VLSI 1

ETHZ HS19 227-0116-00 January 2020 Noah Hütter

Github: noah95/formulasheets

I. Introduction

I.1 Economic Aspects

There are 4 business models:

- IDM: Integrated device manufacturer Chip design and fab (Intel)
- Fab-lite Fabless chip vendor (Samsung, Infineon)
- Chip, Fab + Device (Samsung)
- Fab only (TSMC)

Fab-less
- Qualcomm

POVs

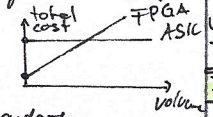
#1 records: Tech. technology > 10^9 transistor count

#2 marketing: General purpose vs. application specific

#3 manufacturing: Semi custom: Given gate array, add custom metal = custom circuit

FPL: Field programmable logic (FPGA) LUT/FF that are connected esp. switchboxes

Full-custom and standard part



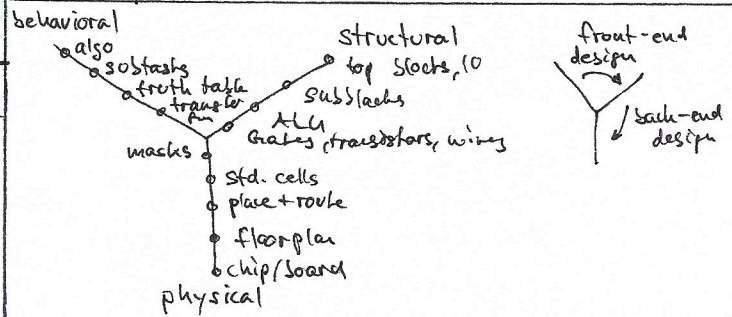
#5 business models: 4 from # plus IP-vendors and system house: integrates HW, SW into product

#4 Design engineer

Standard cell:	FF / LUT
Mega cell:	ADC / DAC
Macro cell:	RAM / ROM

II. Designflow & FPGA

Y-Chart



Major stages

1. System-level design

Alternatives, ~~Fct~~ Functionality, interfaces

→ sys.-level model

2. Algorithm design

Allocation, scheduling, binding

→ Bit-true SW model

3. Architecture design

→ Floorplan, RTL code

4. Physical design → masks, verification

FPGA

LUT perform logic operations

FF register element stores result from LUT

Interconnect/Wires Connect elements to one another, both logic and clock

IO physically available ports get signals in and out of FPGA

} main comp.

special BRAM for memory, DSP, PLL

memory
- Block RAM dedicated memory ressource
- LUT as RAMs interconnected
- FFs in CLBs used as RAM

flow of tool

1. Elaboration Generates netlist of generic logic and arithmetic elements from RTL description
2. Synthesis Maps netlist components to elements available on FPGA and directly connect. Results in netlist and timing/resource estimates
3. Implementation Synthesized netlist mapped onto target device: Place & route
4. Gen. Bitstream Generate sequence of bits to program device

III. HDL Design & SystemVerilog

About

- Case sensitive
- Logic values 0, 1, X, Z
- Compact
- Weak type checking
- for structural, behavioral, timing and testbenches (all)

Hierarchy

module name of block, parameters and signal names. Practice: One file per module

```
module top #(
    parameter int Width = 16
)(
  input  logic             clk_i,
  output logic [Width-1:0] res_o
);
```

Body Describes functions, wires and instances

```
logic [Width-1:0] test;
ffs #(.Width(16)) i_reg (.clk_i(clk_i), .out_o(test));
assign foo = a | b;
```

