VHDL Design Flow

BEHAVIORAL AND LOGIC SYNTHESIS

El Mostapha Aboulhamid
Dépt. IRO, Université de Montréal
CP 6128, Succ. Centre-Ville
Montréal, Qc. H3C 3J7

Phone: 514-343-6822 FAX: 514-343-5834
aboulham@iro.umontreal.ca

Acknowledgment: François Boyer (boyerf@iro.umontreal.ca) had a major contribution in the development of the labs contents.

Invoking the scheduler 26
HDL descriptions and semantics 1
Objectives 2
Pre-synthesis model 3
The design 4
The design (cont'd) 5
Behavioral processes 6
Behavioral processes (cont'd) 7
Clock and Reset 8
Synchronous resets 9
Synchronous resets (cont'd) 10
Asynchronous resets 11
I/O Operations 12
I/O Operations 13
I/O Operations
(cont'd) 14
Flow of Control 15
Fixed bound FOR loops 16
General loops 17
Pipelined loops 18
Pipelined loops 19
Pipelined loops and Fixed I/O mode 20
Other I/O modes 21
Memory inference 22
Memory code 23
Memory timing 24
Memory timing (cont'd) 25
Other memory considerations 26
Synthetic components 27
DesignWare developer 28

Preserved functions 29
Pipelined components 30
I/O modes 1
I/O modes 2
I/O modes (cont'd) 3
Cycle-Fixed Mode 4
Cycle-Fixed Mode
(Test bench) 5
Fixed Mode rules
(Straight line code) 6
Fixed Mode rules
(Loops) 7
Loops in fixed mode 8
Nested loops and FM 9
Successive loops and FM 10
Complex loop conditions 11
Superstate-Fixed Mode 12
Superstate-Fixed Mode
(Implications) 13
Superstate Rules
(continuing superstate) 14
Superstate Rules
(separating write orders) 15
Superstate Rules
(Conditional superstate) 16
Superstate Rules
(Escaping from the loop) 17
Free-Floating Mode 18
Explicit Directives and Constraints 1
Labeling

(Default naming) 2
Labeling
(user naming) 3
Labeling
(improved naming) 4
Scheduling Constraints 5
Scheduling Constraints
(cont'd) 6
Shell Variables 7
Shell Variables (cont'd) 8
Shell Commands 9
RTL Design Methodology 1
RTL Design flow 2
RTL Design flow 3
Design refinement 4
HDL FF Code 5
HDL latch Code 6
HDL AND Code 7
MUX inference 8
MUX modeling 9
Synthesized gate-level netlist simulation 10
Netlist simulation (cont'd) 11
Simulation of commercial ASICs 12
Design for Testability 13
Design Re-use 14
Designing with DW Components 15
FPGA Synthesis 16
Links to layout 17
DC and DA environments 18
DC and DA environments

(cont'd) 19
Target, Link, and Symbol Libraries 20
Libraries generation 21
VHDL RTL SEMANTICS 1
Types, signals and variables 2
Buffer mode modeling 3
STD_LOGIC 4
Arithmetic 5
Unwanted latches 6
Asynchronous reset 7
Synchronous reset 8
VHDL specifics 9
VHDL specifics
(cont'd) 10
Finite state machines 11
State encoding 12
HDL description of a state machine 13
Recommended style 14
Enumerated types and encoding 15
General description of FSM 16
Guidelines for FSM coding 17
fail-safe behavior 18
Memories 19
Memory behavior 20
Barrel shifter 21
Multi-bit register 22
Methodology for RTL synthesis 1
Objectives 2
Synthesis constraints 3
Design rule constraints 4

I. General Design Flow

Top-down design

+: Rough outlines explored at the highest possible level

+: Fine-grained optimization at lower levels

+: Wider variety of design can be explored at the higher levels
- Functionality partitioned into blocks and processes
- Blocks can be mapped to software or hardware

+: At lower levels exploration limited to:
- Area
- Speed
- Testability
- Power consumption

-: Iteration between levels may be inevitable

Description paradigms and abstraction levels

Data flow

- Input data as stream of samples
- Stream oriented operations
- Tools: graphical or dataflow oriented languages (Silage)

Control oriented

- Emphasis on states and transitions
- Ex: Protocol descriptions
  Graphical or languages or both: SDL
- FIFOs
- High level synchronization mechanisms
- Non-determinism
- Output can be sent either to RTL synth. or behav. synthesis
  - Depending on states corresponding to circuit states or not

Description paradigms and abstraction levels (cont'd)

Behavioral

- Language based
- Offers scheduling and allocation
- Front-end to RTL synthesis

RTL

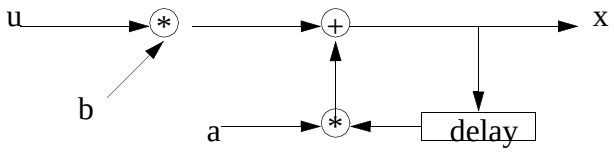
- Language or graphical
- Hierarchy
- Finer optimizations
- Technology independent

Gate

Data Flow Descriptions

Data represented as streams of samples

$x = axz^{-1} + bu$: Two synchronized streams produce a third stream



Abstraction enables very fast simulation

Synthesis

Transform a stream oriented description into a time-oriented description

$x_k = ax_{k-1} + bu_k$

Synthesis at either RTL or behavioral level

Why considered high level?

Stream through a time-varying channel

Statistical analysis, work load ...

Control Oriented Descriptions

Emphasis on states and transitions

Input: graphical or textual or both

May be hierarchical

Synthesis

Mapping to a HDL text

RTL or behavioral synthesis depending on the level of abstraction

Differences with data flow representation

DF: Collection of streams where each element of a stream is processed in a similar way

ST: Data and treatment less regular

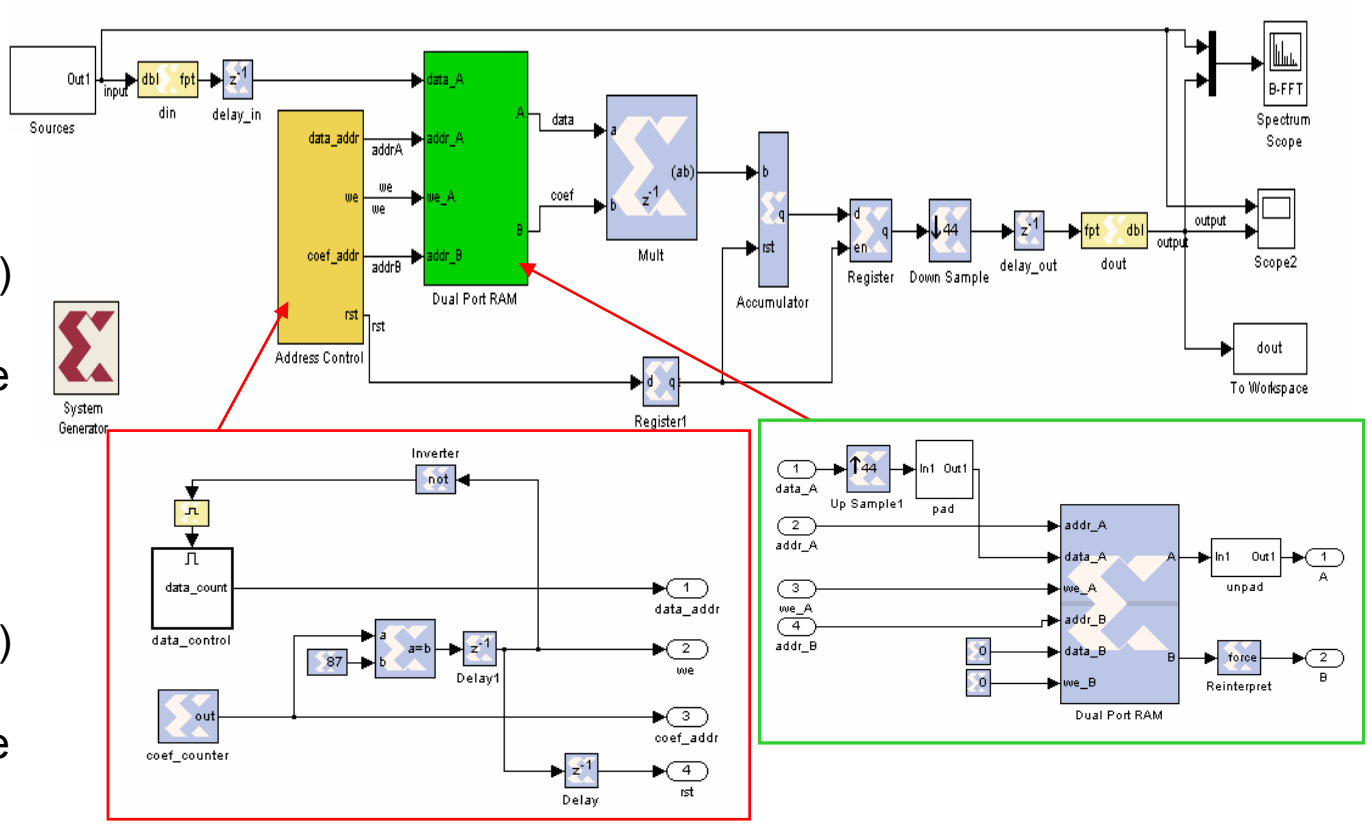
Different behaviors in different states

Lab 7: Building a MAC FIR Filter Solution

Slice Count: (without MULT18x18)
169 slices
Performance
~132 MHz

Slice Count: (with MULT18x18)
119 slices
Performance
~132 MHz

Speed files:
PRODUCTION v1.93



Behavioral Descriptions

Backend to Dataflow or Control oriented descriptions

General purpose

Language based: VHDL, Verilog, C, ISP, Pascal, etc.

HDL advantages:

Standardized

Simulatable

Readable interchange formats

VHDL and Verilog

+ High quality simulators

+ Existing RTL and logic synthesis tools

+ Large customer base of designers

- Closed tools (academic point of view)

Behavioral Synthesis (input)

Input = one process

```
process
 variable x, u: integer := 0;
begin
 u := inp;
 x:= a*x + b*u;
 outp <= x;
 wait until clock'event and clock=1;
end;
```

Clock edges may be added during behavioral synthesis

If many process: each process scheduled independently

Mixed descriptions allowed: glue logic, RTL processes, behav. processes

Scheduling

Input= process ⇒ Output= FSM + datapath
Operations assigned to states
User Responsibility in RTL synthesis
States are part of an FSM
Additional states if allowed by the user
States = actual machine states
Transitions correspond to machine's clock edge
Machine clock (10 Mhz) may be much faster than the sample clock (50KHz)
In RTL and behavioral machine clock considered
In Data Stream sample clock is considered

Allocation

Operations assigned to functional hardware

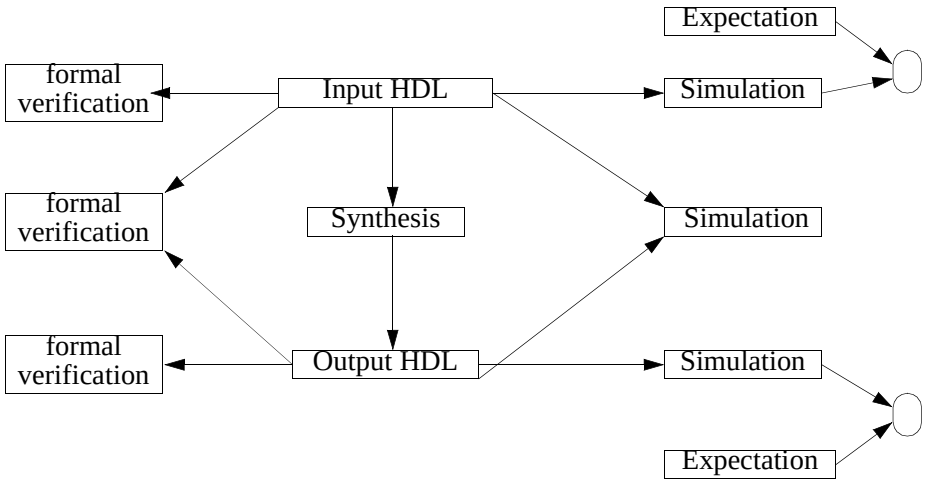
Data values assigned to storage elements

Optimization algorithm based on variables lifetime

Broader possibilities at behavioral level / RTL

Trade-off area/latency
Register area/combinational-interconnect area

Design validation



Simulation and verification

Simulation

- Test the response of the design under a selected set of inputs
- Never exhaustive
- Generation and application very time-consuming
- Coverage has to be defined
- Present way of validating designs

Formal verification

- Test mathematical properties
- Proof “equality” of two designs
  - Equality of boolean expressions
  - Bissimultaion in process algebra (CCS)
- Not yet the main stream
- Strict methodology for specification
- Alleviates simulation limitations

RTL and behavioral design

Behavioral synthesis

A gap between domain specific tools and RTL synthesis tools

A higher level of abstraction for the designer to logic synthesis

HDL design flow

Initial model in C or C++ or "simulation VHDL"

Define and test the functional aspects of the design:

- bit widths, operation ordering, rounding strategies in a filter design
- number of operations necessary to unpack a field and store a packet in a ATM packet router

Timing, States and other properties at an abstract level: unlimited queues.

