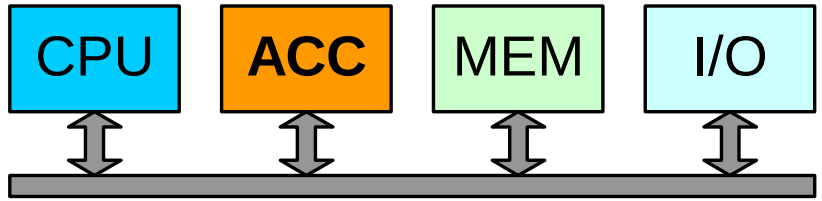
Reconfigurable Instruction Set Extensions using FABulous eFPGAs - when and how

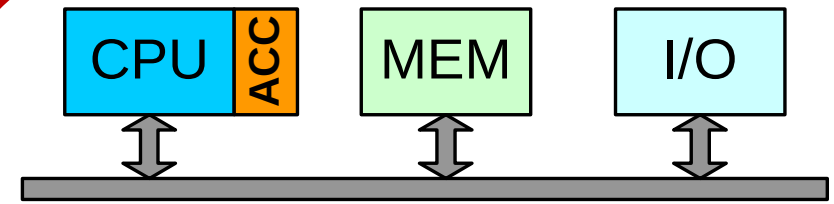
dirk.koch@ziti.uni-Heidelberg.de

Accelerator Coupling



Loosely Coupled

- For „large“ acceleration tasks
  - Compress image / video
  - Encrypt
- Called through drivers
- Hardware-centric (mostly stand alone processing)
- Complex to design



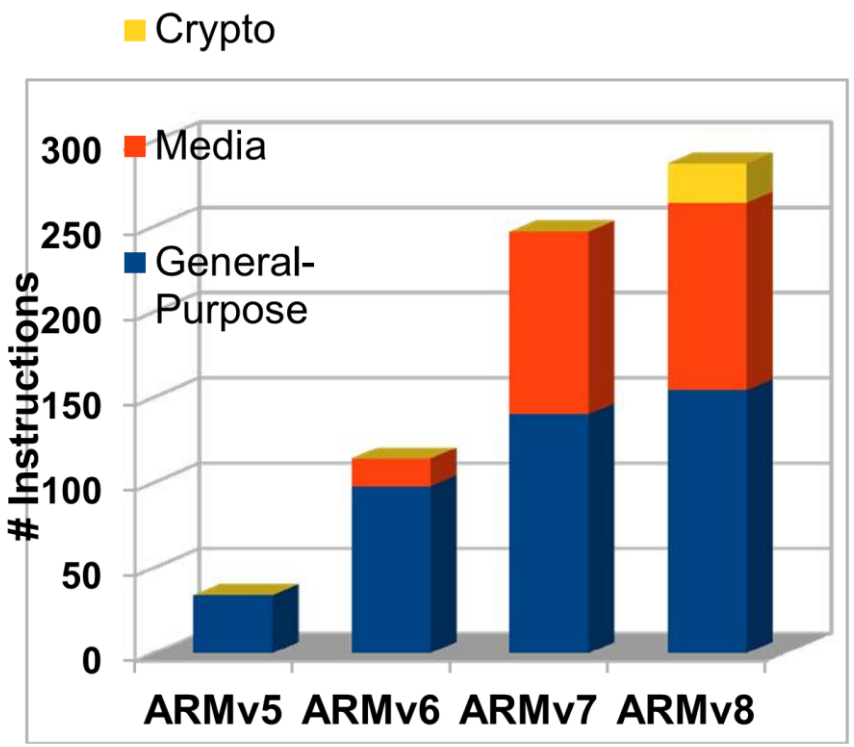
Tightly Coupled

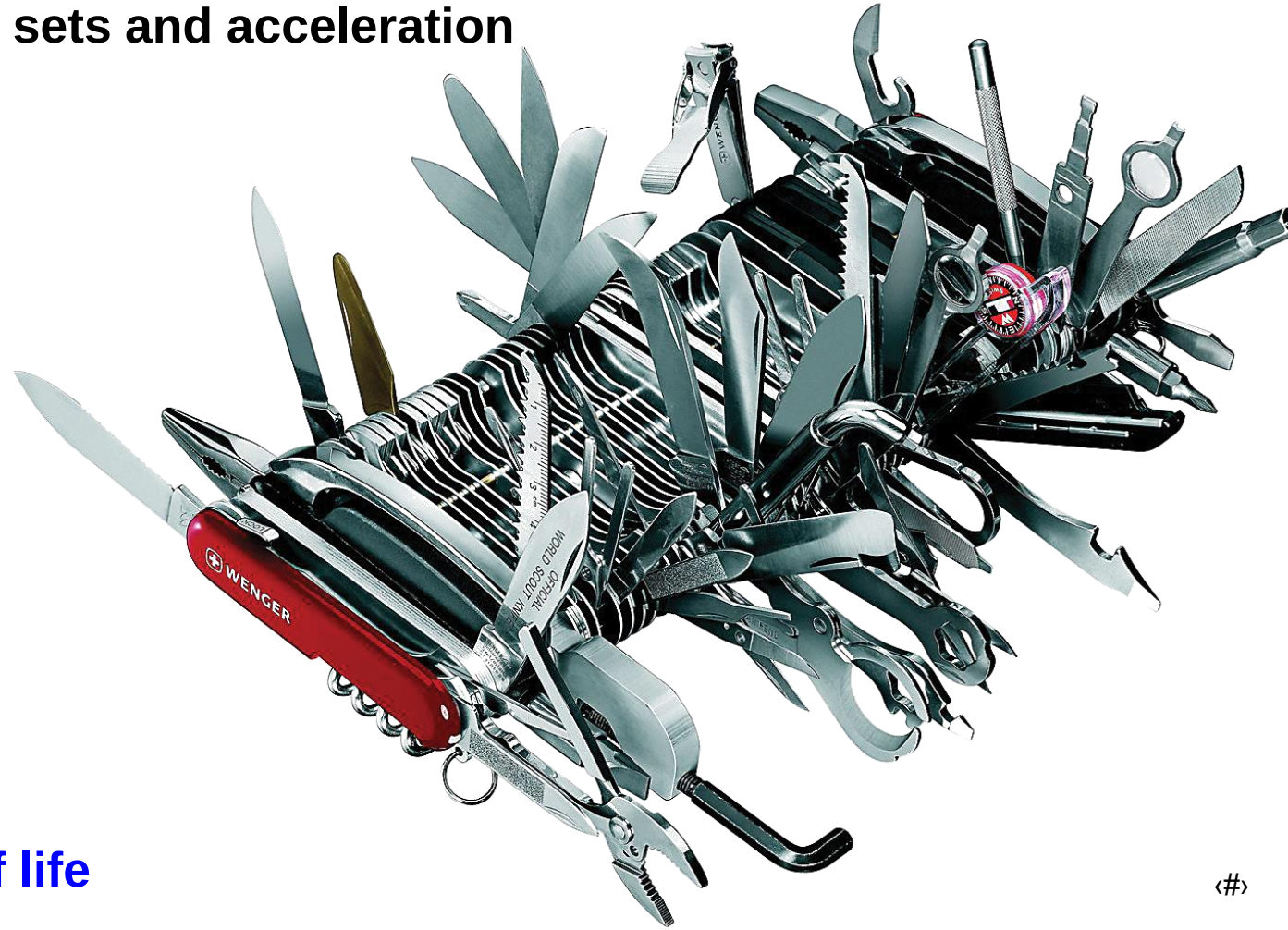
- For „small“ acceleration tasks
  - Parity, count-ones
  - CRC
- Called in user mode
- **<u>Software-centric</u>** (fine-grained function calls)
- Easy to design

Reconfigurable Instruction Set Extensions

- **Present GP CPU micro architectures leave not much headroom for optimization**
- **CPU clock is limited by power**

  → trend to feature-rich instruction sets and acceleration





- **Note that instructions have a shelf life**

Possibilities to add Custom Instructions

- Instructions implemented statically in hardware
- Instructions implemented with reconfigurable hardware (e.g., eFPGA)
- ISA subsetting: kick out unused instructions (usually for FPGAs only)
- Instructions emulated in software (may speed-up your system)

Why/How?

- Adding instructions can make your CPU run slower!

→ removing instructions may make your CPU faster

→ will benefit all instructions!

→ can offset the cost for software emulation (if instr. triggered seldomly)

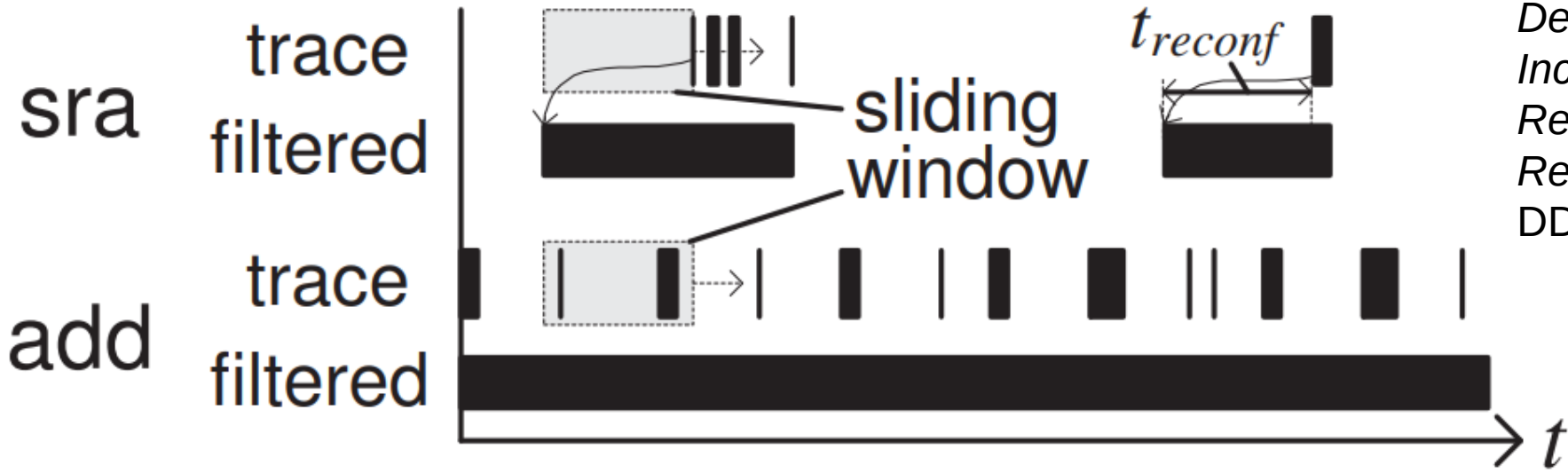
Possibilities to add Custom Instructions

- Static Instructions
- Reconfigurable Instructions
- ISA subsetting
- Software emulation

When what?

