') + R' csc2;
[csc1] = r' (csc0 + csc1);
[csc2] = A' (csc0' (csc1' + a') + csc2);
```

10

Technology Mapping

```
INORDER = r A a R csc0 csc1 csc2;
OUTORDER = [a] [R] [csc0] [csc1] [csc2];
[0] = R';                                  # gate inv:       combinational
[1] = [0]' A' + csc2';                     # gate oai12:     combinational
[a] = a csc0' + [1];                       # gate sr_nor:    asynch
[3] = csc1';                               # gate inv:       combinational
[4] = csc0' csc2' [3]';                    # gate nor3:      combinational
[5] = [4]' (csc1' + R');                   # gate aoi12:     combinational
[R] = [5]';                                # gate inv:       combinational
[7] = (csc2' + a') (csc0' + A');           # gate aoi22:     combinational
[8] = csc0';                               # gate inv:       combinational
[csc0] = [8]' csc1' + [7]';                # gate oai12:     combinational
[csc1] = A' (csc0 + csc1);                 # gate rs_nor:    asynch
[11] = R';                                 # gate inv:       combinational
[12] = csc0' ([11]' + csc1');              # gate aoi12:     combinational
[csc2] = [12] (r' + csc2) + r' csc2;       # gate c_element1:asynch
```

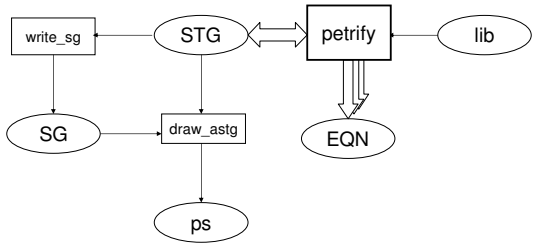
11

The Synthesized Gen-C Circuit

```
INORDER = r A a R csc0 csc1 csc2;
OUTORDER = [a] [R] [csc0] [csc1] [csc2];
[0] = csc0' csc1 (R' + A);
[1] = csc0 csc2 (a + r);
[2] = csc2' A;
[R] = R [2]' + [1];                        # mappable onto gC
[4] = a csc1 csc2';
[csc0] = csc0 [4]' + csc2;                 # mappable onto gC
[6] = r' csc0;
[csc1] = csc1 r' + [6];                    # mappable onto gC
[8] = A' csc0' (csc1' + a');
[csc2] = csc2 R' + [8];                    # mappable onto gC
[a] = a [0]' + csc1';                      # mappable onto gC
```

12

Petrify Environment



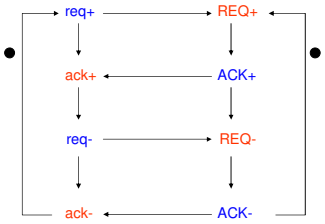
13

Petrify

- Unix (Linux) command line tool
- **`petrify -h`** for help (flags etc.)
- **`petrify -cg`** for complex gates
- **`petrify -gc`** for generalized C-elements
- **`petrify -tm`** for tech mapping
- **`draw_astg`** to draw
- **`write_sg`** to create state graphs
- Documented on line, incl. tutorial
  - See /usr/local_linux/async/doc/petrify

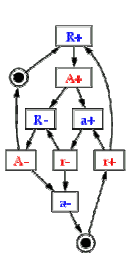
14

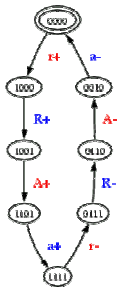
A safer STG?



15

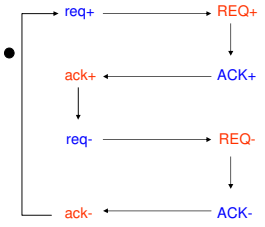
A safer STG?



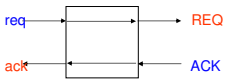


16

The safer STG is a serial circuit

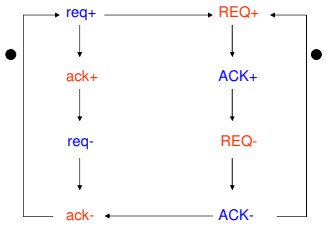



```
INORDER = r A a R;
OUTORDER = [a] [R];
[a] = A;
[R] = r;
```



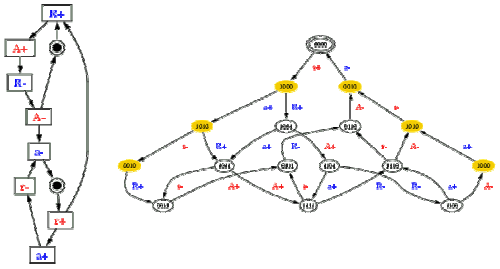
17

Yet another STG?