Initial model translated into an HDL model

Accurate and natural modeling of concurrency and time
Simulation of the interaction between modules
Refinement of interfaces

Synthesis

requires the use of a subset of the HDL

VHDL Synthesizable Subset

Fully supported

Arith, log., relational *operators*

Entity declarations; Architecture bodies, Arrays

Attributes: RIGHT, LEFT,HIGH, LOW, BASE , RANGE, LENGTH

Component declarations and instantiations

Concurrent procedure calls; concurrent signal assignments; constant declarations

Enumerated, integer types;

If, case, loop statements

Next, return statements

Subprograms, declarations, bodies; subprog. and operator overloading

Qualified and static expressions

Type conversions

Package declarations and bodies

VHDL Synthesizable Subset (cont'd)

Partially supported

Aggregates

wait and EVENT on clock edge

Exit only from local or reset loop

Transport delay in pipeline

Limited resolution functions: wired and, or, three-state

No waveforms

Ignored

Access and file type

Aliases ; Assertions

Floating point, Physical types

VHDL Synthesizable Subset (cont'd)

Unsupported

Allocators

Disconnection

TEXTIO

Attributes:

POS, VAL, SUCC, PRED, LEFTOF, RIGHTOF

DELAYED, TRANSACTION

RIGHT(N), LEFT(N). RANGE(N), REVERSE_RANGE(N), HIGN(N), LOW(N)

ACTIVE, LAST_ACTIVE, LAST_EVENT

Special attributes

ARRIVAL, FALL_ARRIVAL, RISE_ARRIVAL

DRIVE, RISE_DRIVE, FALL_DRIVE

LOGIC_ONE, LOGIC_ZERO

EQUAL, OPPOSITE

DONT_TOUCH_NETWORK

LOAD

DONT_TOUCH

MAX_AREA

ENUM_ENCODING

UNCONNECTED

HOLD_CHECK, SETUP_CHECK

MAX_TRANSITION, MAX_DELAY, MIN_DELAY, MIN_RISE_DELAY, MIN_FALL_DELAY

Main Features of Behavioral Synthesis

Automatic assignment of non-I/O operations to states

I/O scheduling is more restricted

Automatic construction of the FSM

Operations mapped onto hardware taking into account different trade-offs

Number of functional units

Latency

Exploration time

Interconnects

Unit selection (carry lookahead vs. ripple carry): may be overridden manually

Main Features of Behavioral Synthesis (cont'd)

Allocation of variables to registers

Optimization algorithm

Based on life-time intervals

Easy changes

Number of states in a pipeline by changing a single constraint

May result in a change in the control FSM

Further steps

Logic optimization

Test insertion

Retiming

RTL Descriptions

RTL descriptions 3 to 5 times longer than behavioral descriptions

More development time to get a good model

More errors

Less readable

Management of next state transition by the user

The user has to manage most of the allocation of registers

More difficult to deal with

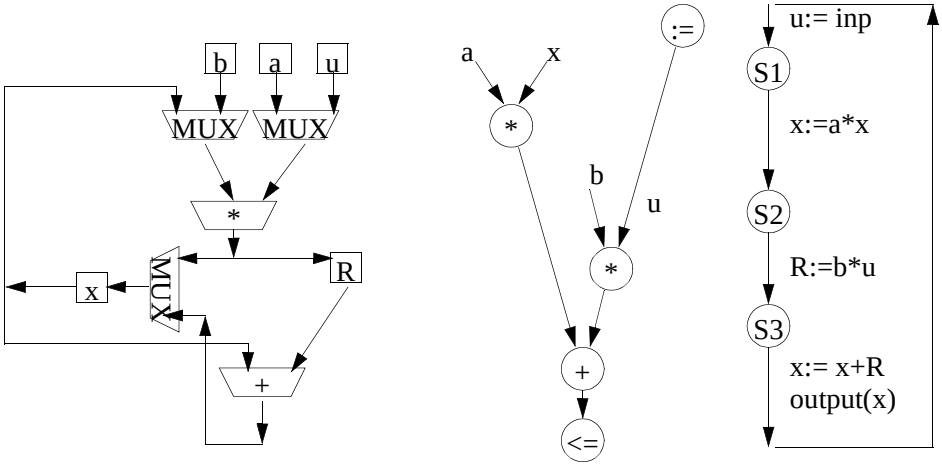
Conditions, pipelining, multiple cycle operations

Memory and register Reads an Writes

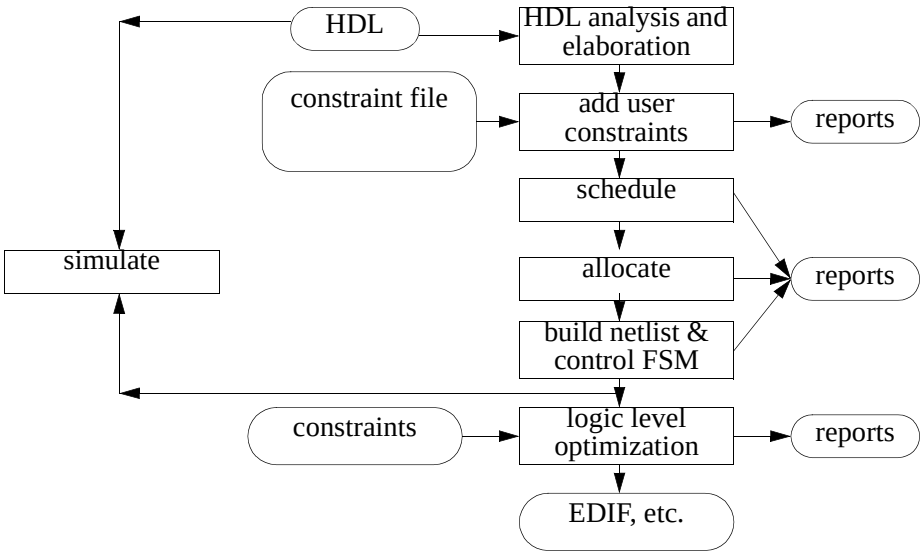
Loops boundaries

subprograms

Scheduling and allocation illustration



Behavioral Compiler Design Flow



Steps of the BC Design Flow

Analysis: parsing and preliminary semantics

Elaboration: different from simulation
- Mixed control/dataflow representation
- Explicit parallelism

Explicit constraints
- I/O operations
- Target technology
- Clock cycle

Early timing analysis (allows chaining of operations)

Scheduling

Allocation

Reports on all the previous steps

If not satisfied reiterate by changing constraints

otherwise proceed with logic synthesis

References

David W. Knapp, “Behavioral Synthesis, Digital System Design Using the Synopsys behavioral Compiler,” Prentice Hall PTR, 231 pages, 1996.
- Covers behavioral synthesis and different case studies

Pran Kurup and Taher Abbasi, “Logic Synthesis Using Synopsys,” Second Edition,

Kluwer, 322 pages, 1997.
- Covers logic synthesis
- Original presentation
- Interesting scenarios

Giovanni De Micheli, “Synthesis and Optimization of Digital Circuits,” McGraw-Hill, 579 pages, 1994.
- Best book on theory and algorithms for HLS
- Pipelines not covered
- Departs from Synopsys view of I/O

On-line Synopsys Documentation
- Hyperlink- documentation
- Complete

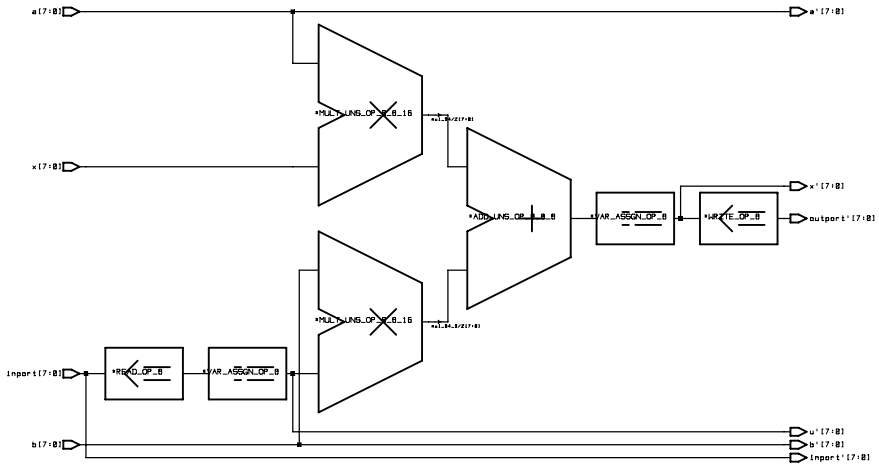
II. Design Flow Example

VHDL code

```
package types is
subtype small_int is integer range 0 to 255;
end types;

library ieee;
use ieee.std_logic_1164.all;


use work.types.all;
entity ex_bhv is
  port(clk,stop: in std_logic; inport,alpha,beta: in small_int;
       outport: out small_int);
end ex_bhv;

library ieee;
use ieee.std_logic_1164.all;

architecture algo of ex_bhv is
begin
 process
```

```
variable a,b,u,x:small_int ;
begin
 Reset_loop: loop
   -- Reset tail
   outport <= 0;
   u:= 0; x:=0;
   a:= alpha;
   b:= beta;
   wait until clk'event and clk='1';
   if stop ='1' then exit reset_loop; end if;
   main_loop: loop
     -- normal mode behavior
     u := inport;
     x:= a*x + b*u;
     outport <= x;
     wait until clk'event and clk='1';
     if stop ='1' then exit reset_loop; end if;
   end loop main_loop;
 end loop Reset_loop;
end process;
end algo;
```


Main loop after elaboration



bc_analyzer> bc_time_design

```
Cumulative delay starting at inport_33:
   inport_33  =  0.000000
    mul_34_2  =  16.996946
     add_34  =  19.182245
      outport_35  =  19.182245
Cumulative delay starting at mul_34_2:
   mul_34_2  =  17.055845
    add_34  =  19.241144
     outport_35  =  19.241144
Cumulative delay starting at mul_34:
   mul_34  =  17.055845
    add_34  =  19.241144
     outport_35  =  19.241144
Cumulative delay starting at outport_35:
   outport_35  =  0.000000
Cumulative delay starting at add_34:
   add_34  =  13.757200
    outport_35  =  13.757200
Cumulative delay starting at beta_28:
   beta_28  =  0.000000
Cumulative delay starting at alpha_27:
   alpha_27  =  0.000000
```


bc_analyzer> create_clock -name “clk” -period 18 -waveform { “0” “9” } { “clk” }

bc_analyzer> bc_check_design

```
Error: Fixed IO schedule is unsatisfiable (HLS-52)
```

bc_analyzer> bc_check_design -io su

```
No errors were found.
```

bc_analyzer> schedule -io su -eff zero

```
*********************************************
*  Operation schedule of process process_20:  *
*********************************************

Resource types
==================================
    beta......8-bit input port
    loop......loop boundaries
    p0........8-bit input port alpha
    p1........8-bit input port inport
    p2........8-bit registered output port outport
    r29.......(8_8->8)-bit DW01_add
    r41.......(8_8->16)-bit DW02_mult
    r47.......(8_8->16)-bit DW02_mult
```

```
                                                    D       D
                                                    W       W
                                              D     0       0
                                              W     2       2
                                              0     _       _
                                              1     m       m
                    p     p     p             _     u       u       p
                    o     o     o             a     l       l       o
                    r     r     r             d     t       t       r
                    t     t     t             d                     t
-------+------+------+-----+-----+------+-------+--------+-----
 cycle | loop | beta | p0  | p1  | r29  |  r47  |  r41   | p2
-------------------------------------------------------------------
   0   |..L3..|.R28..|.R27.|.....|......|.......|........|.W25.
       |..L0..|......|.....|.....|......|.......|........|.....
   1   |..L6..|......|.....|.R33.|......|.o1150.|.o1150a.|.....
   2   |......|......|.....|.....|.o841.|.......|........|.W35.
   3   |..L8..|......|.....|.....|......|.......|........|.....
       |..L7..|......|.....|.....|......|.......|........|.....
       |..L5..|......|.....|.....|......|.......|........|.....
       |..L4..|......|.....|.....|......|.......|........|.....
       |..L2..|......|.....|.....|......|.......|........|.....
       |..L1..|......|.....|.....|......|.......|........|.....
```

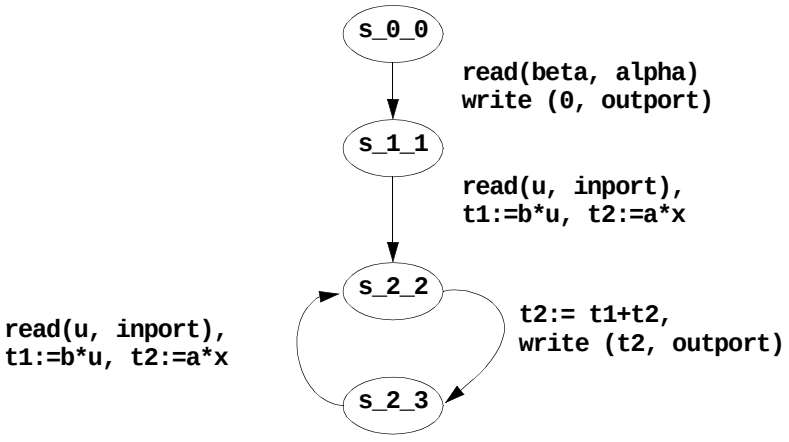


```
Operation name abbreviations
============================
 L0.........loop boundaries process_20_design_loop_begin
 L1.........loop boundaries process_20_design_loop_end
 L2.........loop boundaries process_20_design_loop_cont
 L3.........loop boundaries Reset_loop/Reset_loop_design_loop_begin
 L4.........loop boundaries Reset_loop/Reset_loop_design_loop_end
 L5.........loop boundaries Reset_loop/Reset_loop_design_loop_cont
 L6.........loop boundaries Reset_loop/main_loop/main_loop_design_loop_begin
 L7.........loop boundaries Reset_loop/main_loop/main_loop_design_loop_end
 L8.........loop boundaries Reset_loop/main_loop/main_loop_design_loop_cont
 R27........8-bit read Reset_loop/alpha_27
 R28........8-bit read Reset_loop/beta_28
 R33........8-bit read Reset_loop/main_loop/inport_33
 W25........8-bit write Reset_loop/outport_25
 W35........8-bit write Reset_loop/main_loop/outport_35
 o841.......(8_8->8)-bit ADD_UNS_OP Reset_loop/main_loop/add_34
 o1150......(8_8->16)-bit MULT_UNS_OP Reset_loop/main_loop/mul_34_2
 o1150a.....(8_8->16)-bit MULT_UNS_OP Reset_loop/main_loop/mul_34
```

bc_analyzer> report_schedule -abstract_fsm > fsm_rpt

```
*****************************************************
*  State graph style report for process process_20:  *
*****************************************************
 present      next
 state  input state                actions
-----------------------------------------------------------------
 s_0_0    -  s_1_1   a_0: Reset_loop/beta_28 (read)
                     a_1: Reset_loop/alpha_27 (read)
                     a_4: Reset_loop/outport_25 (write)
 s_1_1    -  s_2_2   a_2: Reset_loop/main_loop/inport_33 (read)
                     a_7: Reset_loop/main_loop/mul_34_2 (operation)
                     a_13: Reset_loop/main_loop/mul_34 (operation)
 s_2_2    -  s_2_3   a_3: Reset_loop/main_loop/outport_35 (write)
                     a_22: Reset_loop/main_loop/add_34 (operation)
 s_2_3    -  s_2_2   a_2: Reset_loop/main_loop/inport_33 (read)
                     a_7: Reset_loop/main_loop/mul_34_2 (operation)
                     a_13: Reset_loop/main_loop/mul_34 (operation)
```