Execution pattern	Implementation technique
Frequent	Static hardware
Burst	Reconfigurable hardware
Infrequent	Software emulated
Non-occurring	Removed

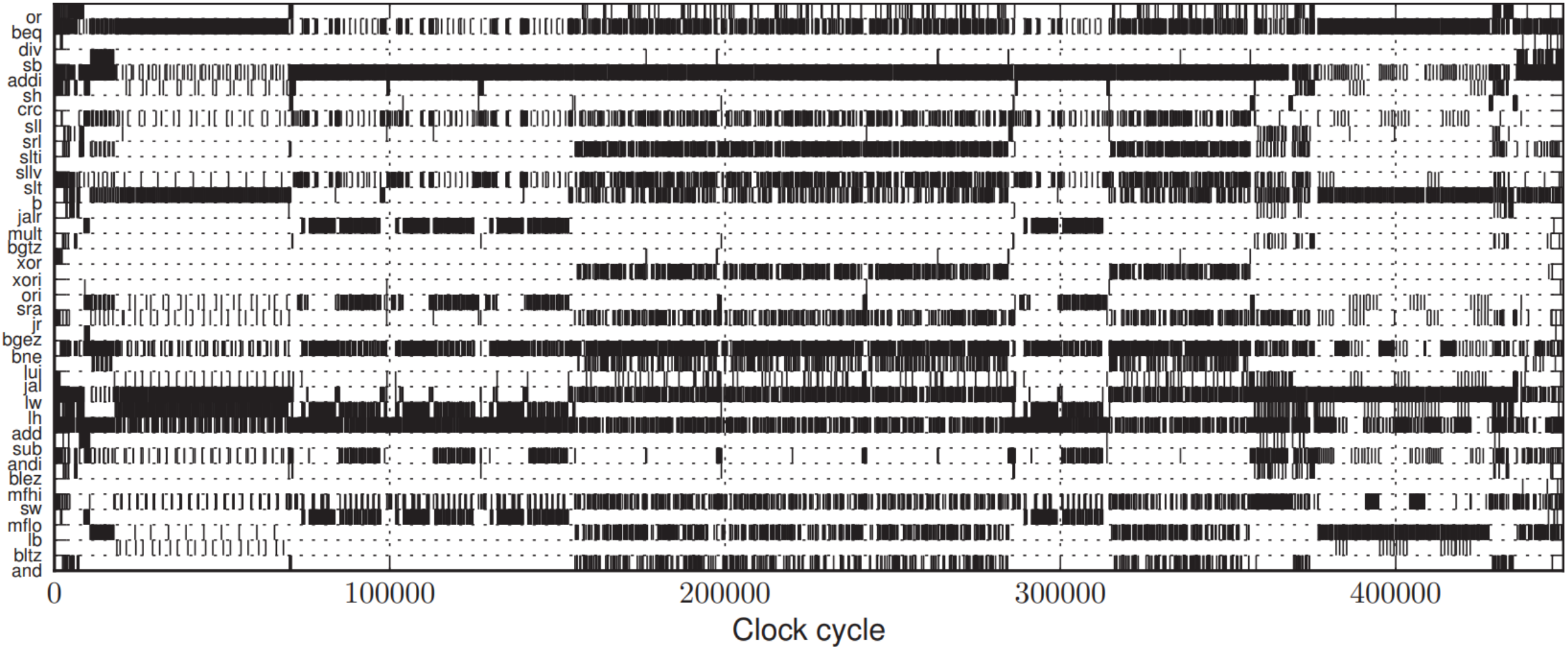
Possibilities to add Custom Instructions



Design Techniques for Increasing Performance and Resource Utilization of Reconfigurable Soft CPUs
DDECS 2018

- Sliding window approach
  - Slide a window of size t_reconfiguration over the instruction stream (for each instruction separately)
  - Empty spots in the filtered trace mean reconfiguration is feasible

Possibilities to add Custom Instructions

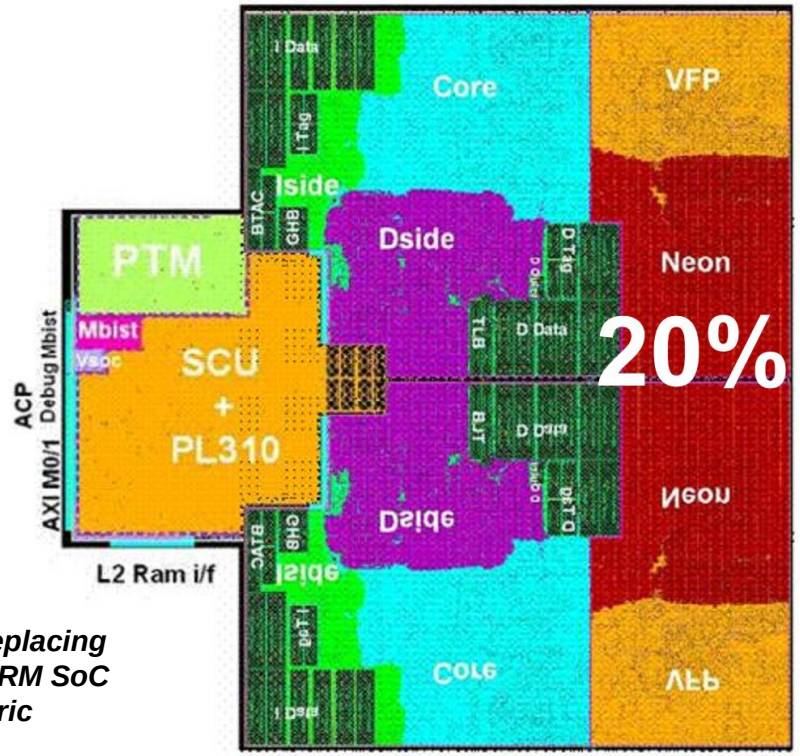


- Configuration is relatively slow → we need course scheduling granularity
- Perhaps at program level (custom HW Kernels)

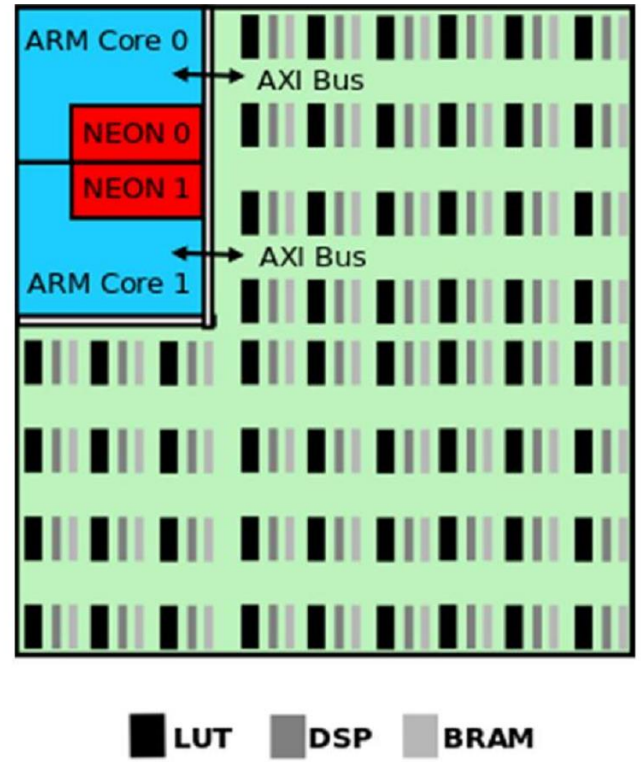
Reconfigurable Instruction Set Extensions

- **Let's replace the NEON vector unit with an FPGA fabric of ~identical size (i.e. 2080 LUTs, 16 DSPs, 8 BRAMs)**
- **Interesting for low precision SIMD arithmetic (128 bits allow 42 3-bit multiplications costing 1764 LUTs)**

Dual ARM A9 SoC Floorplan



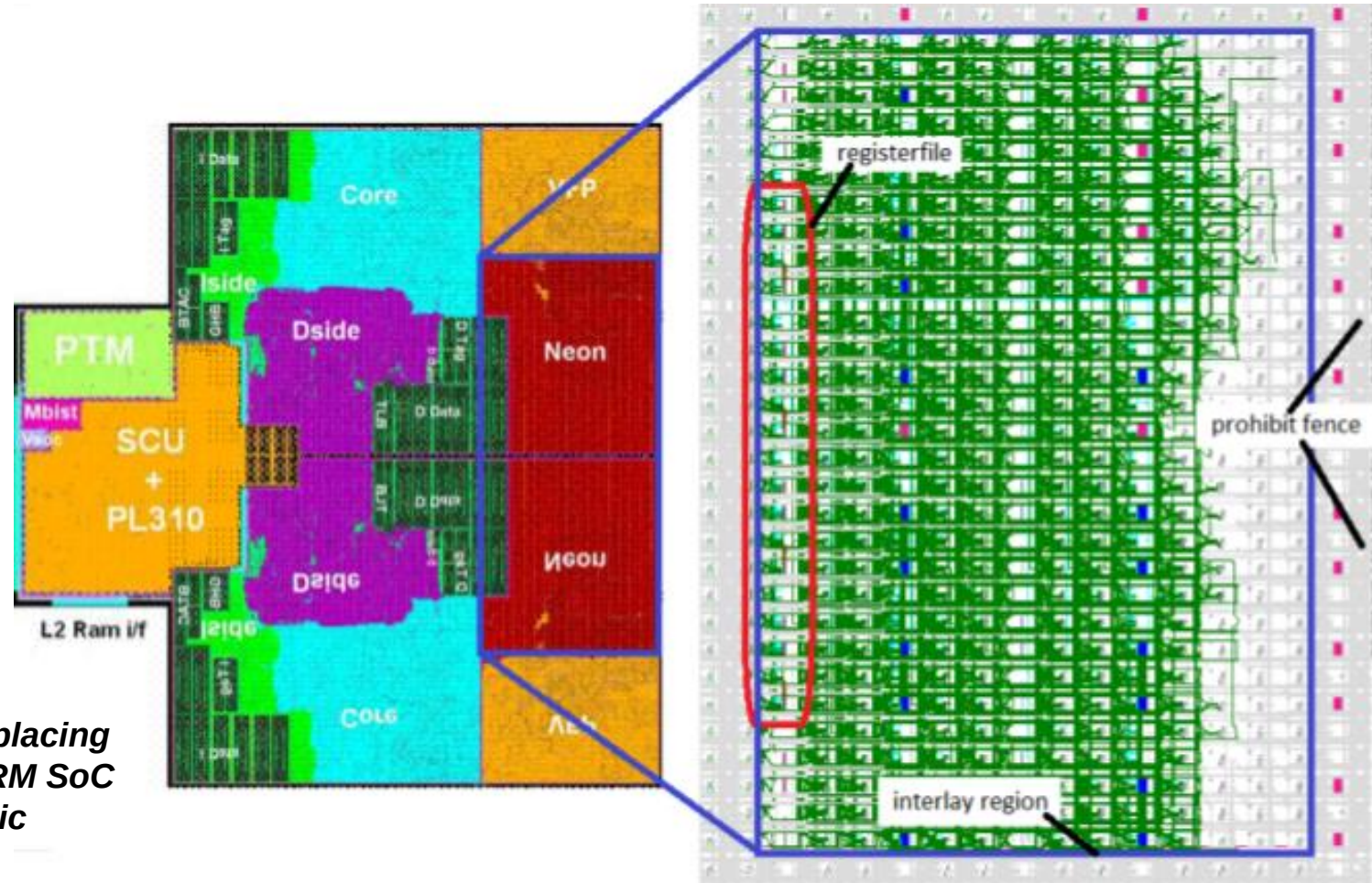
Zynq chip with ARM SoC



soft-NEON: A study on replacing the NEON engine of an ARM SoC with a reconfigurable fabric
ASAP 2016