Wires

0,1 driven value 'X' drive-conflict, don't care
'Z' High impedance

Main type: `logic` for I/O, signals, constants

If 'Z' required, use `wire` or `reg` to R/W to

Arrays:

packed: `logic [7:0] vec` ordered MSB to LSB

unpacked: `int foo [0:15]` can be ordered both ways
`int foo [8]` = `int foo [0:7]`

address array: `first = vec[0]` `range = vec[2:0]`

concat: `y = {a[2], a[1], a[0], a[0]}` `y = {4{a[0]}}`

Expressing numbers N'Bxx `4'b00_11`

N: number of bits
B: base binary, hex, decimal, octal
xx: number '_' are ignored → readability

more types `shortint` 2state 16bit `int` 2state 32bit
`longint` 2state 64bit `byte` 2state 8bit/ASCII
`integer` 4state 32bit `time` 4state 64bit unsigned
`int unsigned` `int signed`

subtypes `typedef logic [7:0] u8_t; u8_t a;`

Naming

_i in _o out _t typs _r asynch. rst
_c clock _n active low rst_ni convination

logic

~ not & and | or ^ xor ↙ ternary operator

multiplexer: `assign res = sel ? a : b;`
`assign res = (sel == 2'b11) ? d3 : (sel == 2'b00) ? ...`

logical: ! not && and || or == equal != not equal

Sequential

Old Verilog style

is only executed if signal in sensitivity list changes

```
always @(posedge clk, rst) begin
  // ...
end
```

New SV style

`always_comb` for combinatorial stuff, no sens. list use blocking '=' operator

`always_latch` latches (bad)
`always_ff` most used for seq. } use '<='

Assignments

`=` Blocking, signal assigned before next line is executed
`<=` unblocking, assigned at end of process

Naming

_d next state signals, assign in always_comb
_q current state, assign in always_ff

Enum

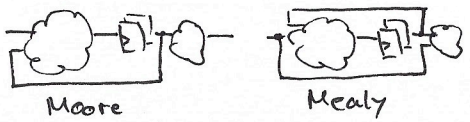
```
typedef enum logic [1:0] {Init, Run, Stop, Wait} state_t;
state_t state_d, state_q;
//...
if (state_q == Run) begin
  state_d = Wait
end else if (-) -
```

case

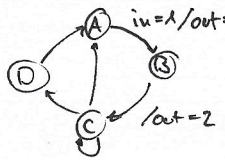
```
case (state_q)
  Run:  state_d = Wait;
  Stop, Wait: begin
    //--
  end
  default: //-- ;
endcase
```

more simple:
`enum logic {a,b,c} state_d;`
if 4 bit state.

FSM



state transition diagram



3 parts:
- next state logic
- state holding element
- output logic

IV. Verification

Functional V. Given set of inputs, what is expected outp.?

Formal V. All requirements described mathematically (formally, then) proven

Directed Testiy Choose small number of input vectors (common, edge cases and random)

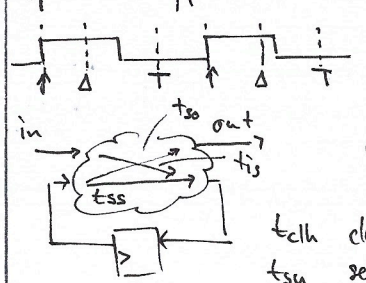
`$random(seed);` returns random int
`$urandom(15,8);` random int within range

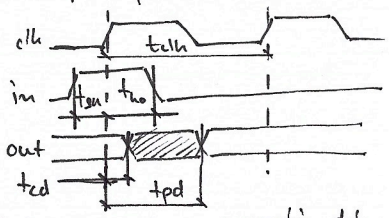
Coverage Checks if all lines of code have been reached, how many times a bit toggled, etc...

Simulation

ATI-Timing specifies clock period, stimuli application time and response acquisition time

↑ order operation Δ apply inp. T test outp
tap stim. appl. time tr resp. acq. time

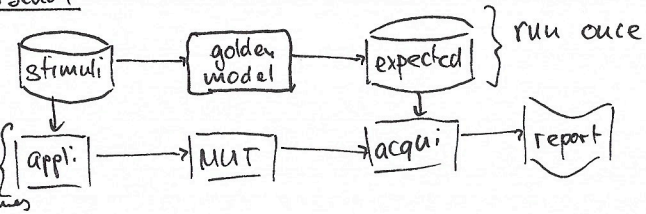




t_{clh} clock period t_{pa} propagation delay
t_{su} setup time t_{ho} hold time
t_{cd} contamination delay, relevant for two of next