FSM



bc_analyzer> remove_design -design

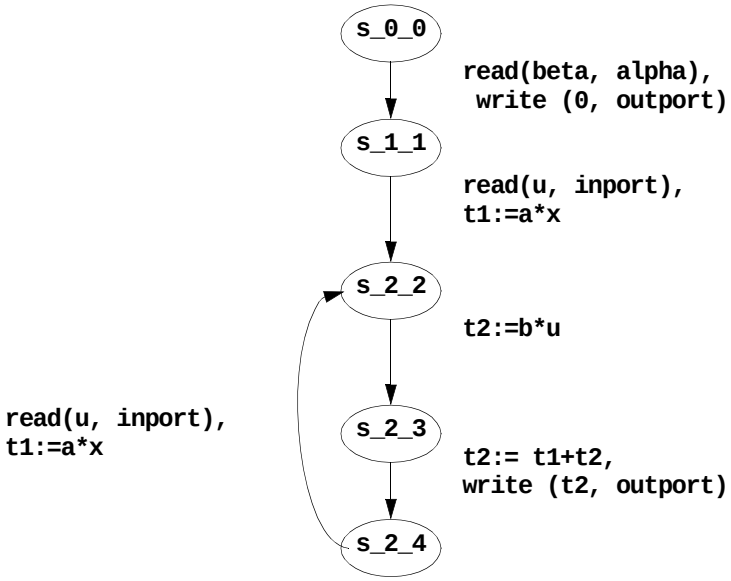
bc_analyzer> schedule -io su -eff zero -area

```
                    p     p     p      _       m      p
                    o     o     o      a       u      o
                    r     r     r      d       l      r
                    t     t     t      d       t      t
 -------+------+------+-----+-----+------+--------+-----
  cycle | loop | beta | p0  | p1  | r29  |  r41   | p2
 ------------------------------------------------------------
    0   |..L3..|.R28..|.R27.|.....|......|........|.W25.
        |..L0..|......|.....|.....|......|........|.....
    1   |..L6..|......|.....|.R33.|......|.o1150a.|.....
    2   |......|......|.....|.....|......|.o1150..|.....
    3   |......|......|.....|.....|.o841.|........|.W35.
    4   |..L8..|......|.....|.....|......|........|.....
        |..L7..|......|.....|.....|......|........|.....
        |..L5..|......|.....|.....|......|........|.....
        |..L4..|......|.....|.....|......|........|.....
        |..L2..|......|.....|.....|......|........|.....
        |..L1..|......|.....|.....|......|........|.....
```



```
 present      next
 state input  state     actions
-------------------------------------------------------------------
 s_0_0 -    s_1_1  a_0: Reset_loop/beta_28 (read)
                   a_1: Reset_loop/alpha_27 (read)
                   a_4: Reset_loop/outport_25 (write)
 s_1_1 -    s_2_2  a_2: Reset_loop/main_loop/inport_33 (read)
                   a_10: Reset_loop/main_loop/mul_34 (operation)
 s_2_2 -    s_2_3  a_7: Reset_loop/main_loop/mul_34_2 (operation)
 s_2_3 -    s_2_4  a_3: Reset_loop/main_loop/outport_35 (write)
                   a_19: Reset_loop/main_loop/add_34 (operation)
 s_2_4 -    s_2_2  a_2: Reset_loop/main_loop/inport_33 (read)
                   a_10: Reset_loop/main_loop/mul_34 (operation)
-------------------------------------------------------------------
```

FSM 2


s_0_0
read(beta, alpha),
write (0, outport)
s_1_1
read(u, inport),
t1:=a*x
s_2_2
t2:=b*u
read(u, inport),
t1:=a*x
s_2_3
t2:= t1+t2,
write (t2, outport)
s_2_4

III. BEHAVIORAL COMPILER

A CONCEPTUAL FRAMEWORK

Objectives

BC description

Inputs and outputs, capabilities and internal structure

Provides a conceptual framework

Understand error messages, the processing of the design

Design good inputs to the BC

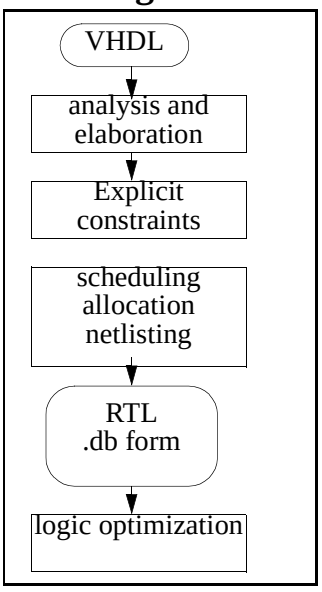
Interfaces

BC is a collection of function embedded in a program: bc_shell

Textual

Graphic interface: Design Analyzer (DA) tool

Design flow



Inputs

Input mechanisms

HDL text

bc_shell command language

Pragma directives: comments embedded in the HDL text

HDLs

VHDL and Verilog

One or more processes + logic external to processes

BC does not process any interaction between processes

Considers a process at a time without any ordering between processes

Processing steps

```
bc_shell> analyze -f  vhdl mydesign.vhd
```

Elaborate Command

```
bc_shell> elaborate -s  mydesign
```

The **s** flag: elaborate for scheduling.

Can be overridden by the attribute **rtl** attached to the process

⇒ mix both behavioral and RTL processes

Elaboration mode determined for each process

User constraints

Fixing a clock period and specifying the clock signal is mandatory

```
bc_shell> create_clock clk -period 9
```

Pairs of operations can be constrained

```
bc_shell> set_cycles 3 -from op1 to op2
```

Implementation of components may be forbidden

Processing steps (cont'd)

Timing analysis

BC is a specific technology scheduler

If long clock cycle operations may be chained

This step is invoked manually

```
bc_shell > bc_time_design
```

BC timing analysis is accurate: *bit_level* instead of *lumped* timing models

Report on combinational chains

Processing steps (cont'd)

Poss. changes after report on timing

Change the clock cycle or technology if needed

Estimate required number of cycles

Guide for manual implementation selection

Indicate multicycle operations: costly in both area and timing

⇒ The user may decide to change the operation, the implementation or the clock cycle

Scheduling

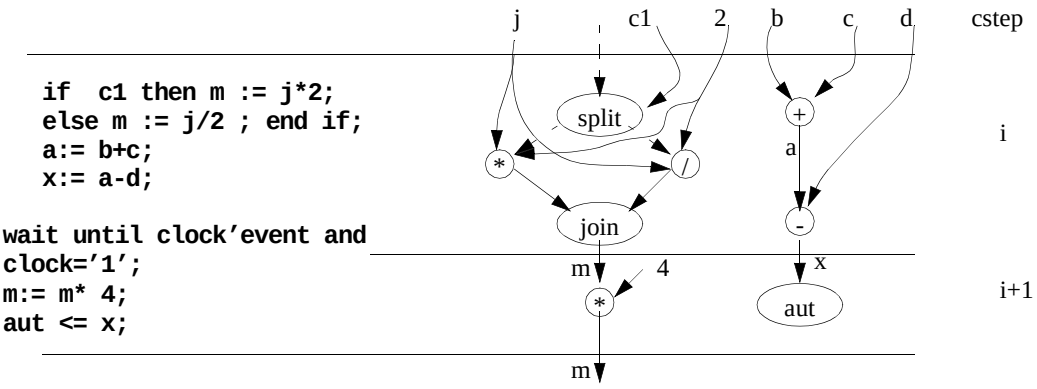
Operations mapped to control steps

Non-concurrent operations may share the same multiplexed hardware

⇒ reduces cost while meeting performance requirements

```
bc_shell> schedule -effort low
```


BC internals
Control-dataflow graph



CDFG

CDFG

Abstract representation of the circuit behavior

Without bias toward any schedule

Terminology

+, -, *, / belonging to cstep i are concurrent

BC (not DC) will allow "-" moved to next cstep

$\Rightarrow$ a dual unit add/sub can be shared

CDFG edges represent precedence

Latency: total number of csteps

CDFG nodes

Data

Arith/logic operations, some function calls

Synthetic nodes share hardware, *random logic* not

Patch boxes: bit and field selection, constant sources

Memory R/W: memory accesses

IO R/W: R/W to ports or signals

Conditional

split, join

Hierarchical

loops and function calls

Place holder

Loop control

loop begin, loop end, loop continue, loop exit

Chaining, multicycling, and pipelining

Controlling chaining

use set_cycles instead of

```
bc_shell> bc_enable_chaining = false
```

Multicycling

Controls and muxes should be registered

If conditional, FSM should commit at cycle i-1 to stabilize registers ⇒ extra cstep

Forcing unicycling regardless of timing analysis, use with caution:

```
bc_shell> bc_enable_multi_cycle = false
```

Pipelining

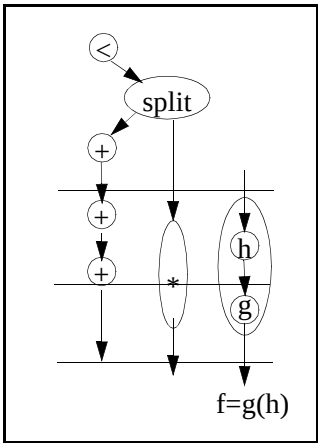
f(x) = g(h(x)): a register isolates h for g

Pipelining either automatic or by implementation directives

More expensive than a k-cycle operation but k times faster

Multiplication and memory operations are prime candidates

Chaining, multicycling, and pipelining (Illustration)



CDFG edges

***Data* edges**

Represent values

***Precedence* edges**

Represent order and control

t and f of a split node

Constraints

bc_shell> set_min_cycles 3 -from sub1 -to add3

Speculative execution

Pre-computed result

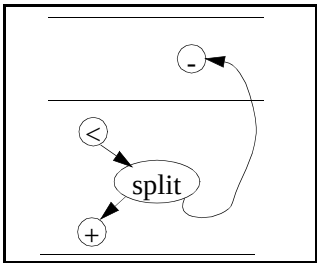
stored into a register

discarded if branch not taken

Default: Turned off

- search space and execution time

bc_shell> bc_enable_speculative_execution



Templates

Precedence and data arcs cannot express maximum allowable duration

Prescheduled sub-design has to be preserved

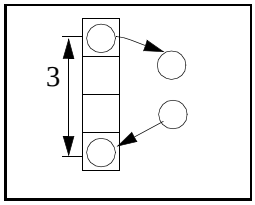
Templates

Collection of operations allowed to move only as a group

Rigid timing relationship between its elements

Slots contain either place holders and/or other nodes

Notice them when impossible schedule



Scheduling

Objective

Minimize hardware cost within user timing constraints

Ex: 2 additions on a single adder if occurring in $\neq$ csteps or mutually exclusive

Minimize cost of registers

Lower(nb registers)= min nb of bits crossing cstep boundaries

Algorithm

```
while all operations not scheduled
 Choose the most important unscheduled
  operation OP
 Assign OP to the most cost effective step
 Mark OP as scheduled
```

Scheduling (cont'd)

Criteria for selecting operations op

- Ready
- highest implementation cost
- mobility

Criteria for selecting the appropriate csetp t

- op must be ready at t
- consumers of op are critical
- Clock cycle must be respected (regarding chaining)
- Resource and/or timing constraints
- Register cost

Scheduling (cont'd)

Additional complexity

- Conditional
- Loops
- Pipelining
- Memory operations

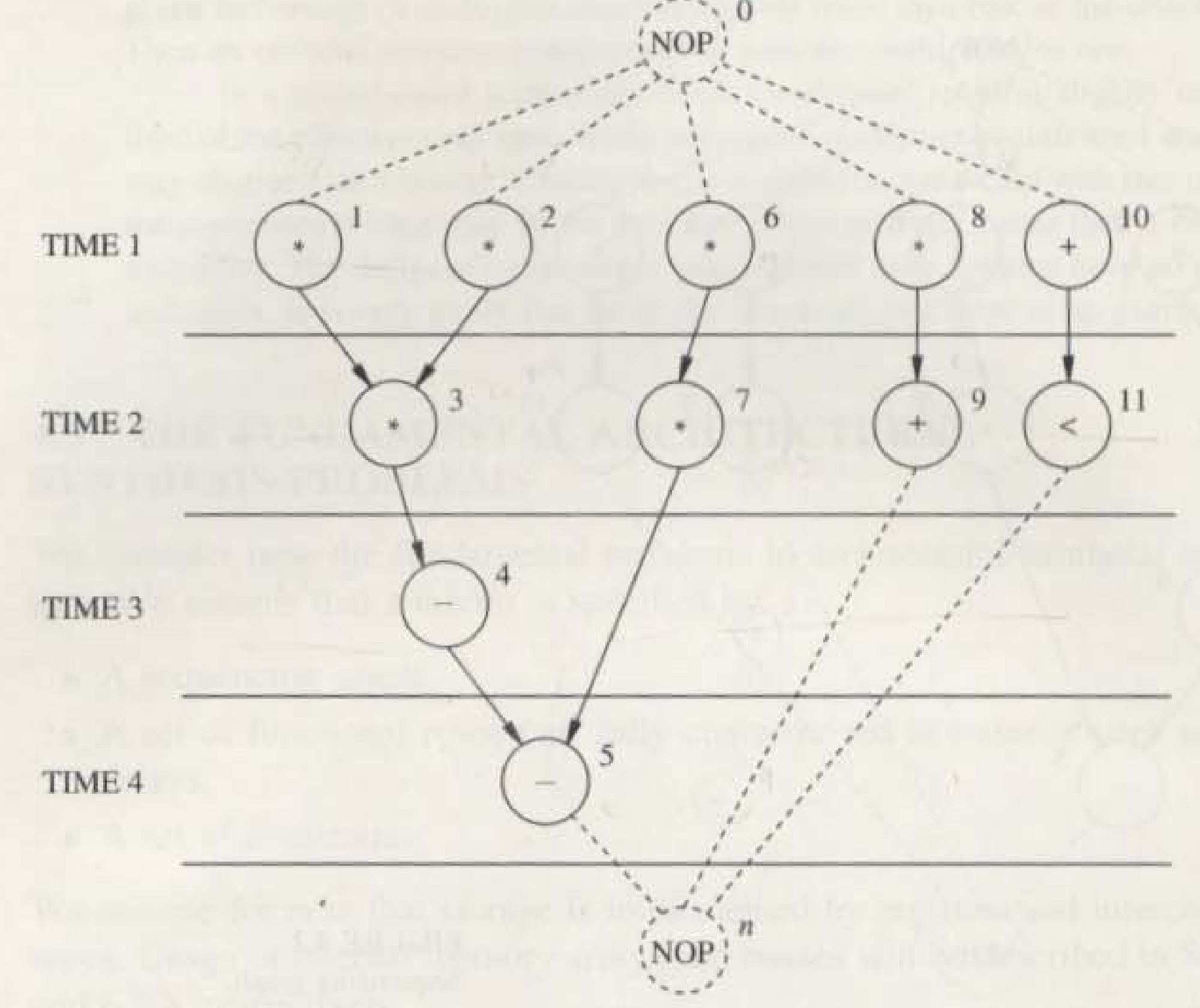
To avoid excessive iterations use achievable constraints