Reconfigurable Instruction Set Extensions

- **The logic resources are about one 32-bit softcore CPU**
- **Vector interface allows more operands and results and…**
- **…allows catching up with the faster hardened part**



soft-NEON: A study on replacing the NEON engine of an ARM SoC with a reconfigurable fabric
ASAP 2016

Tightly Coupled Reconfigurable Instructions

Candidates:

- int foo (int OP_A, OP_B);
- DES, AES, SHA1-3, MD5 Montgomery, CRC, …
- Hash functions
- (De)Compression (Huffman, bit-level)
- Consider internal registers / register files
- Not bound to 2 x input, 1 x output
  → int foo1 (int OP_A, OP_B); int foo2 (int OP_C, OP_D);
  → push / pop
- Breakpoints, watchpoints (complex triggering), event counters
- Replacing defect operations???

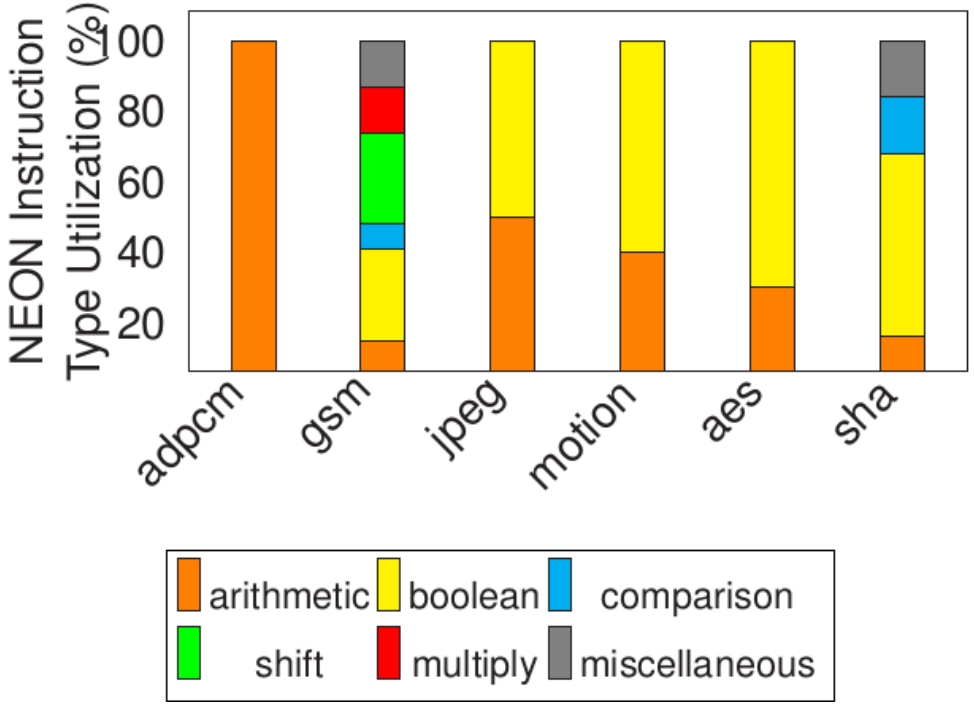
Peer et al. "*Human Skin Colour Clustering for Face Detection*"

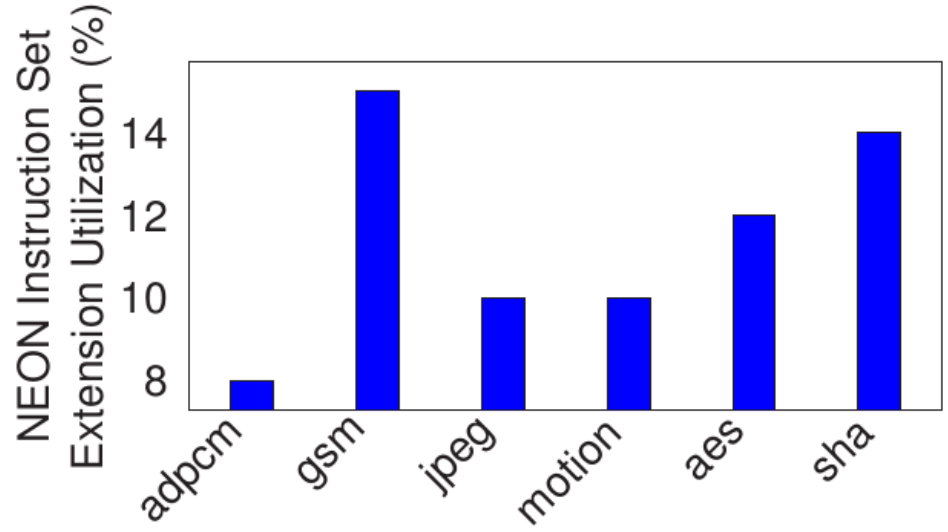
(R, G, B) is classified as skin if:

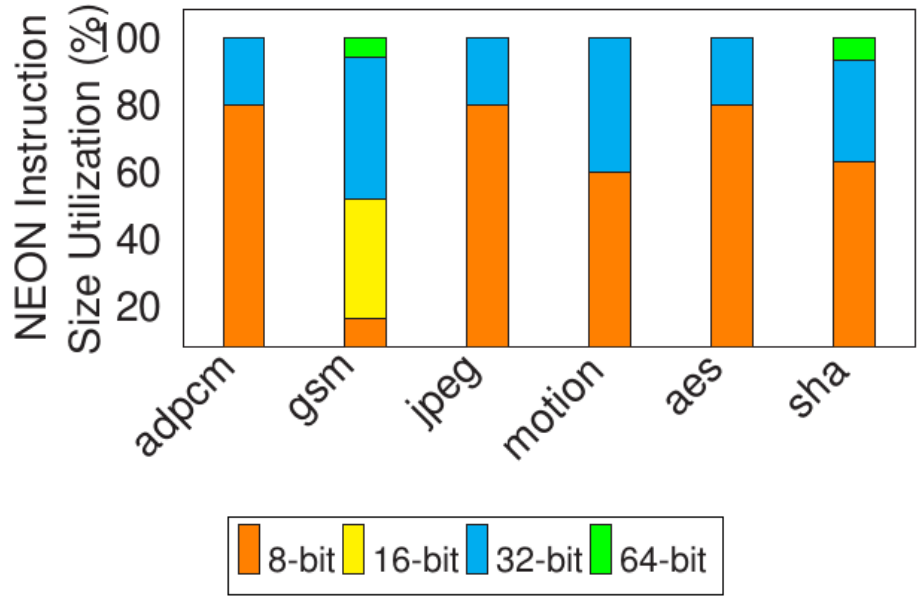
$$R > 95 \text{ and } G > 40 \text{ and } B > 20 \text{ and}$$
$$max\{R, G, B\} - min\{R, G, B\} > 15 \text{ and}$$
$$|R - G| > 15 \text{ and } R > G \text{ and } R > B$$

Study: Soft-NEON

- **An Interlay is reconfigurable!**
  - **ISA subsetting**
  - **Vector width customization**
  - **Operation folding**