$t_{i,max} = t_{clh} - t_{su}$ max input delay
$t_{o,max} = t_{clh} - t_{pd}$ max output delay

Testbench



Formal

property fact about design **assert** directive to prove property

concurrent: `assert property adr > 0;`

assert specify prop. that will hold true

cover check if prop. has occured and count

assume defines constraint for formal verifier

`assert property (@(posedge clk) a |-> b);`

`a |-> b` if a is true b should hold true

`a |=> b` if a is true b should hold true in next cycle

`a |=> ##1 b` b 2 cycles after a

`a ##1 c ##3 d |=> b` if first a, c, , , d then b in next (non consecutive)

`a [*3] ##1 |=>` repeat ×3 with 1 delay

`#[1:4]` delay btw. two values, $ for unbounded

! assertions require high CPU effort !

SystemVerilog

Timing

```
localparam time TC = 100ns;
time sim_time;
```

```
timeunit 1ns; //default tu
timeprecision 0.1ns;
timescale 1ns/100ps; //old
```

`#10;` delay 10 time units

`#(PER/2);` only executed once

`@(rst);`

`@(negedge clk);` wait for condition

```
initial begin
  a = 1'b1;
  #10;
  a = 1'b0;
end
```

`wait (rst==1) $info("reset");` wait to get high

```
always begin
  clk=1; #(PER/2); clk=0; #(PER/2);
end
```

clock gen process

File IO

```
integer fd = $fopen("file","r");
string in;
integer n = $fgets(in, fd);
$fclose(fd);
```

```
logic [3:0] tv [1000:0];
initial begin
  $readmemb("file", tv);
end
```

file:
```
0000
0110
1000
```

```
while (!$feof(fd)) begin
  @(posedge clk)
  #STIM_APPL_TIME;
  $fsanf(fd, "%h\n", stim);
end
```

```
typedef struct packed {
  logic [2:0] a;
  logic       b;
  logic [7:0] c;
} data_t;
data_t stim;
$fsanf(fd, "%5\n", stim);
```

010 1 1111 0000
a b c

Generate

```
generate
  for (genvar ii=0; ii < num; ii++) begin : gen_bus
    my_bus #(.index(ii)) i_my_bus (.foo(foo), .bar(bar[ii]));
  end
endgenerate
```

V. Algorithm to Architecture

Solution Space

Antipodes

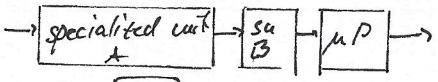
	Algo	Arch	Exec	Datapath	Strengths
General purpose	any	instruction set	fetch-load-execute	ALU	flexible, low cost
Special purpose	fix	dedicated	dataflow	dedicated	max perf., low power

VLSI suitability

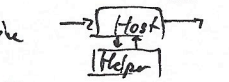
1. Loose coupling btw. major processing tasks
2. Simple control flow
3. Regular data flow
4. Reusable storage
5. Finite precision
6. Non-recursive LTI
7. No transcendental (trigo, root, log, ...)
8. Extensive operations unavailable in instruction sets
9. No wide div/mult
10. Throughput rather than latency

Other platforms

1. Dedicated satellites

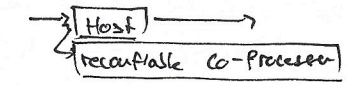


2. Host with helper cycle



3. ASIP: Application specific instruction set processor

4. RC: reconfigurable computing

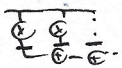
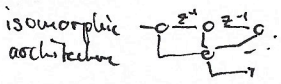


5. DSPP: Domain-specific programmable platform
= combination of all the above. Eg. Zynq

Dedicated Arch.

DDG: Data dependency graph ○ memory-less operation

→ transport —|→ latch —n→ n latches

Equ: $y = \sum \ldots$ DDG:  isomorphic architecture 

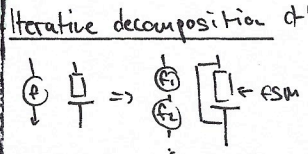
Figures of Merit

Γ	cycles per data item	t_{lp}	longest path delay
T	time per data item	Θ	throughput
L	latency	A	circuit size (GE: technology indep.)
AT	area-time prod.	E	energy per data item
ET	energy-time prod.		