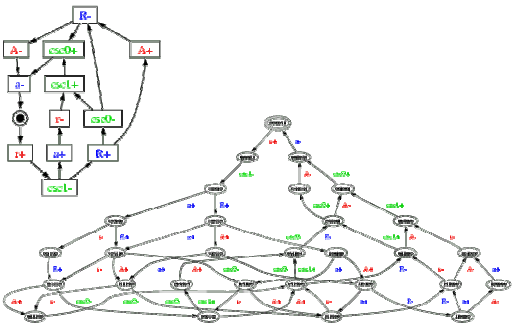


18

Yet another STG?

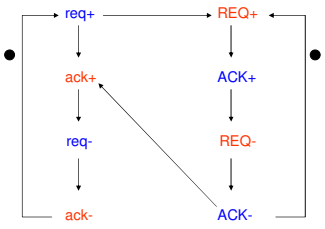


19



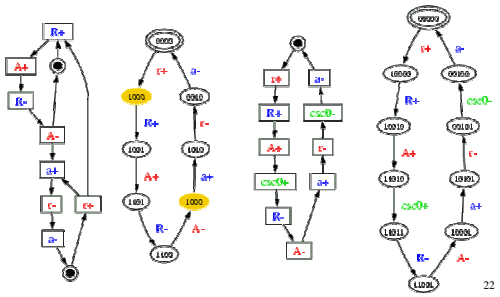
20

Output Handshake First



21

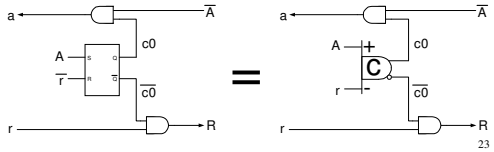
Still a serial controller



22

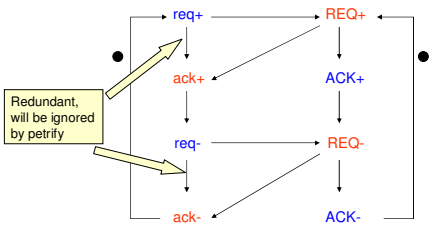
Synthesis

```
INORDER = r A a R csc0;
OUTORDER = [a] [R] [csc0];
[a] = csc0 A';          # gate and2_1:   combinational
[R] = r csc0';          # gate and2_1:   combinational
[2] = A' (csc0' + r'); # gate aoi12:    combinational
[csc0] = [2]';          # gate inv:      combinational
```



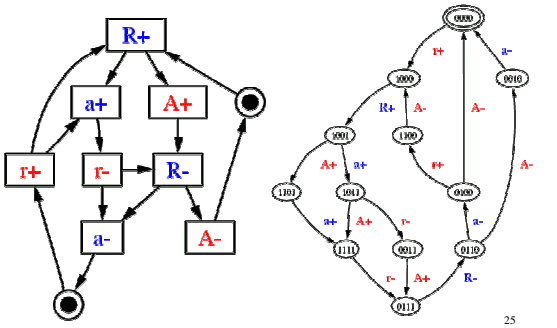
23

A different STG



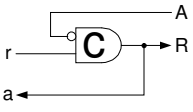
24

A different STG



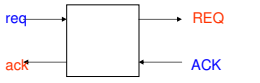
Synthesis

```
INORDER = r A a R;
OUTORDER = [a] [R];
[a] = R;
[1] = r A';
[2] = r' A;
[R] = R [2]' + [1];       # mappable onto gC
→R = R(Ar')'+A'r = R(A'+r)+A'r = A'r + RA' +Rr
```

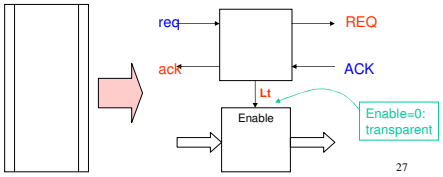


Latch Control

Data-less fifo:



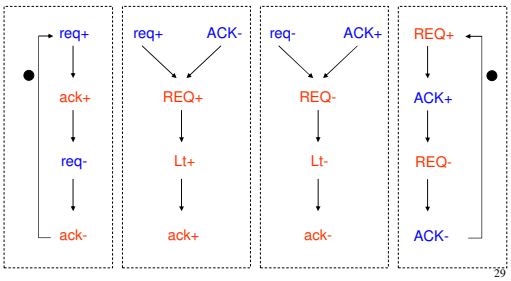
Latch:



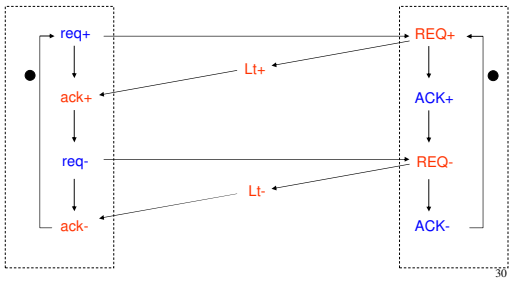
Constraints on sequence of events

- Must keep input data available until after it is latched
- Assume input data available only when req=1