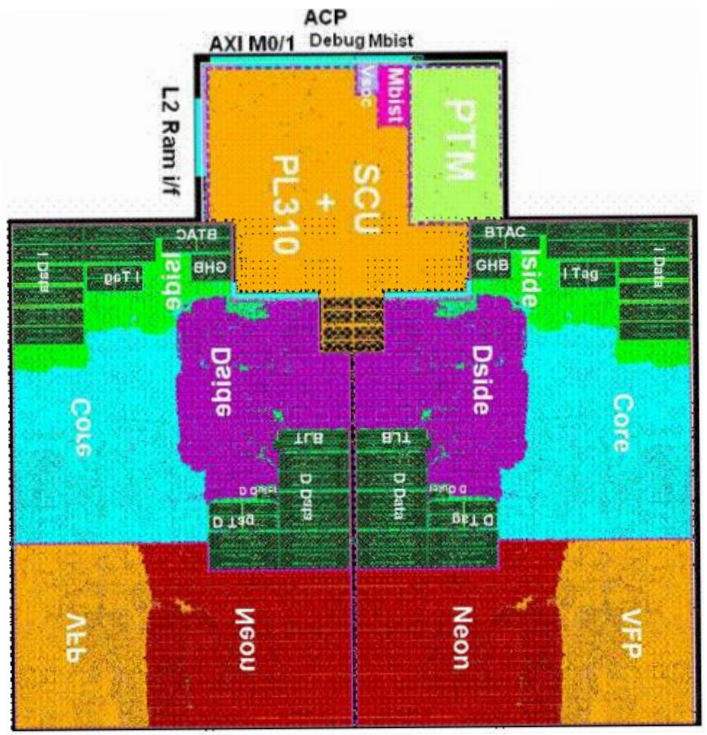




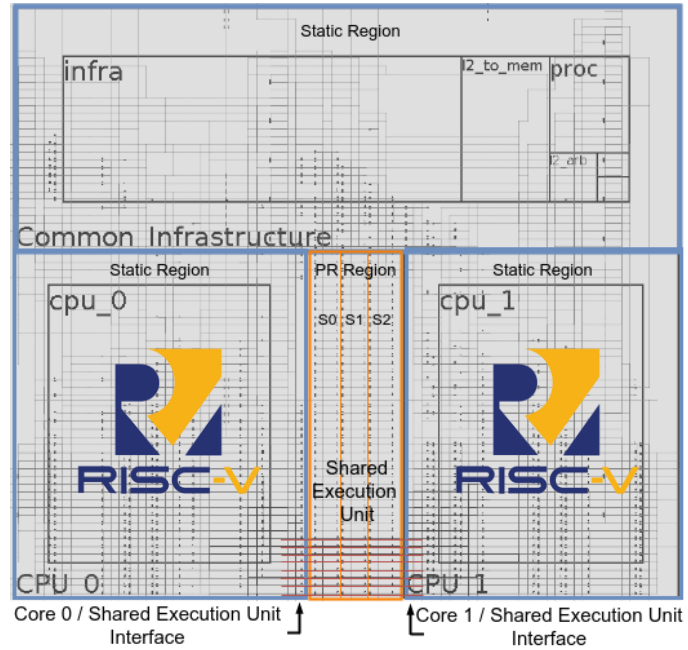


Custom Interlay – RISK-V Prototype

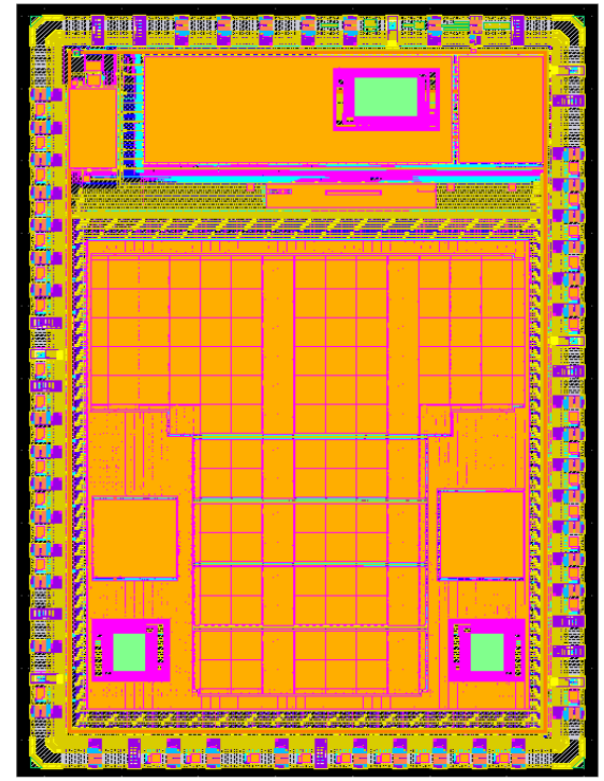
- **As a test vehicle, we implemented a dual-core RISC-V with a shared Interlay**



Cortex-A9 SoC Floorplan
NEON Area: 2080 LUTs, 16 DSPs, 16 BRAMs[2]

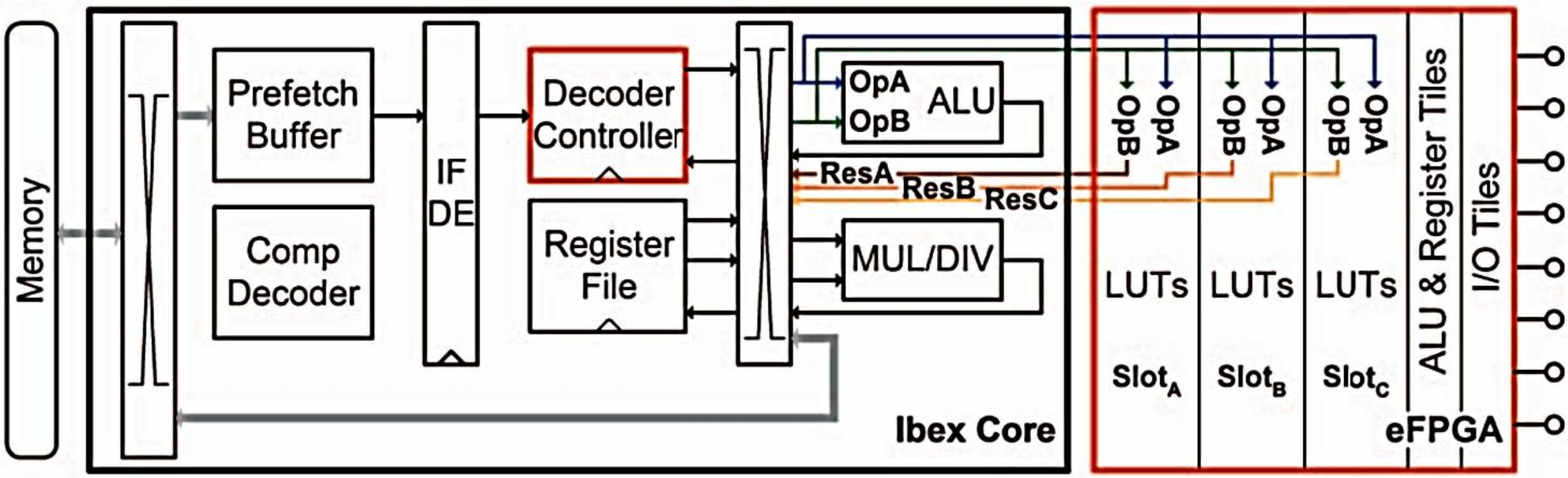


Dual-Core RISC-V with a PR shared custom unit floorplan
PR Shared Area: 2082 LUTs
Slot: 694 LUTs



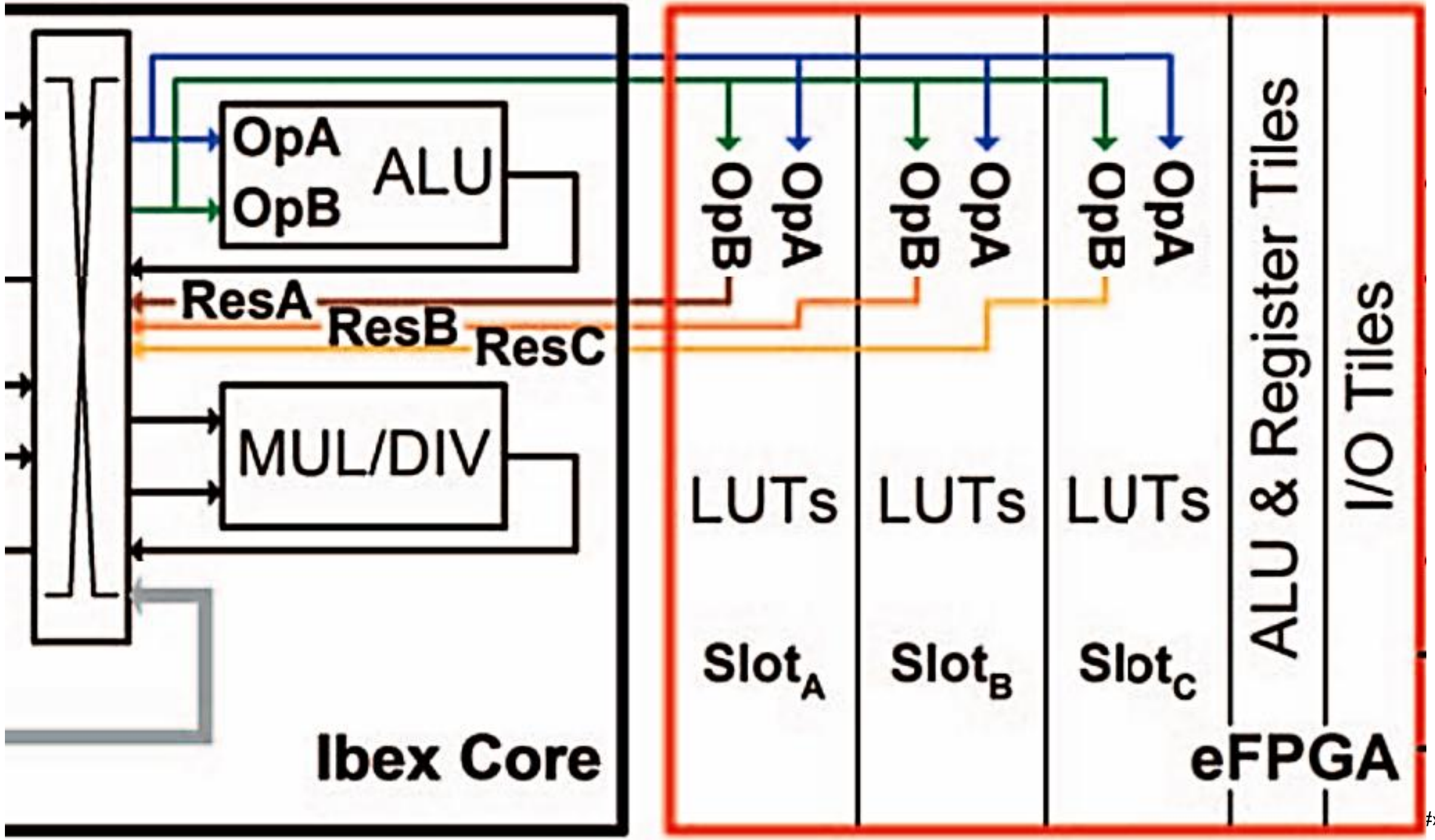
Dual-Ibex-Crypto-eFPGA for cryptography
Google Shuttle - MPW4
https://github.com/nguyendao-uom/ICESOC

Implementation



- Easy method: tab into the operands and multiplex in a result

  →may require different number clock cycles to evaluate


OpA
OpB
ALU
ResA
ResB
ResC
MUL/DIV
Ibex Core
OpB
OpA
OpB
OpA
OpB
OpA
LUTs
LUTs
LUTs
Slot$_A$
Slot$_B$
Slot$_C$
ALU & Register Tiles
I/O Tiles
eFPGA

Reconfigurable Instruction Set Extensions

The FlexBex (Ibex with eFPGA) approach:

We use the following instruction encoding:

eFPGA[*result_select*]**d**[*delay*] *dest, RS1, RS2* // **delay: 0…15 cycles**

- **Register manipulation instruction: dest ← RS1 OP RS2**
- **Inline assembly:**

```
int ina,inb,result;
ina = 10;
inb = 20;
asm volatile
(
  "␣eFPGA3d2␣␣%[z],␣%[x],␣%[y]\n\t"
  : [z] "=r" (result)
  : [x] "r" (ina), [y] "r" (inb)
)
```

Reconfigurable Instruction Set Extensions

Alternative: configurable instruction encoding:

Instruction	Slot	Delay slots
Encoding	[NULL/trap, 1, 2, ...]	[const, dynamic]
Encoding	[NULL/trap, 1, 2, ...]	[const, dynamic]
...		

Idea:

- **After configuring the CI, the OS/driver writes some registers to instruct the CPU ho to decode/use that instruction**

  → allows relocating instructions to different slots (defragmentation!)

  → allows slot-dependent latency

  → maybe key to port instructions to different process nodes

- **Problem: we can have an infinite number of CIs (overload Encoding?)**

ISA Extensions – Discussion

What we haven't discussed:

- **Stateless versus stateful instructions**

  → how do we do context switches? (an OS must be able to discover this)

  → does it need to support reentrant mode? (for recursion or multiple threads)

  → Simple case: `MULACC: acc ← acc + (RS1 x RS2)`

  (needs some thought how to recover state after context switch)

- **Prevent Deadlocks if instruction isn't available**

  → software fallback (usually traps)

- **How do we control configuration?**

  → explicit per config request?