$T = \Gamma \cdot T_{cp}$ $\quad \Theta = \frac{1}{T} = \frac{f_{cp}}{\Gamma}$

Equivalence Transforms

Iterative decomposition d times

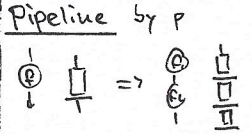


$\frac{A_f}{d} + A_c + A_c \le A \le A_f + A_r + A_c$

$\Gamma = d \quad t_{lp} = \frac{t_f}{d} + t_r \quad L = d$

\+ good if repetitive computation
\- bad if disparative function

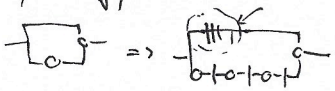
Pipeline by p



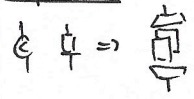
$A = A_f + p \cdot A_r$

$\Gamma = 1 \quad t_{lp} \approx \frac{t_f}{p} + t_r \quad L = p$

\+ can clock p-times faster ! - latency incr.
\+ significantly energy efficient - cannot always split exactly in half

⚠ skimming 

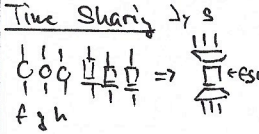
Replication by q



$A = q \cdot (A_f + A_r) + A_{ctl}$

$\Gamma = \frac{1}{q} \quad t_{lp} \approx t_f + t_r \quad L = 1$

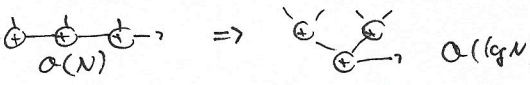
Time Sharing by s



$\max_{f,g,h} A_i + A_r + A_{ctl} \le A \le \sum_{f,g,h} A_i + A_r + A_{ctl}$

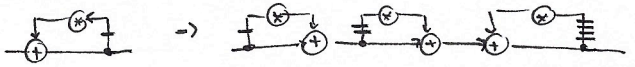
$\Gamma = s \quad t_{lp} \approx \max_{f,g,h} t_{lp,i} + t_r \quad L = s$

\+ good if all streams processed same way
\- bad if subfunctions are disparate

Associativity 

Retiming

Loop Unfolding



Summary

	Iterative Decomp	Pipelin	Replicate	Timeshare	Associativity	Retiming	Loop Unfolding	Better is
A	−/=	=/+	+	−/=	=	=	+	−
Γ	+	=	−	+	=	=	=	−
t_{lp}	−	−	=, max	=	−/+	−	−	−
T	=	−	−	+	−/+	−	−	−
AT	−/=	−/=	=	=/+	−/+	−	+	−
L	+	+	=, max +	+	=	=	+	−
E	−/+	−/+	=	=/+	−/+	=	+	−
ET	+/−	−	−/=	+	−/+	−		−

Temporary Data Storage

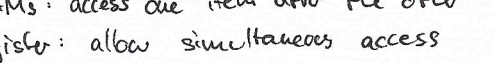
Options: On-Chip (BRAM, LUT-RAM), Off-Chip (SRAM, DRAM)

Data access patterns
- RAMs: access one item after the other
- Register: allow simultaneous access

Off-chip: Extra parts, delays, routing, energy
DRAM: memory refresh cycles

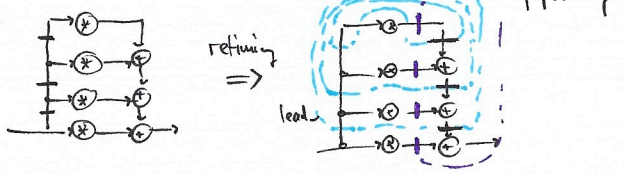
Transform non-recursive computation

Def: result depends on present and past, DDG cycle free

Retiming: 