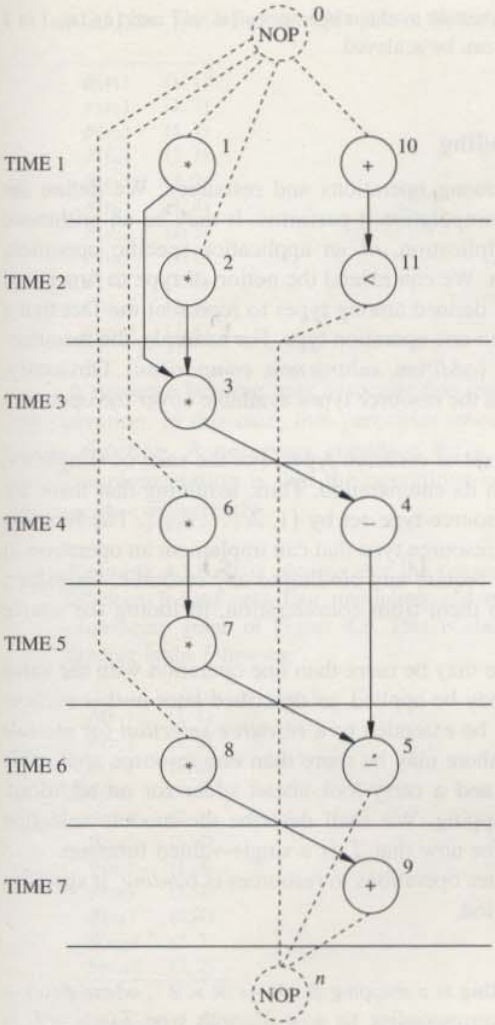
- Iteration are very fast compared to logic level

BC schedules bottom up

- Innermost loops and functions calls first
- Inline each completed level
- Inlined loops encapsulated in templates
- Inconsistencies may appear at higher levels due to templates

```
x(0)= x; y(0)= y; y'(0)= u
y'' + 3xy' + 3y = 0
-----------------------------------------
eqdiff {
  lire (x, y, u, dx, a)
  répéter {
     x1 = x + dx;
     u1 = u – (3 * x * u * dx) – (3 * y * dx);
     y1 = y + u * dx;
     c = x1 > a;
     x = x1; u = u1 ; y = y1;
     }
  jusqu'à (c);
  écrire (y);
}
```


0
NOP
TIME 1
1
2
6
8
10
TIME 2
3
7
9
11
TIME 3
4
TIME 4
5
n
NOP


NOP
0
TIME 1
1
10
TIME 2
2
11
TIME 3
3
TIME 4
6
4
TIME 5
7
TIME 6
8
5
TIME 7
9
NOP
n

Allocation

Operation should be mapped on particular hardware resources

The number of resources is supposed given from the scheduling step

Unit selection and mapping affects both speed and cost

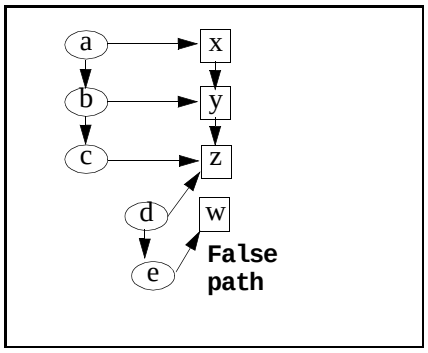
```
Unallocated = operations & values
While unallocated
 choose U
 if not( free(R, time(U)) & Impl(R,U))
 then add new resource R
 assign(U, best (R))
 mark U allocated
```

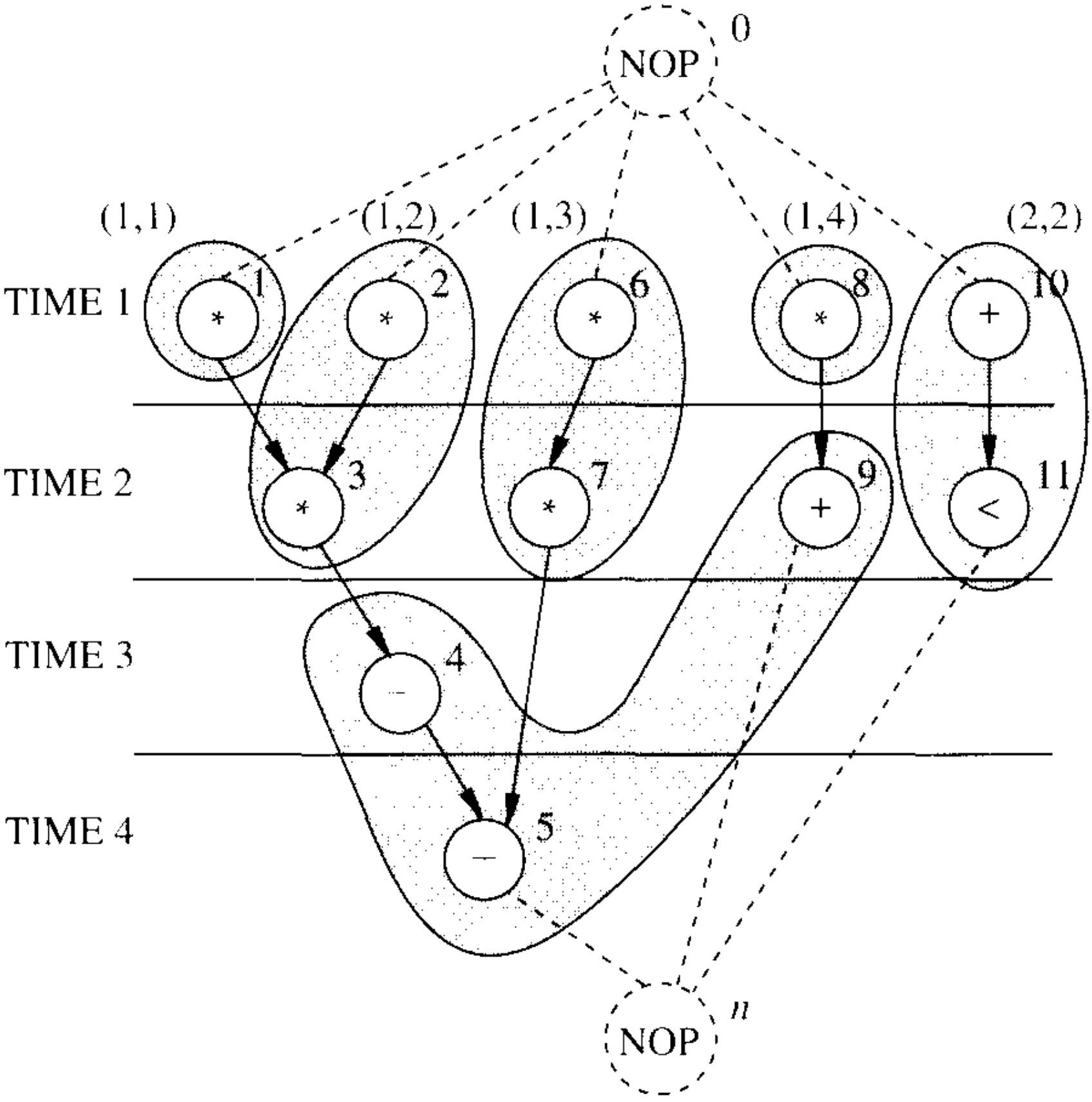

Allocation (cont'd)

Avoid false path otherwise logic synthesis will have hard time

(a, b, c), (d, e) chained operations

Diff. csteps




NOP
0
(1,1)
(1,2)
(1,3)
(1,4)
(2,2)
TIME 1
TIME 2
TIME 3
TIME 4
1
2
3
4
5
6
7
8
9
10
11
*
*
*
*
*
+
+
<
−
−
NOP
n

Allocation criteria

Cost

Allocate the most expensive operations first

Critical path

Operations and operands affecting the clock cycle first

Interconnect

Cluster operations and operands

⇒ Minimize interconnects

⇒ Avoid false paths

```
> set_common_resource op1 op2 op3 -mincount 2
```


Netlisting

Output of BC goes to logic synthesis

Random logic required by the user is instantiated

Register, operators, memories are instantiated according to the allocation step

MUXes, nets, connectivity hardware are constructed to connect the datapath

Whole design connected to signals and ports

Status and control points recorded for later hookup to control FSM

Control FSM

A state graph is constructed

A set of control actions is constructed

Each of these drives a control point

Actions are annotated on the transitions of the state graph

Status points are mapped from the scheduled CDFG

Netlist augmented with Control Unit (CU)

Inputs to CU are status signals

Outputs are connected to the control points

A State Table is constructed

It will serve as input to the FSM compiler

States and csteps

One to one mapping except:

Loop has one more cstep than states

Mutually exclusive loops have disjoint states mapping to the same csteps

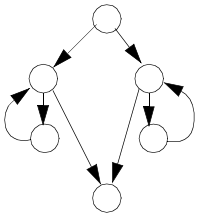
```
while cond loop
 wait until clk'event
  and clk='1';
end loop;
```

BC constraints on loops

Loop-end at least one cstep after loop-begin

Loop exit at least on cstep after cond. evaluation

1+ clock edges inside a loop (O not allowed)



```
if cc then
 L1: while cond loop
   wait until clk'event and
   clk='1';
   wait until clk'event and
   clk='1';
 end loop;
else
 L2: while cond loop
   wait until clk'event and
   clk='1';
```


Invoking the scheduler

schedule command automatically invokes timing (if necessary), scheduling, allocation, netlisting and control unit synthesis.

```
bc_shell> set_cycles 5 -from_begin loop1 -to_end loop1
bc_shell> set_common_resource op1 op2 -min_count 1
bc_shell> schedule -effort med -io_mode super
effort: zero, med, high
```

BC outputs

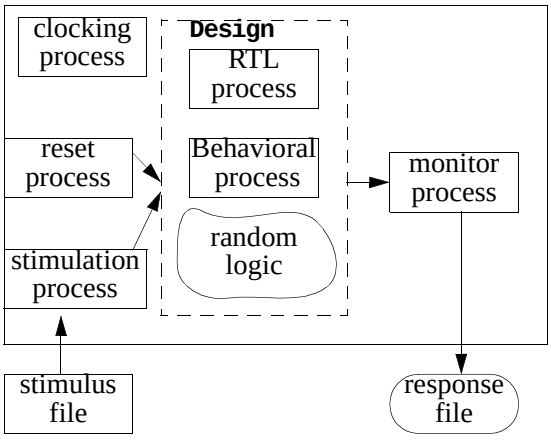
```
bc_shell> report_schedule -operations -variables
bc_shell> write -hierarchy -format vhdl -out mydesign.vhd
compile -map_effort medium
optimize_registers
write -format edif -hierarchy
```

IV. HDL descriptions and semantics

Objectives

VHDL styles for synthesis
Overall structure of models for simulation
BC interpretation of constructs
Main feature of BC
    Simulation and comparison of design before and after synthesis
Design should be tested thoroughly before synthesis
Pre-synthesis simulation faster than post-synth. simulation
    $\Rightarrow$ Test as much as poss. at behav. level
Development of good test benches is very important
It is also very time-consuming
    May be more than the development of the model itself

Pre-synthesis model



Test bench

The design

Must be represented by

A VHDL entity

An associated architecture

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity design is
 port (clk, reset: in std_logic;
 ip: in signed (7 downto 0);
 op: out signed(7 downto 0));
end design;
```

The design (cont'd)

```
architecture behave of design is
-- type, signal, component, ... declarations
begin
 -- component instantiations
 -- concurrent statements
 -- RTL processes
 -- behavioral processes
end behave;
```


Behavioral processes

BC schedules behavioral processes only

Random logic outside processes and RTL processes preserved during scheduling

Multiple behavioral processes scheduled independently

No attempt by BC to maintain synchronicity

User must maintain synchronicity by providing strobes, ready signals, etc.

Synchronization more difficult when non-cycle-fixed I/O is present

Local variables to a process are mapped to registers

- Optimization based on life time

Behavioral processes (cont'd)

No sensitivity list

Variables only visible within the process

```
P1: process
 -- local variables
 variable x: signed (7 downto 0);
 ...
begin
   -- behavioral statements;
   ...
   wait until clk'event and clk ='1';
end process P1;
```


Clock and Reset

BC supports

A single-phase edge-triggered clock

Synchronous or asynchronous resets

Interpretation

Each clock edge forces the process to await the next active clock edge before proceeding

An output write may be forced to fall one cycle after another operation

User may insert any number of clock edges in the model

Edge polarities cannot be mixed inside the same process

Different processes may use different edges, clock nets, and frequencies

Sensitivity lists are not allowed in a behavioral process

Inside a behavioral process only one signal and one polarity can be the argument of any wait statement

Synchronous resets

```
main: process
begin
 reset_loop: loop
  --reset tail
  pc := (others => '0'); sp:= (others =>'1');
  wait until clk'event and clk='1';
  if reset ='1'  then exit reset_loop; end if;
  main_loop: loop
   -- normal mode
   instr := memory(pc);
   wait until clk'event and clk='1';
   if reset ='1'  then exit reset_loop; end if;
   case (instr) is
     when "00100000" => ...
  end loop main_loop;
 end loop reset_loop;
end process main;
```


Synchronous resets (cont'd)

Exit statement after each clock edge

Loop encloses the entire process behavior

Loop begins with reset specific behavior

BC infers a reset if no reset branch is missing and all branches are identical

BC reports well formed resets

A global synchronous reset has been inferred

A reset can be included at bc_shell level

> set_behavioral_reset reset -active high

A reset net or port should be provided

Unused net is deleted during elaboration

⇒ add a dummy port or logic which uses the reset net

Asynchronous resets

Use set_bahavioral_async_reset or

If needed for pre-synthesis simulation

Readability ➘

```
wait until (clk'event and clk ='1')
     --synopsys synthesis_off
     or ( reset'event and reset ='1')
     --synopsys synthesis_on
if reset ='1'  then
  exit reset_loop;
end if;
```


I/O Operations

```
entity design is
 port (clk, reset: in std_logic;
 ip: in signed (7 downto 0);
 op: out signed(7 downto 0));
end design;

architecture behave of design is
 signal sig: signed(7 downto 0);
begin
 P1: process
  variable v1,v2: signed (7 downto 0);
 begin
  wait until clk'event and clk ='1';
  v1 := ip; --read
  v2:= ip; -- different read
  wait until clk'event and clk ='1';
  sig <= v1; -- write
  wait until clk'event and clk ='1';
  op <= v2 + sig   -- read and write
  wait until clk'event and clk ='1';
 end process P1;
end behave;
```

I/O Operations

I/O R/W inferred from references to architecture signals or entity ports

Note different reads in "the same cycle"

Cycle stretched in 2 I/O modes

If one read wanted then re-use v1

BC registers output ports and written signals

New value appears at the next cycle

Avoid "<=" except for communicating with outside

I/O Operations (cont'd)

R/W signals may serve as milestones in a very complex design

Constrain the schedule

Reduce the search space

Make testing easier

BC assumes registered inputs

Flow of Control

Most constructs supported

For, while , infinite loops

If-then-elsif-else, case statements

Functions, procedures

Next, exit

Associated with a reset, or

Affect the immediate enclosing loop

Fixed bound FOR loops

Unrolled by default at elaboration time

Eliminates hardware evaluating the conditional

Allows writing loops containing no clock statements

Allows simultaneous scheduling of operations outside the loop and operations from different iterations

To force keeping complex loops rolled