  → implicit per trap? (e.g., run n-times emulation before configuration)

ISA Extensions – Discussion

What we haven't discussed:

- **Do we want to share instructions among cores?**

  → creates resource conflict (needs arbitration)

  → creates extra latency

  → may that cause deadlocks?

  → security (e.g., Spectre-kind of attacks)?

  → much more complex but possibly better resource utilization

- **Can tasks move to different cores?**

  → needs moving the configuration

- **Can customs instructions be ported among systems?**

- **Can we support iteration intervals of 1?**

ISA Extensions – Discussion

What we haven't discussed:

- **How do allocate reconfigurable resources among CIs?**

  → we may have multiple Tasks

- **Must be implemented efficient (or the benefit of CIs will be offset)**

  → instruction cycles & code density

  e.g.: if a CI saves 20 instructions each extra cycle eats 5% efficiency

  → worse in reality as we are not firing the CI all the time

  → CIs are usually doing more work but also more latency…

- **In summery: the potential using reconfigurable CIs is huge but it is non-trivial to feature this in a full-blown OS with multicore, multi tenancy, etc. support.**

Reconfigurable Instruction Set Extensions using FABulous eFPGAs - when and how

What is FABulous offering?

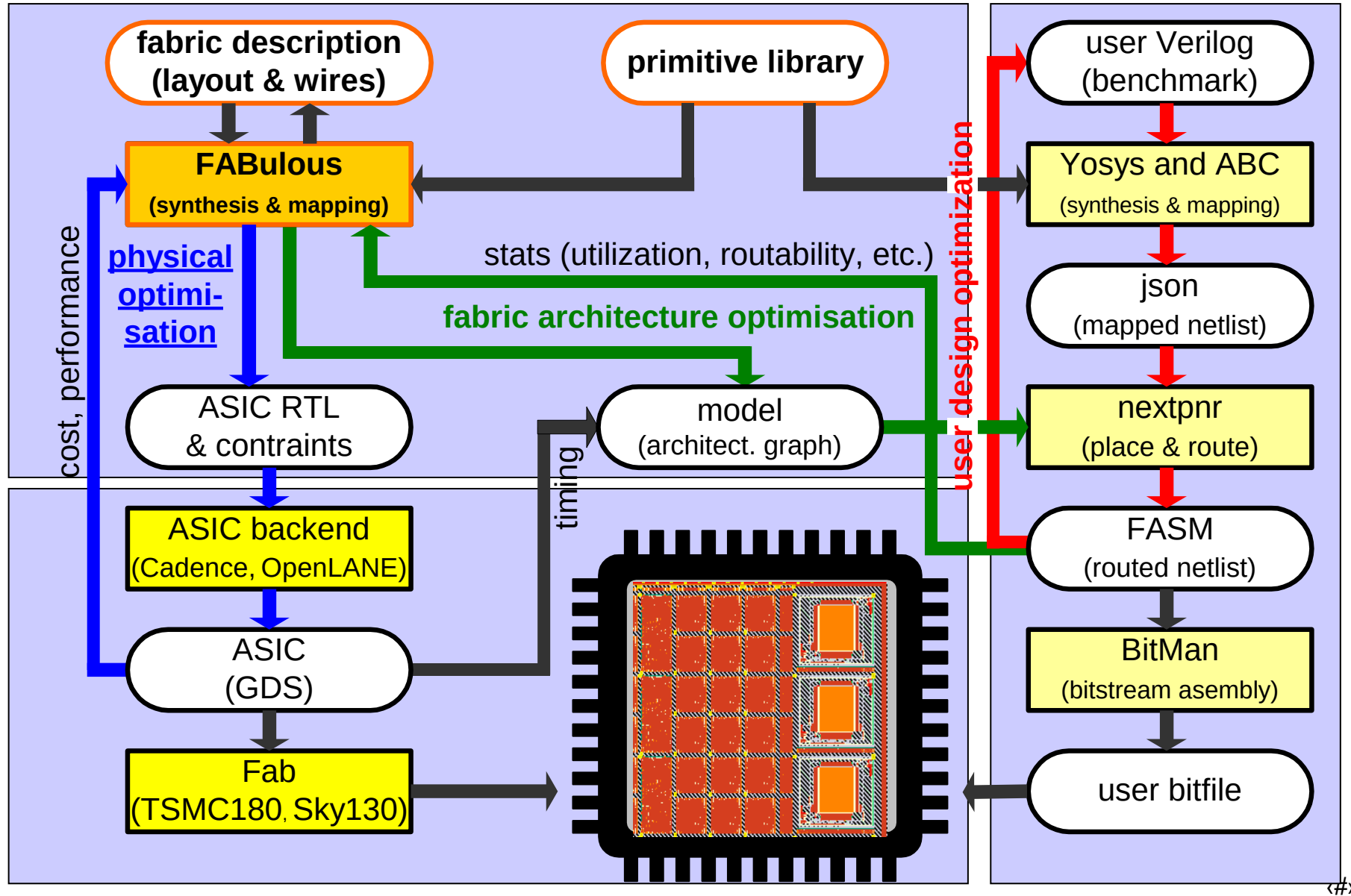
- Fully integrated open-source FPGA framework with good quality of results (area & performance)
- Entirely open and free, including commercial use (we integrated many other projects: Yosys, ABC, OpenRAM)
- Supports custom cells (if provided) → some tooling is on the way
- Supports partial reconfiguration
- Designed for ease of use while providing full control as needed
- Versatile
  - Different flows (OpenLane ←→Cadance) (Yosys/nextpnr ←→ ~~VPR)~~
  - Easy to customize, including the integration of own IP

The FABulous Framework

- Fully integrated framework for eFPGAs

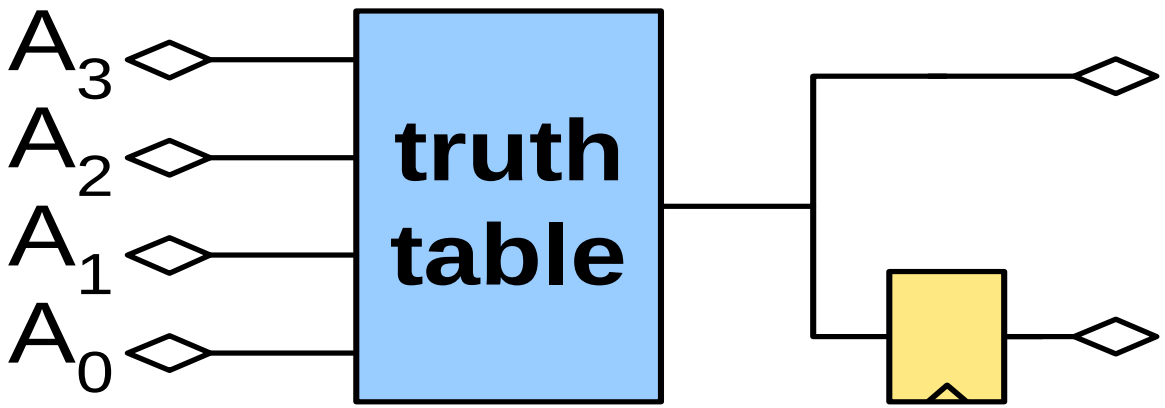
Uses many projects:

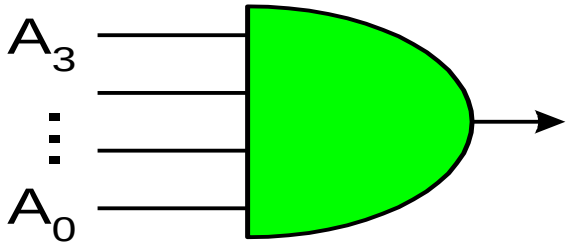
- Yosys & ABC
- nextpnr
- OpenLANE
- VPR
- OpenRAM
- Verilator