Rules:
- Neither outputs nor sources of time-varying inputs may be part of a supervertex that is to be retimed
- When a supervertex is assigned a lag (lead) by ℓ comp. cycles, the weights of all its incoming edges are in- (de-) cremented by ℓ
- No negative edge weights
- Any circular path must always include at least one edge of strictly positive weight

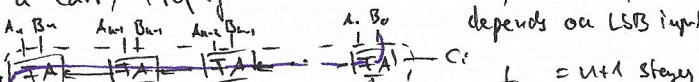
example:



+ t_{lp} greatly reduced at zero HW area cost

Transforming recursive computation

Def: Result depends on earlier outcomes: circles/loops in DDG

$H(z) = \frac{Y(z)}{X(z)} = \frac{N(z)}{1 - a^p z^{-r}}$ $N(z) = \sum a^n z^{-n}$ 

− not suitable for higher order loops

Nonlinear FB-loops are in general not amenable to throughput multiplication by applying unfolding.

VI HLS: High Level Synthesis

Idea: Use C/C++ to design VLSI

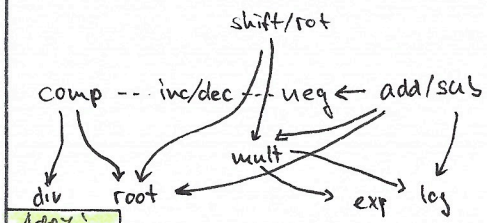
Behavioral Description
- Code needs to base on synthesizable sub-set (static pointers, bounded arrays, no recursive functions, ...)
- Timing directives required

Flow: Basic building blocks: Functional units, storage elements, interconnect, control logic

Steps: 1. Lexical Preprocessing 2. Algo optimization 3. Control data flow graph analysis 4. Resource allocation 5. Library processing 6. Scheduling 7. Register allocation and binding 8. Output processing

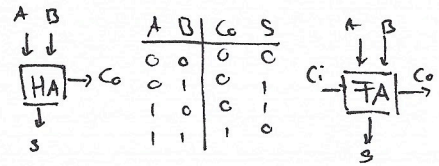
RTL designers for HLS: HLS is not a push-button flow. Need to know HW to be able to program it.

VII Computer Arithmetic Basics



→ all ops require add!

Addition



Half-adder — Full-adder

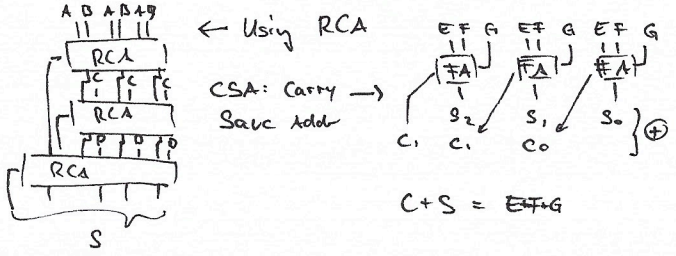
RCA: Ripple Carry Adder

is a Carry Propagate adder (CPA)

- Curse of the carry: MSB output depends on LSB input

$t_{lp} = n+1$ stages

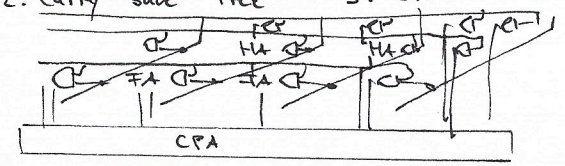
Add multiple numbers



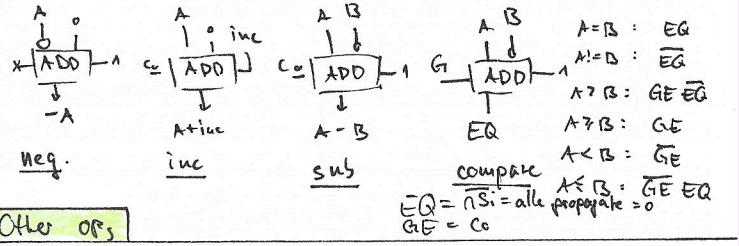
CSA principle: An array of CSA reduce the inputs to two, a final CPA merges the two. Performance mostly dictated by CPA.