```
attribute dont_unroll: boolean;
attribute  dont_unroll of loop_C: label is true;
...
loop_C: for i in 0 to 1000 loop ...
```

General loops

Not unrolled

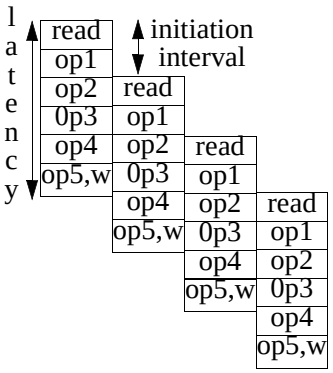
Infinite loops

While loops and loops with “dynamic” range

Loops with explicit conditional exit

Pipelined loops

```
loop
 a:= inputport;
 wait until clk’event and clk =’1’;
 b:= op1(a);
 wait until clk’event and clk =’1’;
 c:=  op2(b);
 wait until clk’event and clk =’1’;
 d:= op3(c)
 wait until clk’event and clk =’1’;
 e:= op4(d);
 wait until clk’event and clk =’1’;
 outputport <= op5(e);
 wait until clk’event and clk =’1’;
end loop;
```



latency: multiple of II
II=2; L=6
Throughput = 1/II = 0.5

Pipelined loops

Previous example hypothesis

No chaining possible

Operations so diff. they cannot share the same hardware.

Before pipelining

1/6 resource utilization

1/6 throughput

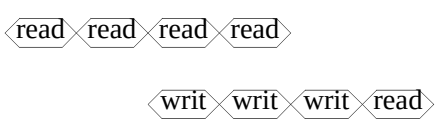
After pipelining

1/2 utilization

1/2 throughput

Pipelined loops and Fixed I/O mode

```
ppl: while (cond) loop
 u := inp; --read
 x := x*a + u*b;
 output <= transport x after 20 ns -- 2cycles
 wait until clk'event and clk ='1';
end loop ppl;
wait until clk'event and clk ='1'; -- purge pipeline
wait until clk'event and clk ='1';
wait until clk'event and clk ='1';
output <= in_order_output
```



Other I/O modes

Implicit declaration of a pipeline (as in fixed mode) is not possible nor necessary

> pipeline_loop ppl -initiation 1 -latency 3

Regardless of I/O mode re-using the outputs cannot be too close to the end of the pipelined loop

No exit later than II+1 to avoid explosion of states

Rolled loops cannot be nested inside pipelined loops

BC cannot determine statically the concurrency between iterations

Memory inference

Memories specified using arrays

Memories consist of words

BC schedules accesses and controls ports

RAM accesses are synthetic

BC makes conservative assumptions about address conflicts

An address conflict occurs: two accesses to same mem. one access is a write

```
M(14) := 5;      M(14) := 5;
x := M(14);      x := M(13);
True conflict    False conflict
```

BC does not distinguish between false and true conflicts

Override BC deduction:

> ignore_memory_precedences -from op1 to op2

Memory code

```
architecture beh of mem_dsg is
 subtype resource is integer;
 attribute variables: string;
 attribute map_to_module: string;
 type mem_type is array (0 to 15) of signed(7 downto
 0);
begin
 behavP: process
  constant Mem1: resource:=0; --physical memory
  attribute variables of Mem1: constant is “M”;
  attribute  map_to_module of Mem1: constant is
  “DW03_ram1_s_d”;
  variable M: mem_type;   --logical memory
 begin
  ...
  M(12) := “00101100”; -- mem. w.
  outport <= M(2*x);  -- mem. r.
```


Memory timing

Normal loops

Iterations are insulated

R/W of one iteration strictly precede R/W of next iteration

Pipelined loop

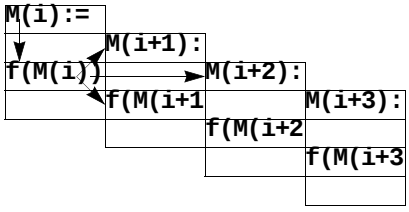
Iterations co-exist
⇒ inter-iteration conflicts appear

These conflicts may be false

> ignore memory_loop_precedences {op1 op2}

Memory timing (cont'd)

```
for (i in 0 to Msize) loop
 M(i) := inport;
 outport <= transport (f(M(i)) after 20 ns;
 wait until clk'event and clk='1';
end loop;
```



II cannot 1 or 2 due to false memory conflict, it may be 3

Other memory considerations

RAM operations

appear just as array references

but may be multi-cycle operations

use registers if poss.

Declaration of memory forgotten or misspelled ⇒

Array of words becomes a large register

Busses used for reads

MUXes uses for writes

Area and logic optimization become impractical

If memory contains records

Accessing a single field means accessing the whole record

⇒ partition the memory in different memories

Synthetic components

Component synthesized on the fly when needed

Ex: adders, multipliers ...
Encapsulated in DesignWare libraries
Sharable resources during allocation

≠ modules, each module has ≠ implementations

adder module, add/sub module, ≠ carry chain implementations

DesignWare developer

Function or procedure used in more than one place

Is not in the DW lib.

Wish the hardware implementation sharable

ex: MAC op. for DSP with ≠ implementations

repeated random logic modules

Define a function instead of code

```
if (cond) then
  x := d1;
else
  x:=d2;
end if;
```

Then use the map_to_module pragma

⇒ use DW module instead of inlining the function

simplify the FSM by moving parts to Data Path

Size of the FSM exponential in number of inputs

Preserved functions

By default, BC inlines subprograms during elaboration

To prevent inlining:

```
function fid (...) is
  -- synopsys preserve_function
```

Inlining controlled at subpgm definition

Equivalent to a DW part with some restrictions

No signal R/W

No sequential DW parts

No clock edge statements

No rolled loops

No unconstrained types

No multiple implementations

Cannot be used in an RTL process

Pipelined components

Comb. logic as a synth. comp. may have excessive delay.

1. Lengthen the clock cycle

   Bad solution

   Increases chaining while diminishing sharing

2. Allow multi-cycle operations

   Latency penalty

   Registered inputs

3. Pipelining

   May be obtained by retiming (optimize_registers)

   Some DW components are pipelined

   Use DW developer

   Use a directive

   > set_pipeline_stages {op1 op2} -fixed_stages 3

V. I/O modes

I/O modes

Three I/O modes

three different interpretations of HDL semantics

Modes define equivalence between the pre-synthesis and post-synthesis models

Pre and post synthesis designs perform the same operation at the same time on their inputs

Very strict, rules out scheduling

Fixed I/O mode:

The I/O behavior is always the same

A communication protocol working with the model will work with the synthesized design

Test bench will work with both

Strict discipline, if computation two long, BC will exit with error

I/O modes (cont'd)

Superstate mode

I/O operations order is preserved, time may be stretched

Input and design distinguishable only by counting clock cycles

Test bench preserved if independent of number of clock edges

A good balance between optimization and verification

Free floating mode

I/O operation are freely shifted in time

Allows maximum optimization

Difficult verification

Cycle-Fixed Mode

Any scheduled mode has a fixed counterpart

A timing diagram not achievable in fixed mode

⇒ Not achievable in any mode

Source can talk correctly to its environment

⇒ Synthesized process will

Source should be written allowing BC synthesis

Without ± any clock cycle

I/O timing preserved except for reset

Resets not needed in simulation

1+ cycles needed to startup the FSM in synthesized design

BC always registers the process outputs

⇒ 1 cycle skew with simulation

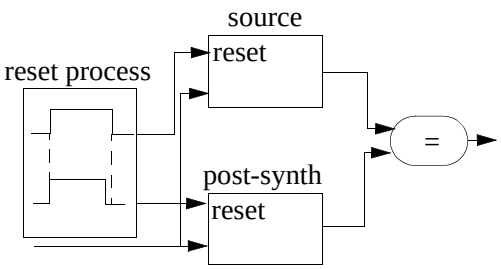
Cycle-Fixed Mode
(Test bench)

Provide two reset pulses

Reset source one cycle longer

Other signals from the test bench should not transition exactly on the clock edge

Otherwise setup hold violations



Fixed Mode rules
(Straight line code)

Source should be written allowing BC synthesis without ± any clock cycle

BC fails if a series of operations cannot fit in the allocated time

Its decision takes into account

- Chaining
- Sequential operations
- Multicycle operations
- Manual constraints

A multicycle operation can *only* be chained with an output operation

Be careful about muticycle operations

- Not obvious by simple inspection
- Especially memory operations

Fixed Mode rules (Loops)

Loop boundaries are not free to be rescheduled

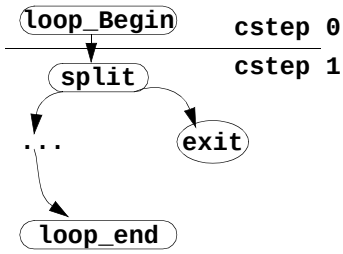
A loop is mapped to 2+ csteps

Loop test is performed inside the cycle of the loop

⇒ No transition goes past the loop

⇒ Must be a clock edge between loop test and any succeeding output

```
while (not ready) loop
 wait until clk'event and clk='1';
end loop;
-- Illegal: no wait
outdata <= data;
```



Loops in fixed mode

Mental representation of a while loop

```
free_loop: loop
 if ready then
  wait until clk'event and clk='1';
  exit free_loop;
 end if;
 wait until clk'event and clk='1';
end loop;

  dataout <= data;
```

Nested loops and FM

```
while (not done) loop
 -- A: wait needed
 while (not ready) loop
   wait until clk'event and clk='1';
 end loop;
 --B: wait needed
end loop;
```

A: otherwise two condition must be tested in the same cycle

B: otherwise one branch of nested loop without a clock edge

Successive loops and FM

```
while (not done) loop
 wait until clk'event and clk='1';
end loop;
-- wait needed
while (not ready) loop
 wait until clk'event and clk='1';
end loop;
```

Complex loop conditions

Complex conditions may take more than 1 cycle

```
while (x*inport1 < y-inport2) ...
```

Two reads locked to the same cycle

Operations are performed: 2 cycles

Extra cycle should be taken into account in the subsequent code

Superstate-Fixed Mode

Properties

Preserves the I/O ordering but

Not necessarily the number of clock edges between I/O operations

Latency of the design may change by user commands without changing the HDL

```
> pipeline_loop main_loop -latency 16 -initiation 4
```

A superstate is the interval between 2 source clock edges.

BC is allowed to add clock edges to a superstate

Equivalence

Any I/O write will take place in the last cycle of the superstate

An I/O read can take place in any cycle of the superstate

Superstate-Fixed Mode (Implications)

Any 2 writes happening in the same superstate must be simultaneous

Input data must be held stable during a superstate

I/O protocols must handle extra delays possibly added by BC

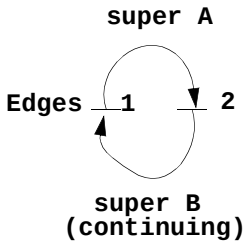
⇒ handshaking is a candidate protocol

Superstate Rules (continuing superstate)

The first superstate of a loop contains any I/O

⇒ no superstate containing a loop continue may contain an I/O write

```
while (not ready) loop
 tmp := inport ;  --read
 -- edge 1
 wait until clk'event and clk='1';
 -- edge 2
 wait until clk'event and clk='1';
 outport <= data; --illegal
  end loop;
```



Superstate Rules
(separating write orders)

There must be a clock edge between a write and the beginning of a loop whose first superstate contains a write operation

```
this_port <= some;
-- must have a wait
loop
 that_port <= any ;
 wait until clk'event and clk='1';
end loop;
```

Ex: 1st superstate starts outside of the loop

⇒ outside write has to migrate inside the loop (contradiction)

Superstate Rules
(Conditional superstate)

A write can never precede a conditional superstate boundary if any I/O operation succeeds the boundary

```
thisport <= some;
-- must have a wait
while strobe loop
 wait until clk'event and clk='1';
end loop;
reg1 := thatport;
```

Superstate Rules
(Escaping from the loop)

No I/O write can occur between the exit and the last clock edge before the exit

```
busy: while strobe loop
 wait until clk'event and clk='1';
 thisport <= some;
   if (interrupt) then exit busy;
 end if;
end loop;
```


Free-Floating Mode

I/O operations are free to float with respect to one another

Operations on single port are partially ordered

Series of reads can be permuted

No ordering between operations on different ports

Data precedences and constraints respected

Deleting or adding clock edges permitted

If two signals are logically bound then express it using manual constraints

VI. Explicit Directives and Constraints

Labeling
(Default naming)

If "+" falls in line 35 the default name is

P1/outloop/innerloop/add_35

If > one "+" then add_35_1, add_35_2

Default names created for unlabeled loops

If unrolled loop: add_35_i_3 for iteration 3

> find -hier cell > names.txt

Drawback: cell names change if source edited

```
P1: process
 outloop: loop
  innerloop: loop
   x := a+b;
  end loop;
 end loop;
end process;
```

Labeling
(user naming)

Use pragma

New name not sensitive to editing is

P1/outloop/innerloop/alu

Limitations

Ambiguity when many operations on the same line,

Not applicable to I/O and memory operations loop boundaries

```
...
 x := a+b; -- synopsys label alu
...
```


Labeling
(improved naming)

Labeling lines

P1/outloop/innerloop/add_thisline

If multiple operation: names generated from left to right