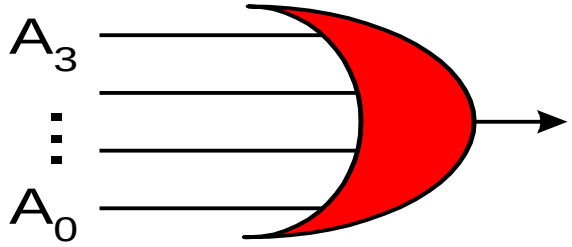


FPGA Basics – Logic

Look-up tables (LUTs) as the basic building block for implementing logic



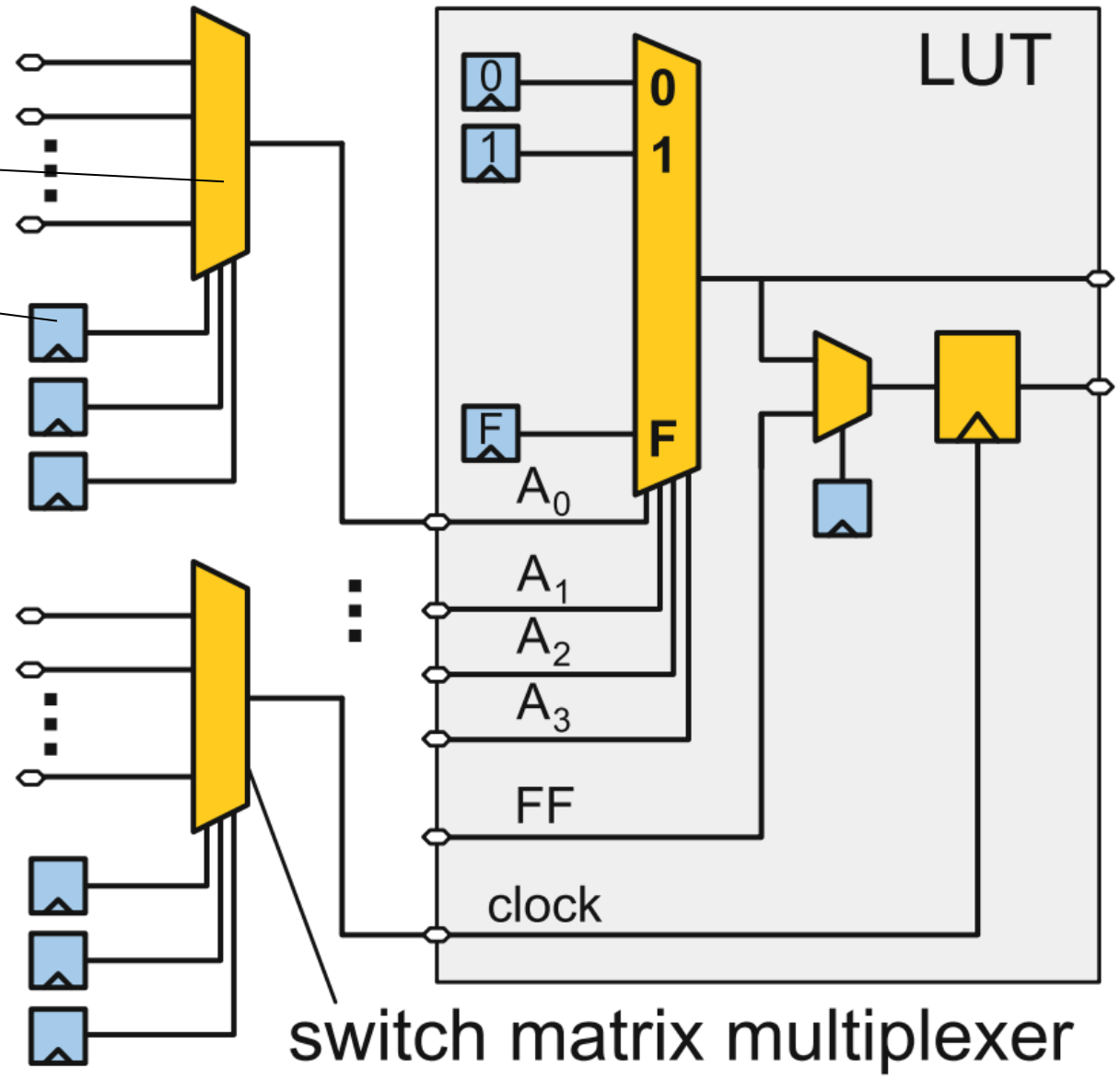




1	A_3, A_2, A_1, A_0	LUT-value AND gate	LUT-value OR gate
0	0000	0	0
1	0001	0	1
2	0010	0	1
3	0011	0	1
4	0100	0	1
5	0101	0	1
6	0110	0	1
7	0111	0	1
8	1000	0	1
9	1001	0	1
A	1010	0	1
B	1011	0	1
C	1100	0	1
D	1101	0	1
E	1110	0	1
F	1111	1	1

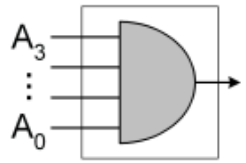
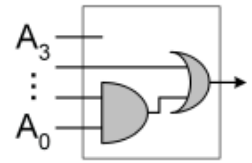
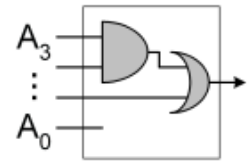
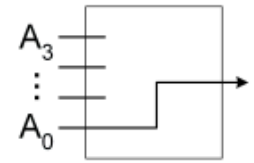
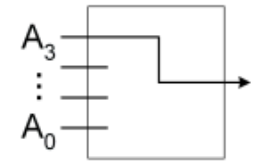
Routing

- FPGAs are made vastly of:
  - (wide) **Multiplexers**
  - Configuration **Latches**
- Customizing these *tactical cells** provides most efficiency gain



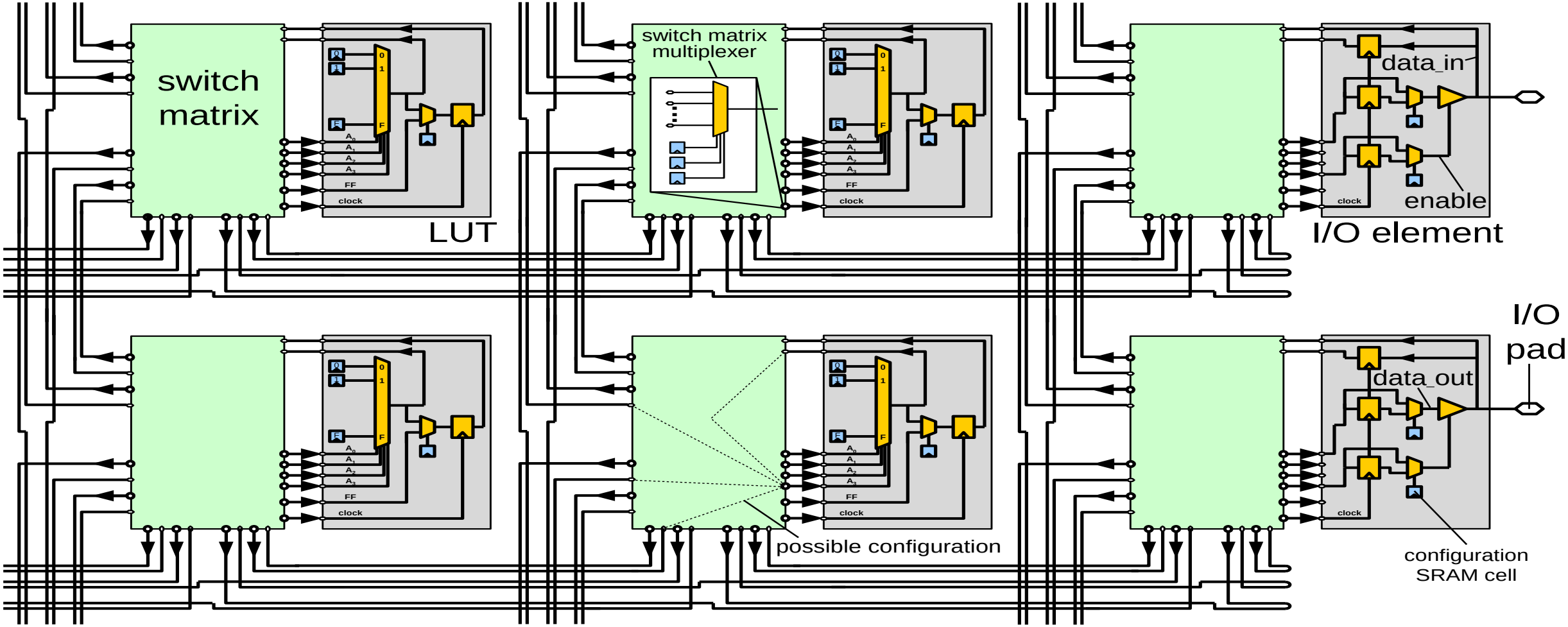
*Victor Aken'Ova. 2005. Bridging the Gap Between Soft and Hard eFPGA Design. MSc Thesis. UBC

LUTs help with the routing (pin swaps are for free)

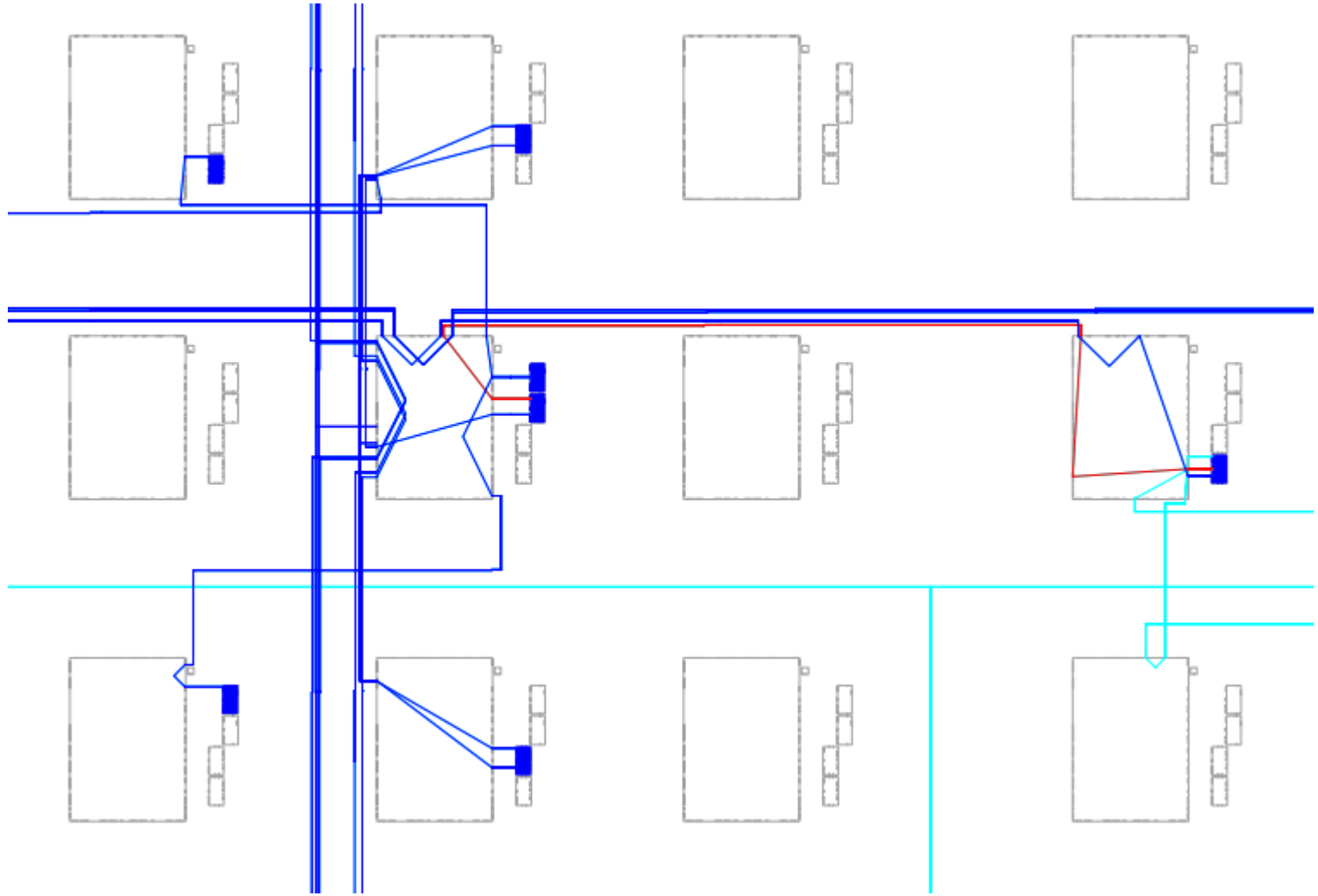
	A_3	A_2	A_1	A_0					
0:	0	0	0	0	0	0	0	0	0
1:	0	0	0	1	0	0	0	1	0
2:	0	0	1	0	0	0	1	0	0
3:	0	0	1	1	0	1	1	1	0
4:	0	1	0	0	0	1	0	0	0
5:	0	1	0	1	0	1	0	1	0
6:	0	1	1	0	0	1	1	0	0
7:	0	1	1	1	0	1	1	1	0
8:	1	0	0	0	0	0	0	0	1
9:	1	0	0	1	0	0	0	1	1
A:	1	0	1	0	0	0	1	0	1
B:	1	0	1	1	0	1	1	1	1
C:	1	1	0	0	0	1	1	0	1
D:	1	1	0	1	0	1	1	1	1
E:	1	1	1	0	0	1	1	0	1
F:	1	1	1	1	1	1	1	1	1

FPGA Basics – FPGA Fabric

- Example of an FPGA fabric composed of LUTs, switch matrices and I/O cells. Other common primitives: memories, multipliers, transceivers, …