Multiplication

1. Partial product generation (simple shift)
2. Carry save tree 3. CPA



OPs based on add



Other OPs

Shift: Logical: shift and insert 0
Arithmetic: shift and if right shift, fill with old MSB

Rotate: rotate circularly

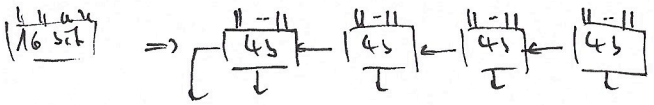
Division: Uncommon, mostly implemented iteratively using mul, comp, sub

Exp/Log/Trig: Use LUT

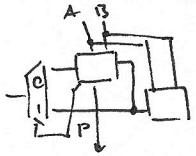
Generate Propagate

Extend FA by two outputs: Generate = A&B and Propagate = A^B G=A&B P=A^B S=P^Ci Co=G|(Ci&P)

Then divide an N-bit adder into multiple n-bit adder

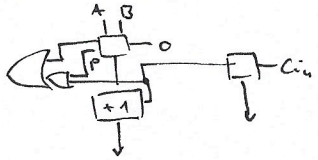


CSKA: Carry Skip



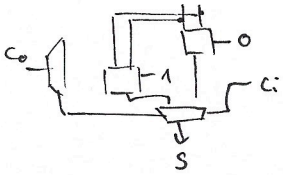
P=0 : carry is generated in this part
P=1 : propagate carry from prev. stage

CIA: Carry Increment



Cin is assumed 0. When carry is known, worst case: output is incremented

CSLA: Carry select



Two parallel adders, one assumes Cin=0, the other Cin=1. Once Cin is valid, the correct calc is selected.

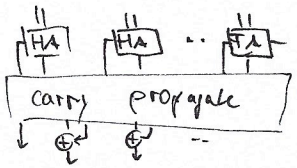
CLA: Carry lookahead

$C_0 = C_i$ $C_1 = C_0 + P_0 C_i$ $C_2 = G_1 + P_1 G_0 + P_1 P_0 C_i$ -

PPA: Parallel Prefix adders

present a systematic approach to designing optimized adders. Stages: pre-processing, carry propagation, post-processing. Only CP is different

CP



$(G_{i:j}^k, P_{i:j}^k)$ are the generate/propagate signals of bits i:j at propagate stage k. Initial is $(G_{i:j}^0, P_{i:j}^0)$. Goal is to know all (GP) for all stages k.

merge two G,P ranges (includes logic)

feedthrough, no logic

Number of • determines circuit area. Number of stages the critical path and max fanout the stage delay.

Examples Sk: Sklansky, BK: Brent-Kung, KS: Kogge-Stone

HC: Huan-Carlsson

	A	T	opt
RCA	7u	2u	aaa
Sk	$\frac{3}{2}u\log n$	$2\log n$	ttt
BK	10u	$4\log n$	att
KS	$3u\log n$	$2\log n$	—
HC			
CIA	10u	$2.8u^{1/2}$	att
CSKA	8u	$4u^{1/2}$	aat
CSLA	14u	$2.8u^{1/2}$	-

RCA: efficient
BK: compromise
Sk: high fanout
KS, HC: fast but high routing

Miscellaneous

Moore FSM

```
// define states
enum logic [1:0] {a,b,c,d} state_d, state_q;


// comb. for next state
always_comb begin
  // important! defaults
  state_d = state_q;
  data_d  = data_q; // other internals
  case(state_q)
    a : begin
      if (req_i) state_d = b;
    end
    b: begin
      data_d = data_i;
      if(!req_i) state_d = a;
    end
  endcase
end

//state regist
always_ff @(posedge clk_i, negedge rst_ni) begin
   if (~rst_ni) begin
      state_q  = a;
   end else begin
      state_q  = state_d;
   end
end

//out assign
assign ack_o  = (state_q ==b)? 1'b1 : 1'b0;
```