```
...
x := a+b; -- synopsys line_label thisline
...
```

Scheduling Constraints

> preschedule p2/res_loop/main/sub_107 4

Forces the named operation into a particular cstep
The cstep is relative to the beginning of the enclosing hierarchical context
sub_107 will be put in the 5th cstep of loop main

> set_cycles 3 -from op1 -to op2

op2 must start exactly 3 cycles after op2 started

> set_cycles 3 -from_begin loop4 -to_end loop4

> set_min_cycles 5 -from_end loop4 -to_begin loop6

Scheduling Constraints (cont'd)

chain_operations equivalent to set_cycles 0
dont_chain_operations equivalent to set_min_cycles 1
remove_scheduling_constraints removes all explicit constraints

> set_common_resource op1 op2 op3 -min_count 2

Shell Variables

> bc_enable_chaining = false

Globally turns off chaining of synthetic operations.

Use more specific constraints.

true by default

bc_enable_multi_cycle: true by default

bc_enable_speculative_execution: false by default

Shell Variables (cont'd)

bc_fsm_coding_style

- one_hot
- counter_style
- two_hot
- use_fsm_compiler (default)

reset_clears_all_bc_registers when set to true

Clear pins of all registers connected to the reset net

With set_behavioral_async_reset provides asynch reset to all registers

Shell Commands

set_margin controls the margin

allowed for control and muxing delays

when timing the design before scheduling

> register_control -inputs -outputs

Forces registers on inputs and/or outputs of the control FSM

May improve the cycle time but

May increase latency if conditionals on the critical path

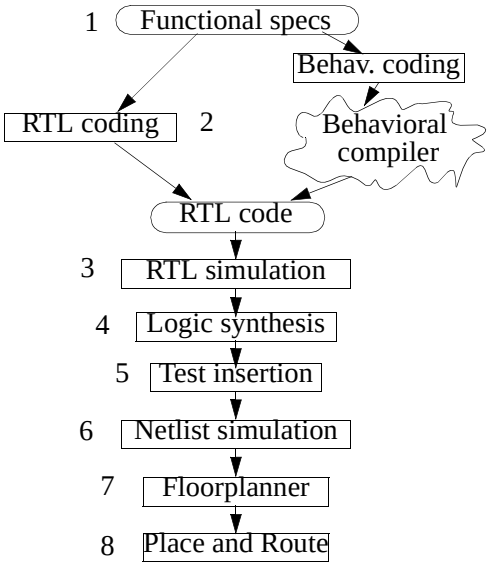
set_stall_pin

Used to stop the design for some external event to occur

Equivalent to a gated clock

VII. RTL Design Methodology

RTL Design flow



RTL Design flow

It is an iterative process

If simulation not satisfactory, goto RTL code

If timing requirements of the clock not met after synthesis

- modify code
- change synthesis strategy
- hack the netlist

After P&R

Back-annotate real delay values

Perform in place optimization to meet routing delays

Design refinement

Block diagram of ASIC created after step 1

HDL coding of each block

Style of coding important for synthesis

Knowledge internals
⇒ write good synth. code

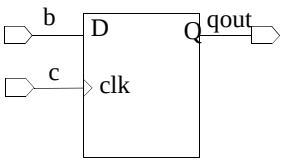
Hierarchy based on func. specs.
⇒ critical path may traverse hierarchy boundaries

Best results when critical path in one block

Ensure registered output blocks
⇒ avoid complicated timing budgeting

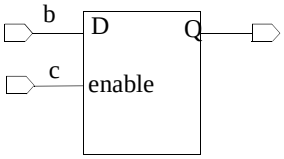
HDL FF Code

```
entity comp is
 port (b, c: in bit; qout: out bit);
end comp;
architecture FF of comp is
begin
 P1: process
 begin
  wait until c'event and c ='1';
  qout <= b;
 end process P1;
end FF;
```



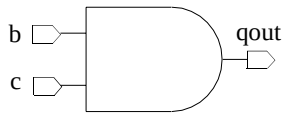
HDL latch Code

```
entity comp is
 port (b, c: in bit; qout: out bit);
end comp;
architecture latch of comp is
begin
 P1: process (b, c)
 begin
  if (c ='1') then qout <= b;
   end if;
 end process P1;
end latch;
```



HDL AND Code

```
entity comp is
 port (b, c: in bit; qout: out bit);
end comp;
architecture and2 of comp is
begin
 P1: process (b, c)
 begin
  if (c ='1') then qout <= b;
  else  qout <= '0';
  end if;
 end process P1;
end and2;
```



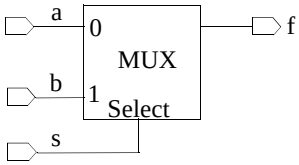
MUX inference

Often gates are inferred instead of MUXes

map_to_entity pragma forces mapping to MUXes or

Function calls or

Instantiating MUXes from Synopsys generic library (gtech.db) and assigning map_only attribute



MUX modeling

```
entity comp is
 port (a,b, s: in bit; f: out bit);
end comp;

architecture mux of comp is
begin
 P1: process (a,b, s)
 begin
  case s is
   when '0'  => f<=a;
   when '1'  => f<=b;
  end case;
 end process P1;
end mux;
```


Synthesized gate-level netlist simulation

VHDL simulation models of technology library cells

Unit Delay Structural Model (UDSM)

Comb. cells delay = 1ns

Seq. cells delay = 2ns

Full-Timing Structural Model (FTSM)

Transport wire delays

Pin-to-pin delays

Zero delays functional networks

Timing constraints violations reported as warnings

Netlist simulation (cont'd)

Full-Timing Behavioral Model (FTBM)

Transport wire delays

Pin-to-pin delays

Very detailed timing verification

Full-Timing optimized Gate-level Model (FTGM)

Transport wire delays

Pin-to-pin delays

Warnings + handling X values

Timing constraints violations reported as warnings

Logic synthesis

Transform RTL HDL to gates

Optimize by selecting the optimal combination of technology library cells

Simulation of commercial ASICs

vdlib.vhd.E : encrypted, contains simulation models with timing delays

vdlib_components.vhd: package, declarations for all the cells of ASIC vendor library

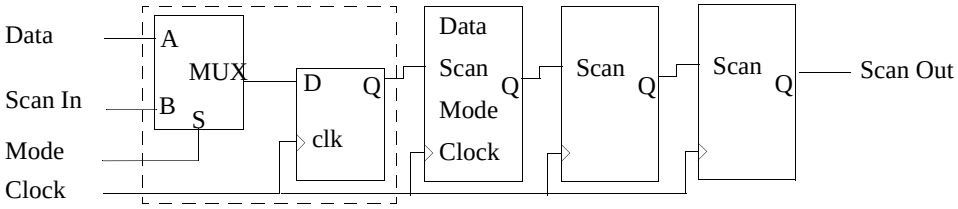
If source available (.lib) the user can control the type of the model by setting the dc_shell variable vhdllib_architecture

write_lib -f vhdl

Design for Testability

Cost of testing important part of the total cost

Scan Design techniques: popular DFT technique



Full Scan ⇒ combinational ATPG

Partial Scan ⇒ sequential ATPG

TC automatically replaces sequential cells by scan cells

TC generates test patterns and computes fault coverage (single s-a-0/1 model)

Design Re-use

Achieves fast turnaround on complex designs

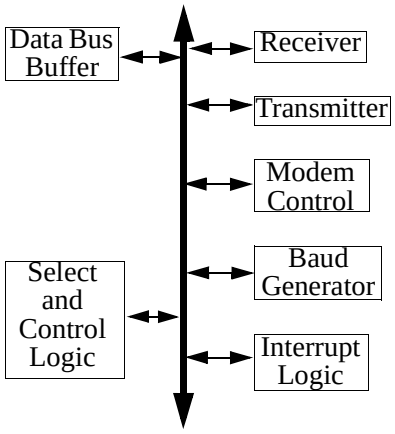
DesignWare is a mechanism to build a library for re-usable components

Generic GTECH Library
- Source read in DC converted to a netlist of GTECH components and inferred DW parts
- gtech.db contains basic logic gates, flip flops, half adder and a full adder

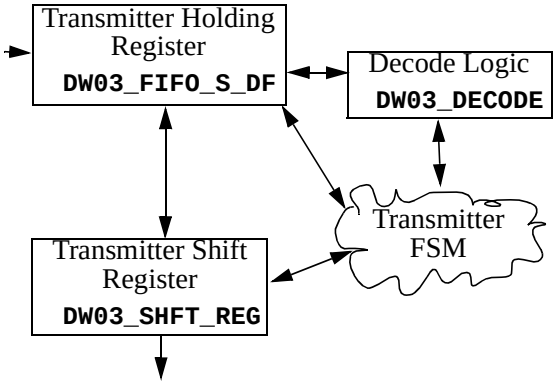
DW libraries
- Standard, ALU, Maths, Sequential, Data Integrity, Control Logic and DSP
- adders, counters, comparators, decoders
- Parts are parametrizable, synthesizable, testable, technology independent
- Parts have simulation models
- When used, implementation selection, arith. optimization and resource sharing are on

Users can create new DW libraries
- Effective mechanism to infer structures that DC would not

Designing with DW Components



Hierarchical view of UART



Transmitter Block

FPGA Synthesis

User programmable IC: set of logic blocks that can be connected using routing resources

Interconnect: wires of diff. lengths and programmable switches

Easy to configure by the user

Implement logic circuits at relatively low cost with a fast turnaround

Hardware emulation: use programmable hardware as a prototype of an IC design

Rapid growth and density of FPGAs $\Rightarrow$ need for synthesis tools

FPGA Compiler for Synopsys:

Map HDL descriptions to logic blocks and provide configuration of switches

Links to layout

Advent of sub-micron tech. ⇒ net delays become significant

while gate delays decrease wire delays increase due to capacitances

Accurate wire loads and physical hierarchy become crucial to synthesis tools

Synopsys Floorplan Manager transfers information between back-end tools and DC

Formats for transfer:

- Standard Delay Format (SDF)
- Physical Data Exchange Format (PDEF)
- Synopsys set_load script

DC and DA environments

Design Analyzer (DA): graphical front end of Synopsys environment
- Used to view schematics and their critical path

dc_shell (DC): command line interface for RTL synthesis
- Can be invoked from DA command window
- (setup -> Command Window)

Startup files
- DC reads .synopsys_dc.setup when invoked
- Recommendation: keep .synopsys_dc.setup in current working directory
- ⇒ design specific variables specified without affecting other designs

DC and DA environments (cont'd)

Ex:

```
search_path = search_path+{".", "./lib","./vhdl","./script"}
target_library = {target.db}
link_library = {link.db}
symbol_library = {symbol.db}
```

DA (setup ->Defaults) should indicate the specified libraries

Specifying libraries in .synopsys_dc.setup is permanent compared to specifying them in DA

Command list <variable_name> gives the current value of the variable

```
> list target_library
```


Target, Link, and Symbol Libraries

Target library

ASIC vendor library

Used to generate a netlist for the design described in HDL

Link library

Used when the design is already a netlist or

When the source instantiates technology library cells

Symbol libraries

Contain pictorial representation of library cells

```
> compare_lib <target_library> <symbol_library>
```

Shows any differences between the two libraries

Libraries generation

Libraries generated from ASIC files (.lib, .slib) files

By Synopsys Library Compiler

Produce (.db, .sdb) libraries

```
> read_lib my_lib.lib
> write_lib my_lib.db
```

```
> read_lib my_lib.slib
> write_lib my_lib.sdb
```

VIII. VHDL RTL SEMANTICS

Types, signals and variables

Use `std_logic` for ports

$\Rightarrow$ no conversion functions needed

`std_logic` $\in$ `std_logic_1164` package

Avoid using mode `buffer`: must percolate through hierarchy

Variables

+ Updated immediately
+ Faster simulation
- May mask glitches

Signals

Need δ-time

Signals used by a process $\notin$ sensitivity list

$\Rightarrow$ RTL $\neq$ gate simulation

Buffer mode modeling

```
entity buf is
 port (a,b: in std_logic, c:out std_logic)
end buf;

architecture test of buf is
 signal c_int: std_logic;
begin
 process
 begin
  ...
  c_int <=  ...
  ...
 end process;
 c <= c_int;
end test;
```


STD_LOGIC

```
TYPE std_ulogic IS (  'U',  -- Uninitialized
                      'X',  -- Forcing  Unknown
                      '0',  -- Forcing  0
                      '1',  -- Forcing  1
                      'Z',  -- High Impedance
                      'W',  -- Weak     Unknown
                      'L',  -- Weak     0
                      'H',  -- Weak     1
                      '-'   -- Don't care
                    );

attribute ENUM_ENCODING of std_ulogic : type is "U D 0 1 Z D 0 1 D";
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

SUBTYPE std_logic IS resolved std_ulogic;
```

Arithmetic

```
library IEEE;
use IEEE.std_logic_1164.all;

package std_logic_arith is

    type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
    type SIGNED is array (NATURAL range <>) of STD_LOGIC;
    subtype SMALL_INT is INTEGER range 0 to 1;

    function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
...
    function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "+"(L: INTEGER; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
...
end Std_logic_arith;
```


Unwanted latches

Ensure

All signals initialized

Case and if stat. completely defined

```
library ieee;
use ieee.std_logic_1164.all;

entity qst is
 port
  (clk: in std_logic;
  d: in std_logic_vector (1
  downto 0);
  q: out std_logic_vector (1
  downto 0));
end qst;
```

```
architecture unwanted of qst is
begin
 process (clk, d)
 begin
  if clk = '1' then q <= d;
  -- incomplete no else
  end if;
 end process;
end unwanted
```

Asynchronous reset

```
entity FF is
 port (x,clk, rst: in bit; z:out bit);
end FF;

architecture async of FF is
begin
 process (clk,rst);
   variable ST: ...;
 begin
  if rst = '0' then
    ST := S0; z <= '0';
  elsif clk'event and clk ='1' then
     case ST is
     ...
     end case;
   end if;
 end process;
end async;
```


Synchronous reset

```
entity FF is
 port (x,clk, rst: in bit; z:out bit);
end FF;

architecture sync of FF is
begin
 process
   variable ST: ...;
 begin
  wait until clk'event and clk ='1';
  if rst = '0' then
    ST := S0; z <= '0';
  else
     case ST is
     ...
     end case;
   end if;
 end process;
end sync;
```

VHDL specifics

Case insensitive

Case statement

Mutually exclusive branches

Exhaustive

Sign interpretation

Depends on data types and associated operations

TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;

std_logic_signed, _unsigned: packages for operations on std_logic_vector

VHDL specifics (cont'd)

I/O modes

in, out, inout, buffer

Avoid buffer

inout: port read& written otherwise use internal signals or variables

Multiple drivers

std_logic is a resolved data-type

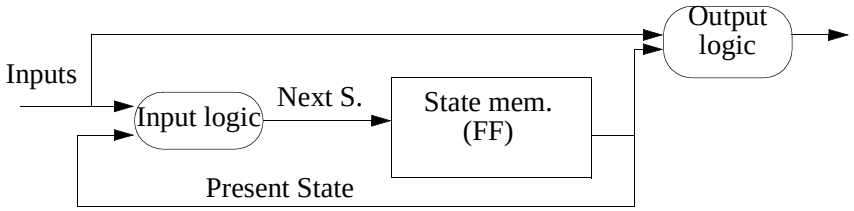
Components

declared

configured

instantiated

Finite state machines



Mealy machine

State encoding

Default

n FF : up to 2^n states

One hot encoding

1 state ↔ 1 FF

Larger area

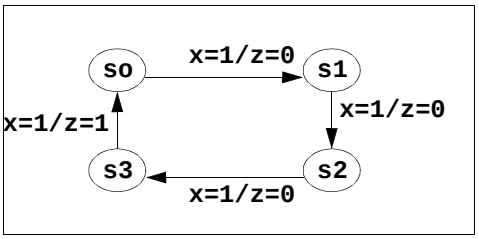
No decoding

Fastest

Gray

HDL description of a state machine