FPGA Basics – FPGA Fabric



FABulous UI

File Edit View

FABulous Fabric

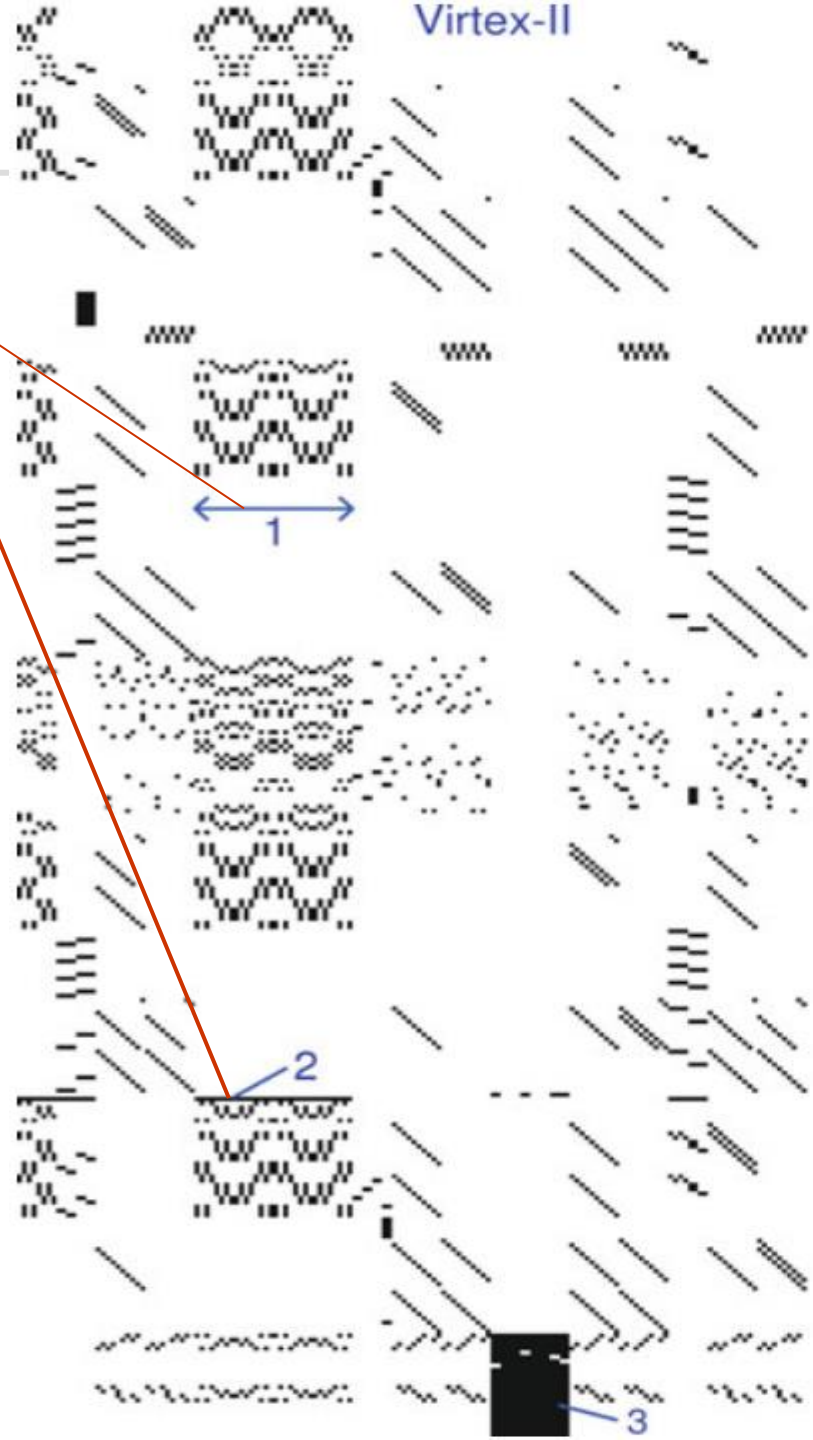
Fabric Elements | HDL Code

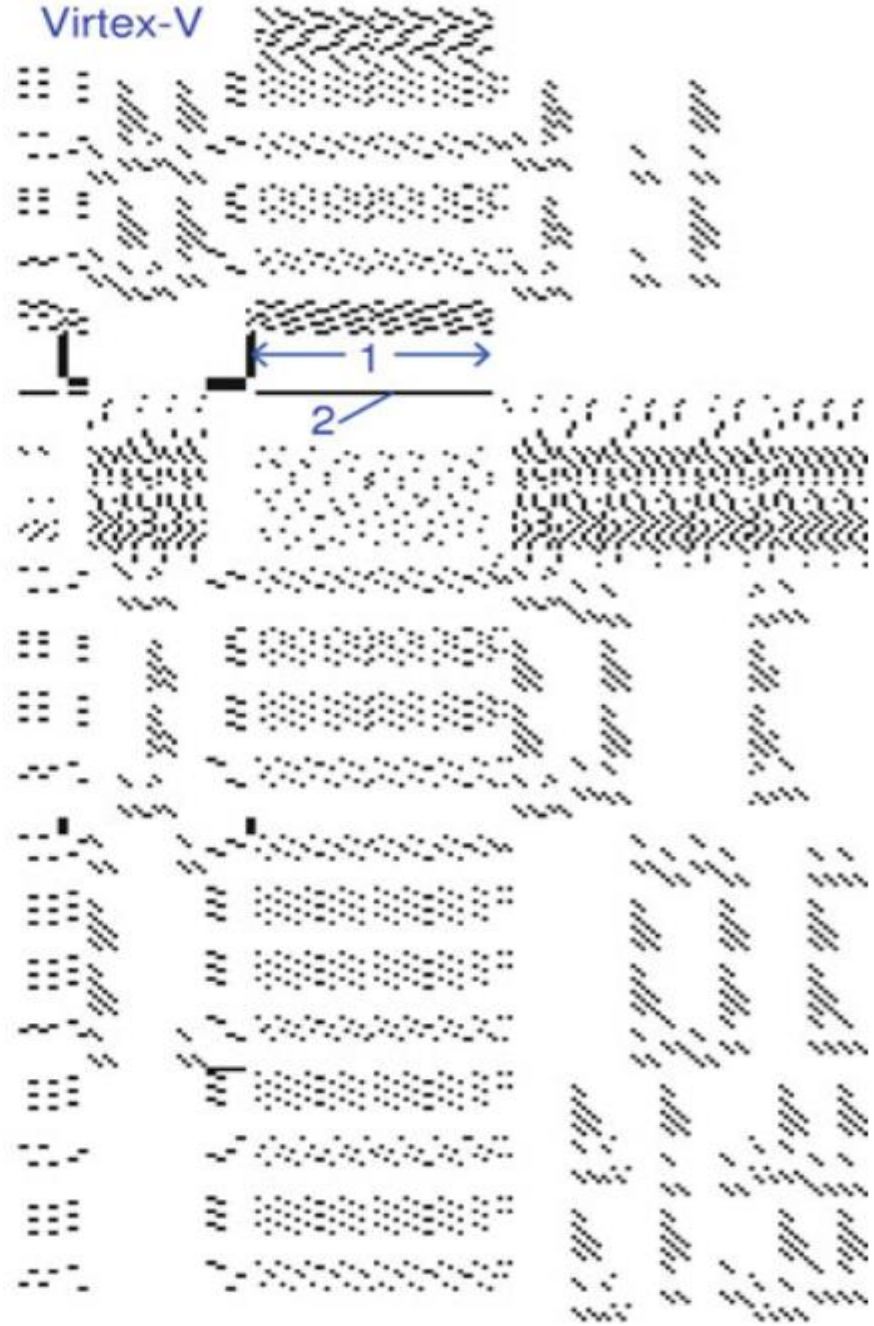
```
// Copyright 2021 University of Manchester
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

(*FABulous, BelMap,
INIT=0,
INIT_1=1,
INIT_2=2,
INIT_3=3,
INIT_4=4,
INIT_5=5,
INIT_6=6,
INIT_7=7,
INIT_8=8,
INIT_9=9,
INIT_10=10,
INIT_11=11,
INIT_12=12,
INIT_13=13,
INIT_14=14,
INIT_15=15,
FF=16,
IOmux=17,
SET_NORESET=18
*)
module LUT4c_frame_config_dffesr (I0, I1, I2, I3, O, Ci, Co, SR, EN, UserCLK, ConfigBits)
        parameter NoConfigBits = 19   // has to be adjusted manually (we don't use an arithmetic p
        // IMPORTANT: this has to be in a dedicated line
        input I0  // LUT inputs
        input I1
        input I2
        input I3
        output O  // LUT output (combinatorial or FF)
        input Ci  // carry chain input
        output Co  // carry chain output
        input SR  // SHARED_RESET
        input EN  // SHARED_ENABLE
        (* FABulous, EXTERNAL, SHARED_PORT *) input UserCLK  // EXTERNAL // SHARED_PORT // ## the
        // GLOBAL all primitive pins that are connected to the switch matrix have to go before th
        (* FABulous, GLOBAL *) input [NoConfigBits-1 : 0] ConfigBits

        localparam LUT_SIZE = 4
        localparam N_LUT_flops = 2 ** LUT_SIZE

        wire [N_LUT_flops-1 : 0] LUT_values
        wire [LUT_SIZE-1 : 0] LUT_index
        wire LUT_out
        reg LUT_flop
        wire I0mux  // normal input '0', or carry input '1'
        wire c_out_mux, c_I0mux, c_reset_value  // extra configuration bits

        assign LUT_values = ConfigBits[15:0]
        assign c_out_mux  = ConfigBits[16]
        assign c_I0mux = ConfigBits[17]
        assign c_reset_value = ConfigBits[18]

//CONFout <= c_I0mux;

        //assign I0mux = c_I0mux ? Ci : I0;
        my_mux2 my_mux2_I0mux(
        .A0(I0),
        .A1(Ci),
        .S(c_I0mux),
        .X(I0mux)
```

Switch matrix

1. LUT input muxes
2. Constant input value
3. LUT and Flop output muxes

- Rest: local routing
- Virtex II
  - 332 inputs
  - 160 multiplexer
- Virtex V
  - 305 inputs
  - 172 Multiplexer