- Once ack+, req- (and input data invalid) can follow very fast
- Lt+ before ack+

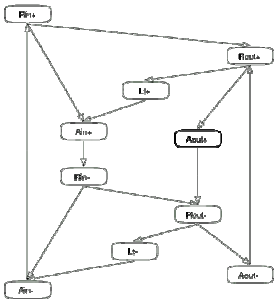
Latch Control: STG Fragments



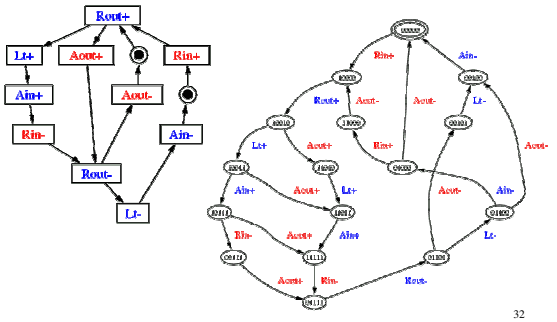
Latch Control: Combined STG



Latch Control: Combined STG

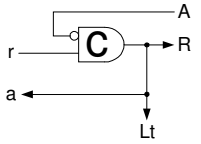


31



32

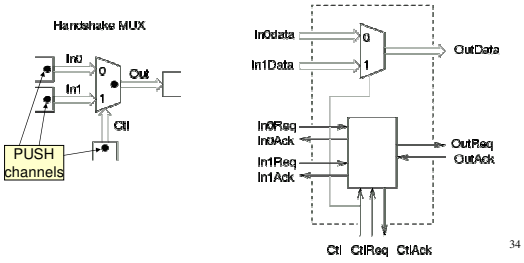
```
INORDER = Rin Aout Ain Rout Lt;
OUTORDER = [Ain] [Rout] [Lt];
[Ain] = Lt;
[1] = Aout' Rin;
[2] = Aout Rin';
[Rout] = Rout [2]' + [1];     # mappable onto gC
[Lt] = Rout;
→R = R(Ar')'+A'r = R(A'+r)+A'r = A'r + RA' +Rr
```



33

MUX Control

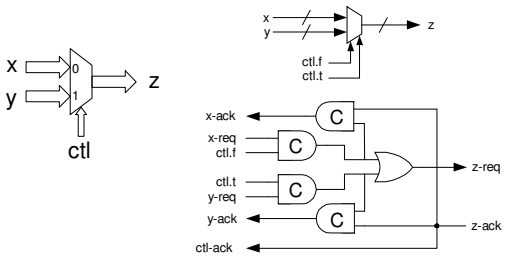
- So far all examples are doable "by hand"
- A deceptively simple example:
  Control for 4-phase bundled data mux



34

4-phase Bundled-data Mux

- We have already drawn this (dual-rail control):

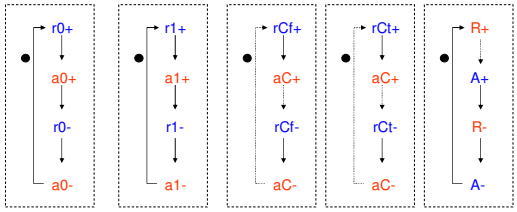


Easier to start with the dual-rail control

35

The environment

- Four *independent* environments (rCf, rCt share aC)
- Must not specify any dependency by mistake:

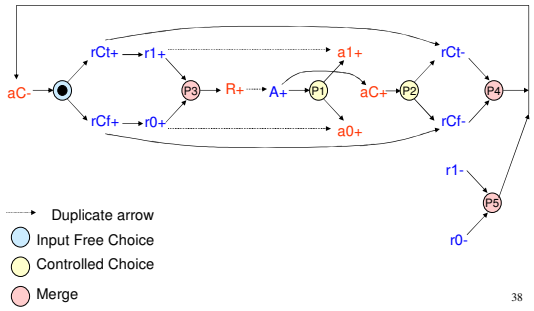


36

The control

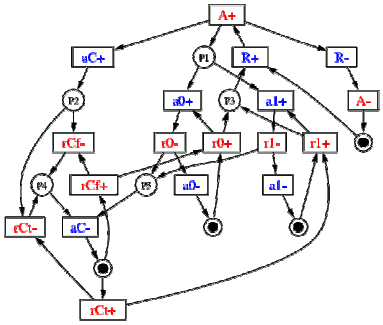
- A+ must precede a0+ or a1+ (make sure data passed through and were captured)