```
package states is
 type state is (s0, s1, s2, s3);
end states;

use work.states.all;
entity ET is
 port (x, clk: in bit;
       z: out bit);
end ET;
```




```
architecture One of ET is
 signal st: state;
begin
 process
 begin
  wait until clk'event and clk ='1';
  if x='0' then z <= '0';
  else
   case st is
     when s0 => st <= s1; z <='0';
     when s1 => st <= s2; z <='0';
     when s2 => st <= s3; z <='0';
     when s3 => st <= s0; z <='1';
   end case;
  end if;
 end process;
end One; -- registered outputs
```


Recommended style

```
architecture Rec of ET is
  signal currentS, nextS: state;
  attribute state_vector: string;
  attribute state_vector of Rec: architecture is "currentS";
begin
  COMB: process( currentS, X)
  begin
   case currentS is
     when s0 => if x = '0' then z <= '0'; nextS <= s0;
                else           z <= '0'; nextS <= s1; end if;...
   end case
  end process; -- Outputs not registered

  SYNC: process
  begin
    wait until clk'event and clk = '1'
    currentS <= nextS;
  end process;
end Rec;
```

Enumerated types and encoding

Default encoding

0, 1, ...

Minimum number of bits

Explicit encoding

```
architecture Rec of ET is
  type state is (s0, s1, s2, s3);
  attribute enum_encoding : string;
  attribute enum_encoding of state: type is “000 110 111 101”;
  signal currentS, nextS: state;
begin
  COMB: process( currentS, X)
  ...
  SYNC: process
  ...
end Rec;
```


General description of FSM

```
package states is
 type state is (s0, s1, s2, s3);
 attribute enum_encoding : string;
 attribute enum_encoding of state: type is “0001 0100 0010 0001”;
end states;

use work.states.all;
entity ET is port (x, clk: in bit; z: out bit);
end ET;

architecture Rec of ET is
  signal currentS, nextS: state;
  attribute state_vector: string;
  attribute state_vector of Rec: architecture is “currentS”;
begin
  COMB: process( currentS, X)...
  SYNC: process...
end Rec;
```

Guidelines for FSM coding

Only *input* or *output* ports

Separate machines == separate designs

State FF driven by same clock

`others` clause ensures *fail-safe* behavior

fail-safe behavior

```
architecture One of ET
 type state is (s0, s1, s2);
 signal st: state
begin
 process
 begin
  wait until clk'event and clk ='1'
  if x='0' then z <= '0';
  else
   case st is
     when s0 => st <= s1; z <='0';
     when s1 => st <= s2; z <='0';
     when s2 => st <= s3; z <='0';
     when others => st <= s0; z <='1';
   end case;
 end process;
 end One;
```

Memories

❶Not synthesized by DC ❷Instantiated as black boxes ❸HDL descr. for simulation

```
library IEEE;
use std_logic_1164.all;
use std_logic_unsigned.all;

entity ram_vhd is
 generic (width: natural :=8
          depth: natural :=16;
          addW: natural:=4);
 port (addr: in std_logic_vector(addW-1 downto 0);
       datain: in std_logic_vector(width-1 downto 0);
       dataout: out std_logic_vector(width-1 downto 0);
       rw,clk: in std_logic);
end ram_vhd;
```


Memory behavior

```
architecture behv of ram_vhd is
  subtype wtype is std_logic_vector(width-1 downto 0);
  type mem_type is array(depth-1 downto 0) of wtype;
  signal memory:mem_type;
begin
  process
  begin
    wait until clk='1' and clk'event;
    if (rw='0') then memory(conv_integer(addr)) <= datain; end if;
  end process;
  process(rw,addr)
  begin
    if (rw='1') then dataout <= memory(conv_integer(addr));
                else dataout <= wtype'(others =>'Z');
    end if;
  end process;
end behv;
```

Barrel shifter

```
library IEEE; use std_logic_1164.all, std_logic_unsigned.all;
entity bs_vhd is
 port (datain: in std_logic_vector(31 downto 0);
       direct: in std_logic;
       count: in std_logic_vector(4 downto 0);
       dataout: out std_logic_vector(31 downto 0));
end bs_vhd;
architecture behv of bs_vhd is
 function b_shift (din: in std_logic_vector(31 downto 0);
  dir:in std_logic; cnt: in std_logic_vector(4 downto 0)
  return std_logic_vector is
  begin
   if (dir ='1')
    then return std_logic_vector((SHR(unsigned(din),unsigned(cnt))));
    else return std_logic_vector((SHL(unsigned(din),unsigned(cnt))));
   end if;
  end b_shift;
begin
 dataout <= b_shift(datain,direct,count);
end behv;
```


Multi-bit register

```
library IEEE; use std_logic_1164.all;
entity reg_vhd is
 generic (width: natural:=8);
 port (r: in std_logic_vector(width-1 downto 0);
       clk,ena,rst: in std_logic;
       data: out std_logic_vector(width-1 downto 0));
end reg_vhd;

architecture behv of reg_vhd is
  signal gclk: std_logic;
begin
  gclk <= clk and ena;
  process(rst,gclk)
  begin
    if (rst = '0') then data <= (others=>'0');
    elsif gclk'event and gclk='1' then data <= r;
    end if;
  end process;
end behv;
```

IX. Methodology for RTL synthesis

Objectives

How to get the best results
Commonly used DC commands
Methodology to optimize a design
General guidelines

Synthesis constraints

Design Rule constraints

- Fanout
- Transition
- Capacitance

Optimization constraints

- Speed
  - set_input_delay
  - set_output_delay
  - max_delay
  - **`create_clock`**
- Area

Design rule constraints

Imposed by the techn. target lib.

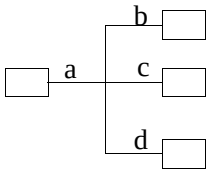
max_fanout

> get_attribute find(pin, “lsi_10k/OR2/A”) fanout_load

Warning: Attribute ‘fanout_load’ does not exist on port ‘A’

> get_attribute lsi_10k default_fanout_load

{1.000000}



$$\sum_{b,\,c,\,d} fanout\_load(i) \leq max\_\text{fanout}(a)$$

DRC

max_transition

Longest time 0-1, 1-0

Specific to a net / whole design

Related to RC time

More restrictive (techn. lib, user)

max_capacitance

Direct control on capacitance

Can be used with max_transition

Violations reported

max_fanout ,max_capacitance $\Rightarrow$ control buffering

$$maxcapa(drivingpin) \geq \sum_{drivenpins} capa(i)$$

Related commands

set_max_transition <value> <design_name/port_name>

set_max_fanout <value> <design_name/port_name>

set_max_capacitance <value> <design_name/port_name>

Optimization constraints

Speed & area constraints by the user

Timing $>_{\text{priority}}$ area

Synch. paths constrained by specifying all clocks

`set_max/min_delay` to specify point to point asynch. constraints

Commands

create_clock

set_input_delay

set_output_delay

set_driving_cell

set_load

set_max_area

Cost functions

Importance ↘

Max delay

Min delay

Max power

Max area

Others

Non-respected setup requir. of a seq. element $\Rightarrow$ violation

Path group = paths constrained by a same clock

Weights attached to $\neq$ path groups

Min indep of groups = worst min

Max power for ECL only

Area optimization performed only if specified

Optimization is an iterative process

Clock specification

Define each clock by `create_clock`

Clock trees must be hand instantiated

Use `set_dont_touch_network` to prevent buffering clock trees

DC considers clock delay network ideal, even gated clocks

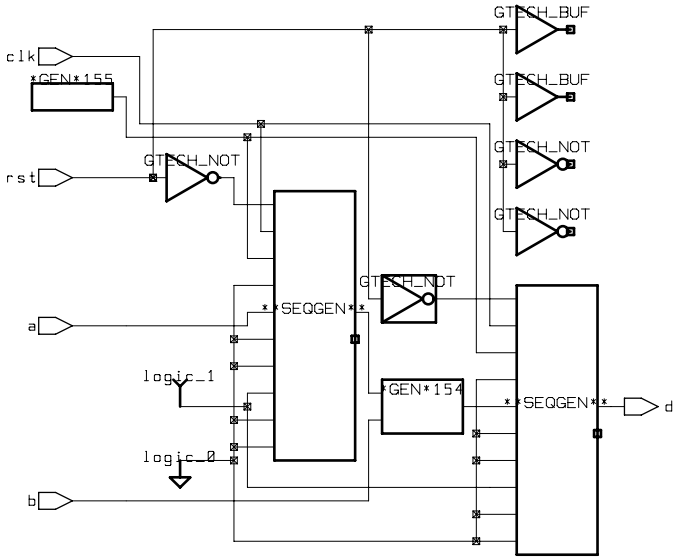
Use `set_clock_skew` to override ideal behavior

Use `set_clock_skew -uncertainty` to specify an upper limit

Timing reports

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FF2 is
port (a,b,clk, rst: in std_logic;
d:out std_logic);
end FF2;

architecture two of FF2 is
 signal f: std_logic;
begin

 process (clk,rst)
 begin
  if rst = '0' then f <= '0';
  elsif clk'event and clk ='1'
  then f <= a;
  end if;
 end process;

 process (clk,rst)
 begin
  if rst = '0' then d <= '0';
  elsif clk'event and clk ='1'
  then d <= f and b;
  end if;
 end process;
end two;
```

Design after read



VHDL after READ

```
entity FF2 is  port( a, b, clk, rst : in std_logic;  d : out std_logic); end FF2;

architecture SYN_two of FF2 is
  component GTECH_NOT  port( A : in std_logic;  Z : out std_logic); end component;
  component GTECH_BUF port( A : in std_logic;  Z : out std_logic); end component;
  component SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT
   generic ( ac_as_q, ac_as_qn, sc_ss_q : integer );
    port( clear, preset, enable, data_in, synch_clear, synch_preset,  synch_toggle, synch_enable,
          next_state, clocked_on : in std_logic; Q, QN : buffer std_logic);
  end component;
  signal a_port, Logic0, f, d56, clk_port, Logic1, d_port, LogicX, n67, n68,
    n70, n71, n72, n73, n74, n75 : std_logic;
begin
  a_port <= a;  clk_port <= clk;   d <= d_port;
  U1 : GTECH_NOT port map( A => rst, Z => n70);
  d56 <= (f and b);
  Logic0 <= '0';
  U11 : GTECH_NOT port map( A => rst, Z => n71);
  U2 : GTECH_BUF port map( A => rst, Z => n72);
  U12 : GTECH_BUF port map( A => rst, Z => n73);
  Logic1 <= '1';
  U25 : GTECH_NOT port map( A => rst, Z => n67);
  U26 : GTECH_NOT port map( A => rst, Z => n68);
  f_reg : SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT
    generic map ( ac_as_q => 5, ac_as_qn => 5, sc_ss_q => 5 )
    port map ( clear => n68, preset => Logic0, enable => Logic0, data_in => LogicX,
```

```
      synch_clear => Logic0, synch_preset => Logic0, synch_toggle => Logic0, synch_enable =>
      Logic1, next_state =>a_port, clocked_on => clk_port, Q => f, QN => n74);
  d_reg : SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT
    generic map ( ac_as_q => 5, ac_as_qn => 5, sc_ss_q => 5 )
    port map ( clear => n67, preset => Logic0, enable => Logic0, data_in => LogicX, synch_clear
      => Logic0, synch_preset => Logic0, synch_toggle => Logic0, synch_enable => Logic1,
       next_state => d56, clocked_on => clk_port, Q => d_port, QN => n75);
  LogicX <= '0';
end SYN_two;

entity SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT is
  generic ( ac_as_q, ac_as_qn, sc_ss_q : integer );
  port( clear, preset, enable, data_in, synch_clear, synch_preset, synch_toggle,
      synch_enable, next_state, clocked_on : in std_logic; Q, QN : buffer std_logic);
end SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT;
```

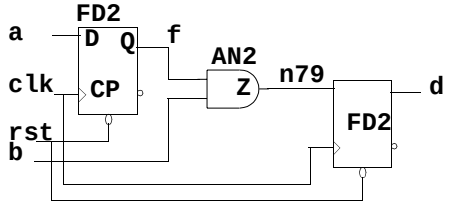

Basic Sequential Element

```
architecture RTL of SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT is
begin
  process ( preset, clear, enable, data_in, clocked_on )
  begin
    -- Check the value of inputs (asynchronous first)
    if ( ( ( preset /= '1' ) and ( preset /= '0' ) ) or ( ( clear /= '1' )  and ( clear /= '0' ) )  ) then Q <= 'X'; QN <= 'X';
    elsif ( clear = '1' and preset = '1' ) then
      case ac_as_q is when 2 => Q <= '1'; when 1 => Q <= '0'; when others => Q <= 'X'; end case;
      case ac_as_qn is when 2 => QN <= '1'; when 1 => QN <= '0'; when others => QN <= 'X'; end case;
    elsif ( clear = '1' ) then Q <= '0'; QN <= '1';
    elsif ( preset = '1' ) then Q <= '1'; QN <= '0';
    elsif ( ( enable /= '1' ) and ( enable /= '0' ) ) then Q <= 'X'; QN <= 'X';
    elsif ( enable = '1' ) then Q <= data_in;  QN <= not( data_in );
    elsif ( ( clocked_on /= '1' ) and ( clocked_on /= '0' ) ) then Q <= 'X'; QN <= 'X';
    elsif ( clocked_on'event and clocked_on = '1' ) then
      if ( ( ( synch_preset /= '1' ) and ( synch_preset /= '0' ) ) or ( ( synch_clear /= '1' ) and ( synch_clear /= '0' ) )  ) then
        Q <= 'X';  QN <= 'X';
      elsif ( synch_clear = '1' and synch_preset = '1' ) then
        case sc_ss_q is when 2 => Q <= '1';  QN <= '0'; when 1 =>  Q <= '0'; QN <= '1'; when others => Q <= 'X'; QN <= 'X';
        end case;
      elsif ( synch_clear = '1' ) then  Q <= '0'; QN <= '1';
      elsif ( synch_preset = '1' ) then  Q <= '1'; QN <= '0';
      elsif ( ( ( synch_toggle /= '1' ) and ( synch_toggle /= '0' ) ) or ( (synch_enable /= '1' ) and ( synch_enable /= '0' ) )  ) then
        Q <= 'X'; QN <= 'X';
      elsif ( synch_enable = '1' and synch_toggle = '1' ) then Q <= 'X'; QN <= 'X';
      elsif ( synch_toggle = '1' ) then  Q <= QN; QN <= Q;
```

```
      elsif ( synch_enable = '1' ) then  Q <= next_state; QN <= not( next_state );
      end if;
    end if;
  end process;
end RTL;
```