- In0 and In1 handshakes must be made MUTEX
- Choices are matches with Merges

37



38

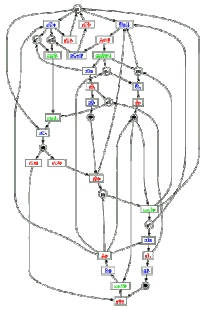
mux.g



39

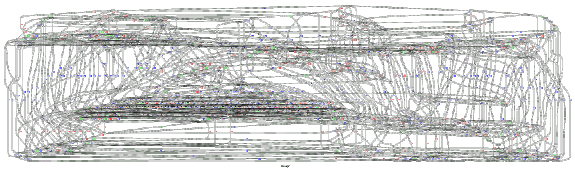
mux.gc

- Replicated transitions: R+, R+/1
- Inserted state variable to retain In0 vs. In1 info



40

Compact state graph



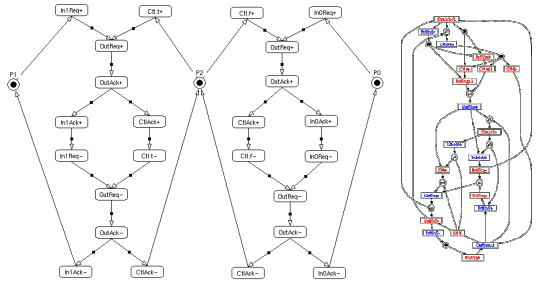
41

Mux Control Synthesis

```
INORDER = r0 r1 rCf rCt A a0 a1 aC R csc0 csc1;
OUTORDER = [a0] [a1] [aC] [R] [csc0] [csc1];
[0] = r0 csc0;
[a1] = csc1 r1;
[2] = csc1 (A + csc0);
[3] = rCt' rCf' csc1';
[aC] = aC [3]' + [2];                  # mappable onto gC
[R] = a0' csc0' csc1 r0 + csc0 csc1';
[6] = a0' csc1 A r0 + csc1' r1 A';
[7] = aC (r0' + a0);
[csc0] = csc0 [7]' + [6];              # mappable onto gC
[9] = r0 A' + csc0 A;
[10] = r0' csc0' r1';
[csc1] = csc1 [10]' + [9];             # mappable onto gC
[a0] = a0 r0 + [0];                    # mappable onto gC
```

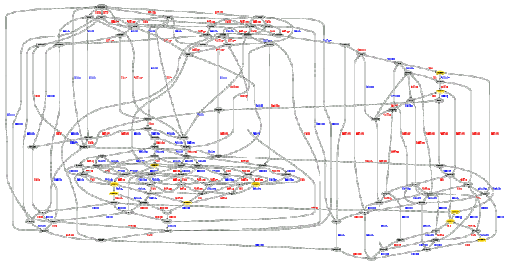
42

Another Mux (4p ctl, bd) *(Fig 6.24)*



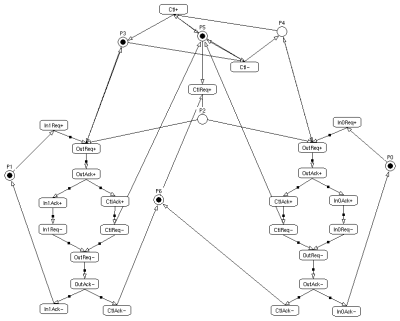
43

SG for the Fig 6.24 mux



44

All-bundled Mux



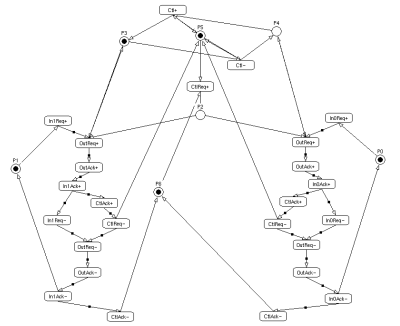
45

All-bundled Mux Synthesis

```
INORDER = In0Req OutAck In1Req Ctl CtlReq In1Ack In0Ack OutReq
   CtlAck csc0;