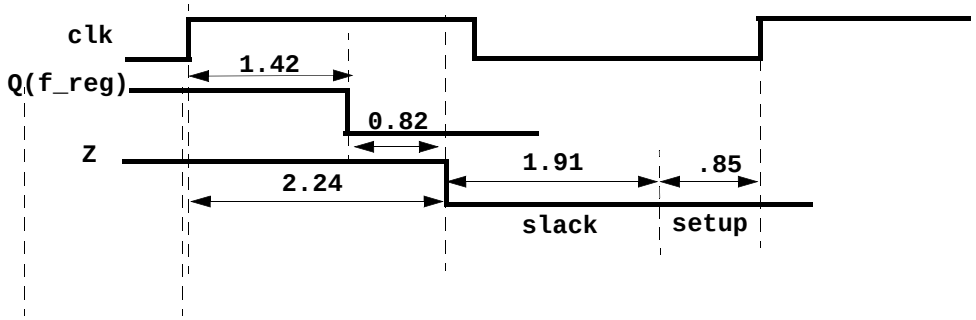

After compilation to lsi_10k;

```
entity FF2 is port( a, b, clk, rst : in std_logic;  d : out std_logic);
end FF2;

architecture SYN_two of FF2 is
 component AN2 port( A, B: in std_logic;  Z: out std_logic); end component;
 component FD2 port( D, CP, CD : in std_logic;  Q, QN : out std_logic);
 end component;
 signal f, n79, n80, n81 : std_logic;
begin
   U28 : AN2 port map( A => f, B => b, Z => n79);
   f_reg : FD2 port map( D => a, CP => clk, CD => rst, Q => f, QN => n80);
   d_reg : FD2 port map( D => n79, CP => clk, CD => rst, Q => d, QN => n81);
end SYN_two;
```



Reports

```
read -f vhdl test.vhd
link_library=target_library=lsi_10k.db
create_clock clk -period 5
compile -exact_map
report_timing -max_paths 5
```



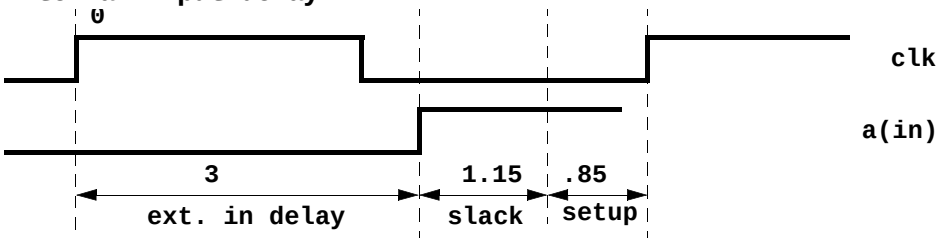

```
Startpoint: f_reg (rising edge-triggered flip-flop clocked by clk)
Endpoint: d_reg (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
Point                                   Incr       Path
-----------------------------------------------------------
clock clk (rise edge)                   0.00       0.00
clock network delay (ideal)             0.00       0.00
f_reg/CP (FD2)                          0.00       0.00 r
f_reg/Q (FD2)                           1.42       1.42 f
U28/Z (AN2)                             0.82       2.24 f
d_reg/D (FD2)                           0.00       2.24 f
data arrival time                                  2.24

clock clk (rise edge)                   5.00       5.00
clock network delay (ideal)             0.00       5.00
d_reg/CP (FD2)                          0.00       5.00 r
library setup time                     -0.85       4.15
data required time                                 4.15
-----------------------------------------------------------
data required time                                 4.15
data arrival time                                 -2.24
-----------------------------------------------------------
slack (MET)                                        1.91
```

```
> set_input_delay  3 -clock clk a
  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clk (rise edge)                    0.00       0.00
  clock network delay (ideal)              0.00       0.00
  input external delay                     3.00       3.00 r
  a (in)                                   0.00       3.00 r
  f_reg/D (FD2)                            0.00       3.00 r
  data arrival time                                   3.00

  clock clk (rise edge)                    5.00       5.00
  clock network delay (ideal)              0.00       5.00
  f_reg/CP (FD2)                           0.00       5.00 r
  library setup time                      -0.85       4.15
  data required time                                  4.15
  -----------------------------------------------------------
  data required time                                  4.15
  data arrival time                                  -3.00
  -----------------------------------------------------------
  slack (MET)                                         1.15
```

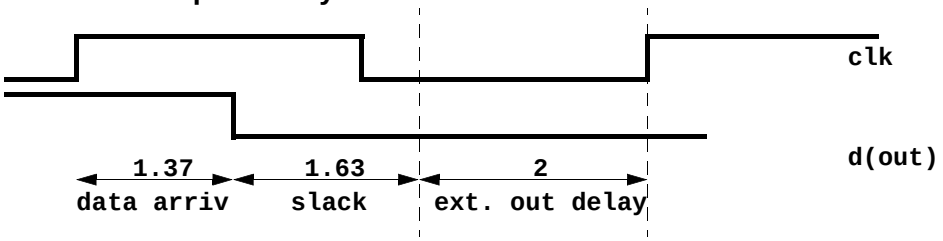


```
> set_output_delay 2 -clock clk  d
  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clk (rise edge)                    0.00       0.00
  clock network delay (ideal)              0.00       0.00
  d_reg/CP (FD2)                           0.00       0.00 r
  d_reg/Q (FD2)                            1.37       1.37 f
  d (out)                                  0.00       1.37 f
  data arrival time                                   1.37

  clock clk (rise edge)                    5.00       5.00
  clock network delay (ideal)              0.00       5.00
  output external delay                   -2.00       3.00
  data required time                                  3.00
  -----------------------------------------------------------
  data required time                                  3.00
  data arrival time                                  -1.37
  -----------------------------------------------------------
  slack (MET)                                         1.63
```

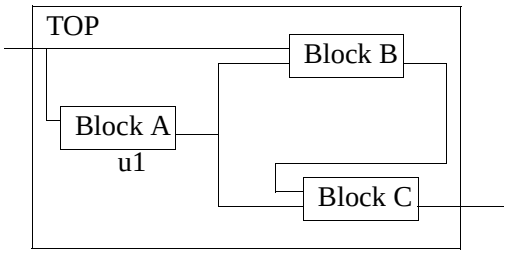
External input delay



External output delay



Set_dont_touch



Useful in hierarchical designs

Assigned to a design or library cell

Allows keeping a subdesign unchanged during re-optimization

Applied to an instance u1

```
current design = TOP
set_dont_touch u1
or
set_dont_touch find(cell,u1)
```

Applied to a design

```
current_design=BlockA
set_dont_touch find(design, BlockA)
```

Applied to a design $\Rightarrow$ all instance

Removing

```
remove_attribute find(design,A) dont_touch
remove_attribute find(cell,C) dont_touch
```

Flattening

Put combin. logic as $\Sigma\Pi$

Achievable for less than 20 inputs

May be expensive

$$\begin{bmatrix} Y1 = (a+b) \\ X1 = Y1C \end{bmatrix} \Rightarrow X1 = ac + bc$$

To specify

```
set_flatten true
set_structuring -timing true
```

To verify options

```
report_compile_options
```


Structuring

Improves area and gate count

Timing driven (by default) or boolean structuring

Boolean struct. $\Rightarrow$ 2X to 4X compilation time

$$\begin{bmatrix} X1 = (ab + ad) \\ X2 = (bc + cd) \end{bmatrix} \Rightarrow \begin{bmatrix} Y1 = (b+d) \\ X1 = aY1 \\ X2 = cY1 \end{bmatrix}$$

Grouping and using

Dealing with hierarchy

`ungroup group` ≡ remove create levels of hierarchy

`ungroup -flatten -all` ≡ recursive ungroup except dont_touch cells

`replace_synthetic -ungroup` ≡ ungroup synthetic designs

```
group {u1, u2} -design_name B1 -cell_name C_N
```

During technology mapping

```
set_dont_use lsi_10k/FD2S
set_prefer lsi_10k/FD2
```

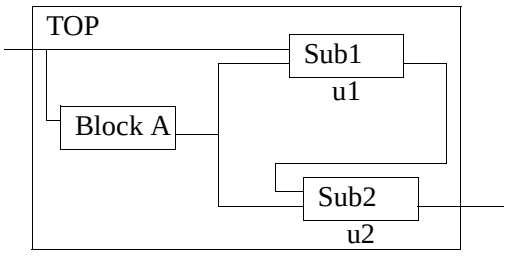

Characterization

Used in hierarchical designs

Constraints on sub-designs depend on environment

`characterize` capture surrounding constraints

```
read -f db TOP.db
characterize u1
current_design = sub1
write script > sub1.scr
compile
current_design TOP
characterize u2
current_design = sub2
write script > sub2.scr
compile
```



Guidelines

Specify accurate timing

Accurate point to point delays for asynch paths

Create_clock, group_path for synch. paths

Register output

Simplifies time budgeting

-ive, +ive FF in ≠ hierarch. modules

simpler debugging and timing analysis

simplifies test insertion

Guidelines (cont'd)

Group FSMs, optimize separately

Size: 250-5000

Middle-of-the road strategy

Balance Hierach. vs. large flat design

Critical path "should not" traverse hierarch. boundaries

Consider alternatives : instantiate logic vs. infer through DesignWare

Put in same level of hierarchy

driving and driven of large fanouts

Sharable resources: e.g. adders

Guidelines (cont'd)

Compile time too long ?

- High map effort
- Design too large
- Declared false paths traversing hierarchies
- Glue logic at top level
- Inappropriate flattening
  - Adders, muxes, XORs
  - Over 20 inputs
- Boolean optimization ON.
- Not enough memory

Perform preliminary synthesis + Place& Route

- Consider re-writing VHDL if necessary

X. Finite State Machines

Extracting FSMs

```
package states is
type state is (s0, s1, s2, s3);
end states;

use work.states.all;
entity ET is
port (x, clk: in bit;
      z: out bit);
end ET;




architecture One of ET is
signal st: state;
begin
process
begin
wait until clk'event and clk ='1';
if x='0' then z <= '0';
else
 case st is
   when s0 => st <= s1; z <='0';
   when s1 => st <= s2; z <='0';
   when s2 => st <= s3; z <='0';
   when s3 => st <= s0; z <='1';
   when others => st <= s0;
 end case;
end if;
end process;
end One; -- registered outputs
```

> read -format vhdl fsm1.vhd

Inferred memory devices in process

Register Name	Type	Width	Bus	AR	AS	SR	SS	ST
st_reg	Flip-flop	2	Y	N	N	N	N	N
z_reg	Flip-flop	1	-	N	N	N	N	N

> compile -map_effort low

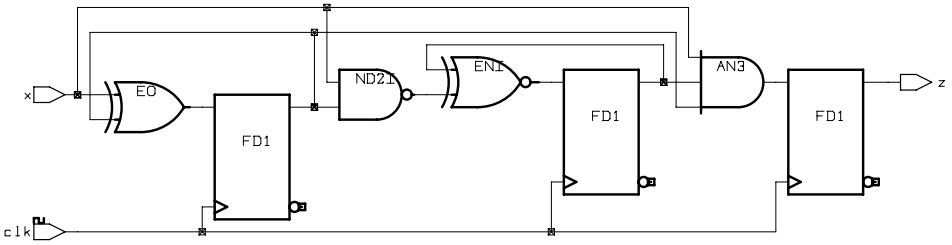
TRIALS	AREA	DELTA DELAY	OPTIMIZATION COST	DESIGN RULE COST
10				
10				

Optimization complete

> report_fsm

The design is not currently represented as a state machine

Schematic



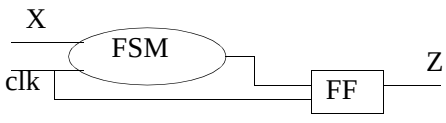
> set_fsm_state_vector { “st_reg[0]” “st_reg[1]” }

> set_fsm_encoding { “s0=2#00” “s1=2#10” “s2=2#01” “s3=2#11”}

> group -fsm -design_name eg1_fsm

> current_design =eg1_fsm

> extract



> report_fsm

```
Clock             : clk         Sense: rising_edge
Asynchronous Reset: Unspecified
Encoding Bit Length: 2
Encoding style     : Unspecified
State Vector: { st_reg[0] st_reg[1] }
```

> write -format st -o state_mach.st

```
...                                 1 s2 s3 ~
1 s0 s1 ~                           0 s2 s2 ~
0 s0 s0 ~                           - s2 ~  0
- s0 ~  0                           1 s2 s3 ~
1 s0 s1 ~                           0 s2 s2 ~
0 s0 s0 ~                           1 s3 s0 1
1 s1 s2 ~                           0 s3 s3 ~
0 s1 s1 ~                           1 s3 s0 ~
- s1 ~  0                           0 s3 s3 0
1 s1 s2 ~
0 s1 s1 ~
```


Coding FSMs in VHDL

```
package states is
type state is (s0, s1, s2, s3);
end states;

use work.states.all;
entity ET is
port (x, clk: in bit;
      z: out bit);
end ET;
architecture One of ET is
  signal st: state;
  attribute state_vector: string;
  attribute state_vector of One:
   architecture is "st";
begin
process
begin
wait until clk'event and clk ='1';
if x='0' then z <= '0';
else
 case st is
   when s0 => st <= s1; z <='0';
   when s1 => st <= s2; z <='0';
   when s2 => st <= s3; z <='0';
   when s3 => st <= s0; z <='1';
   when others => st <= s0;
 end case;
end if;
end process;
end One; -- registered outputs
```

> read -format vhdl fsm2.vhd
same as previous example

> report_fsm
Recognizes the FSM

```
Clock            : Unspecified
Asynchronous Reset: Unspecified
Encoding Bit Length: 2
Encoding style     : Unspecified
State Vector: { st_reg[1] st_reg[0] }
State Encodings and Order:
S0         : 00
S1         : 01
S2         : 10
S3         : 11
```

> compile

> group -fsm -design_name fsm_1hot ***similar to previously***

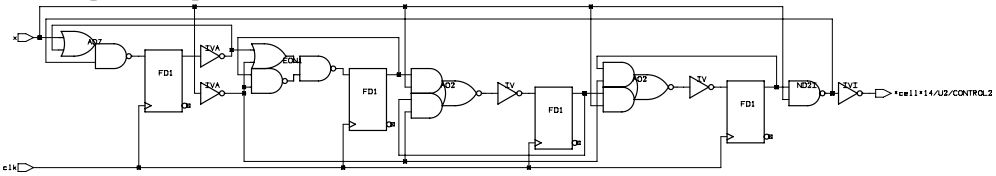
> extract

> set_fsm_encoding_style one_hot

> set_fsm_encoding { “S0=2#1000” “S1=2#0100” “S2=2#0010” “S3=2#0001” }

> set_fsm_minimize true

> compile -map_effort low



> ungroup -all