OUTORDER = [In1Ack] [In0Ack] [OutReq] [CtlAck] [csc0];
[In1Ack] = OutAck csc0';
[In0Ack] = OutAck csc0;
[2] = CtlReq (In1Req csc0' + In0Req Ctl');
[3] = CtlReq' (In1Req' csc0' + In0Req' csc0);
[OutReq] = OutReq [3]' + [2];          # mappable onto gC
[5] = OutAck' csc0;
[CtlAck] = CtlAck [5]' + OutAck;       # mappable onto gC
[7] = OutAck' CtlReq';
[8] = CtlReq Ctl;
[csc0] = csc0 [8]' + [7];              # mappable onto gC
```

46

Reduced concurrency mux



47

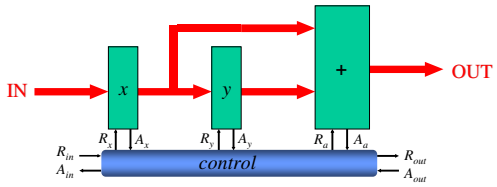
Following slides borrowed from Jordi Cortadella, UPC

A simple filter: specification

```
y := 0;
loop
   x := READ (IN);
   WRITE (OUT, (x+y)/2);
   y := x;
end loop
```

48

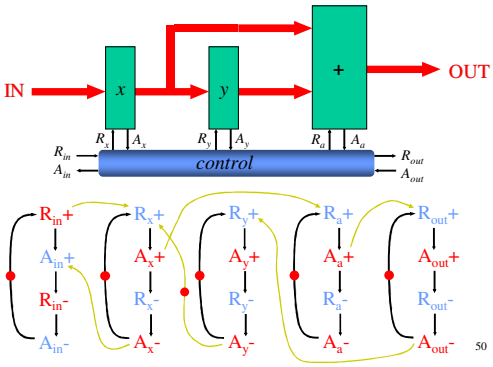
A simple filter: block diagram



- x and y are level-sensitive latches (transparent when R=1)
- + is a bundled-data adder (matched delay between R_a and A_a)
- R_{in} indicates the validity of IN
- After A_{in}+ the environment is allowed to change IN
- (R_{out},A_{out}) control a level-sensitive latch at the output

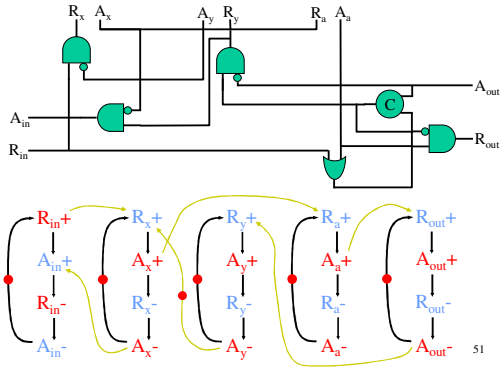
49

A simple filter: control spec.



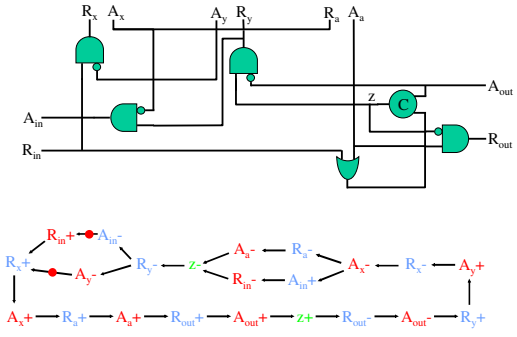
50

A simple filter: control impl.



51

Control: observable behavior



52