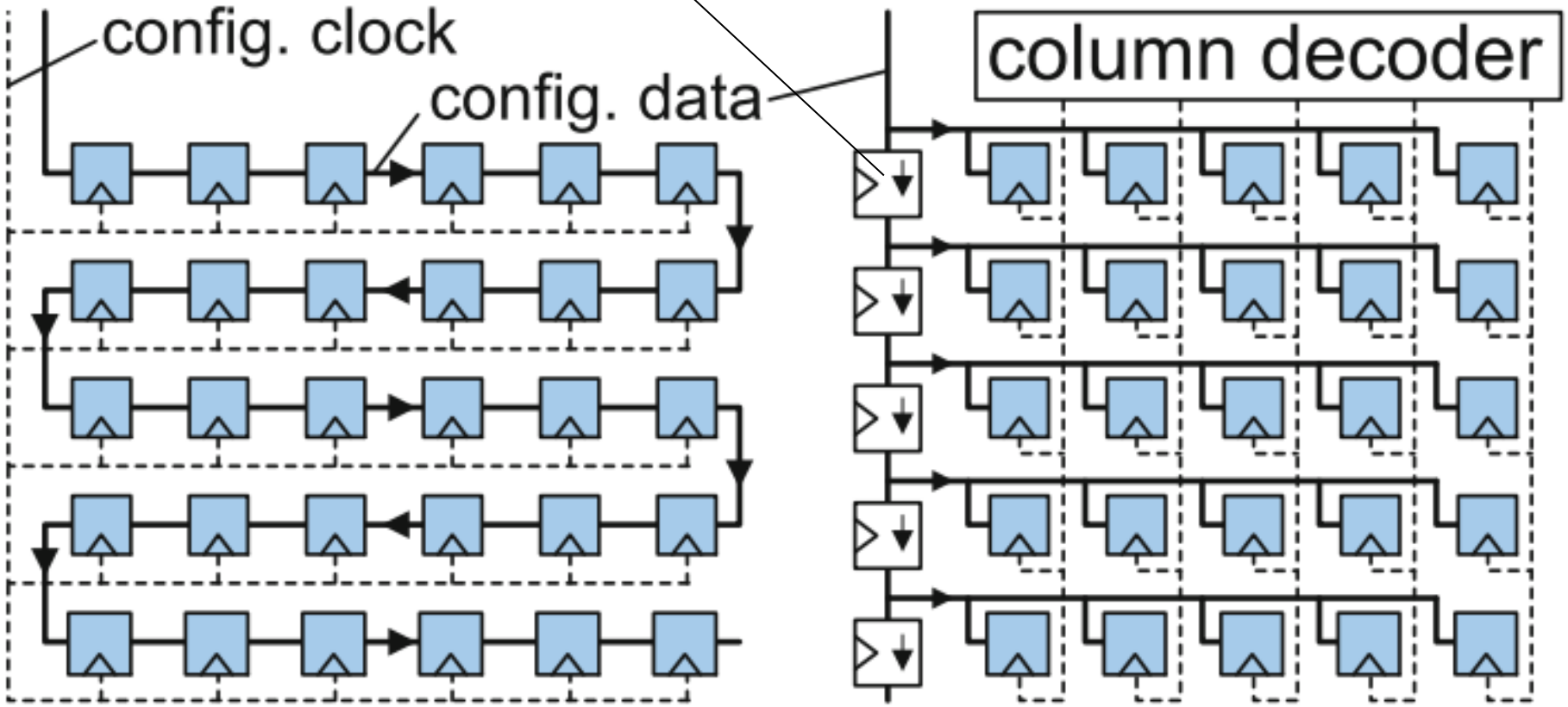
Rough Cost Estimate

In total per CLB					12,560	1,600
Virtex-6			Transistors/mux		In total	
Routing resource	# muxes	# inputs	Pass	Conf.	Trans.	Conf. bits
LUT inputs	48	24	29	50	3,792	480
FF_in	8	28	34	55	712	88
D_in	4	22.5*	27.5	50	310	40
CLK and GFAN	4	14	18	40	232	32
WE|CR|SE	2|2|2	24|23|12	29|28|16	50|50|35	416	54
Local routing	96	20.5*	25.5	50	7,248	960
Longlines	4	12*	16	35	204	28
Sum routing					12,914	1,682
LUT truth table		Trans.: 8×448	Conf. bits: 8×64		3,584	512
In total per CLB					16,498	2,194

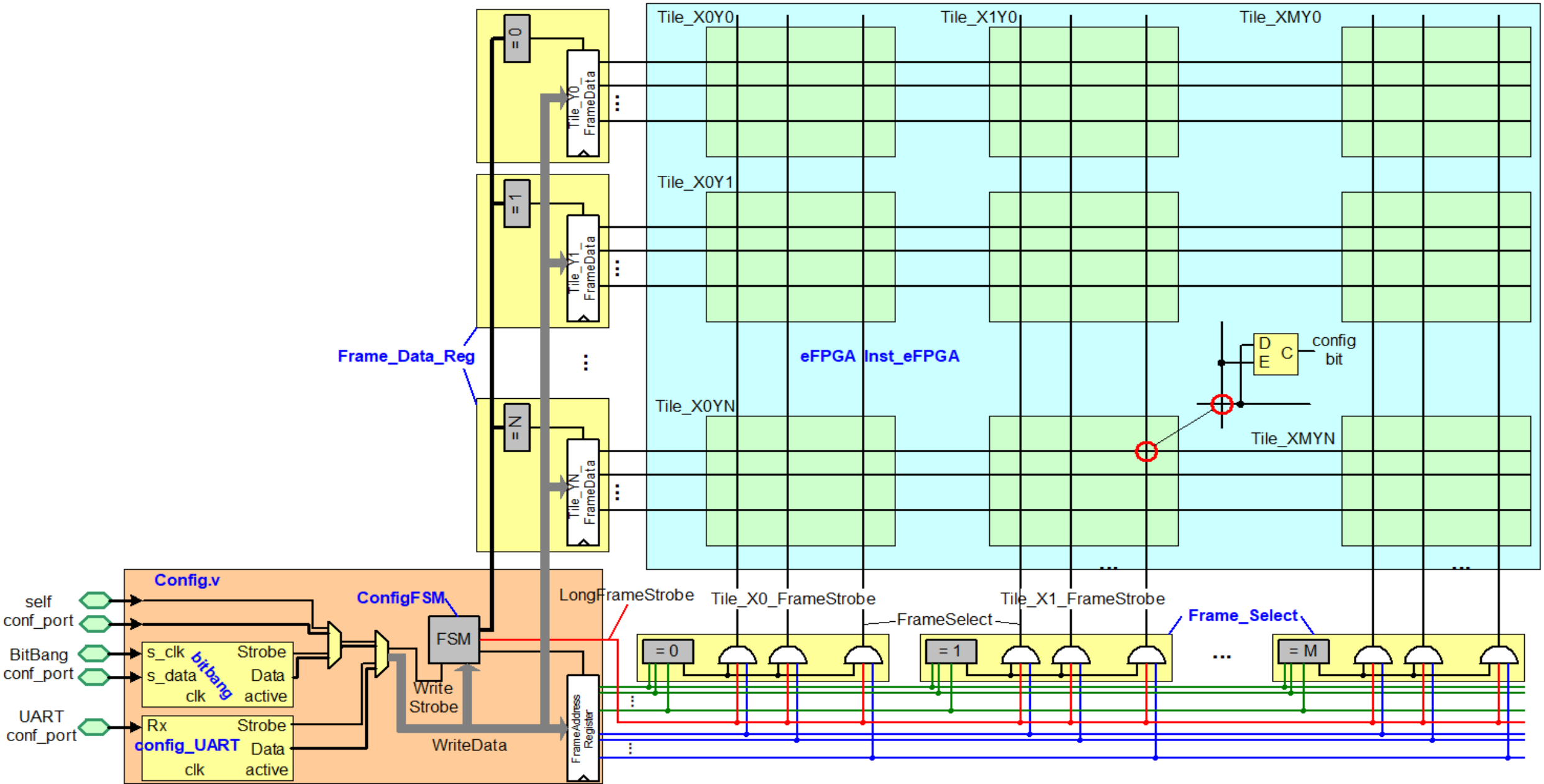
*Average value

FPGA Configuration

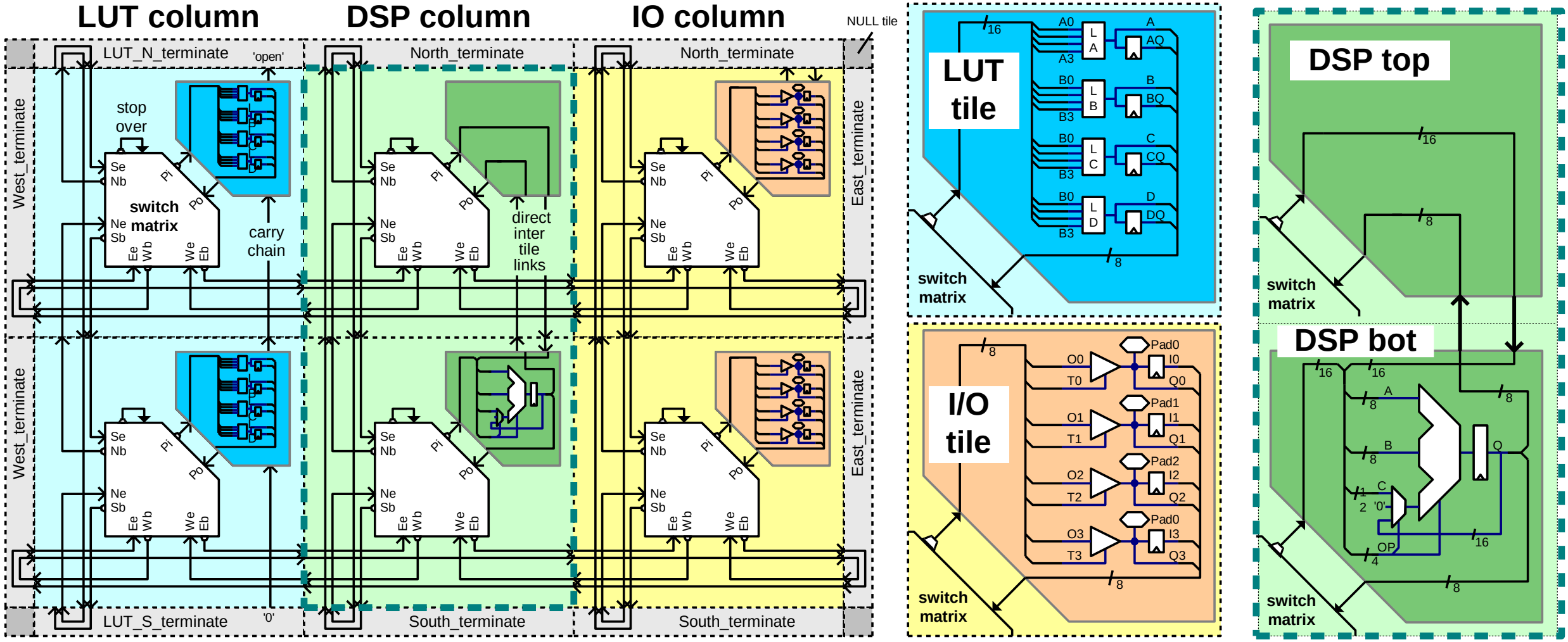
- The easiest way to implement configuration storage is using a shift register
- Bit-wise addressing is way too expensive!
  → frame-based reconfiguration
- But how do we update individual switch matrix multiplexers?



FPGA Configuration (as used in FABulous)

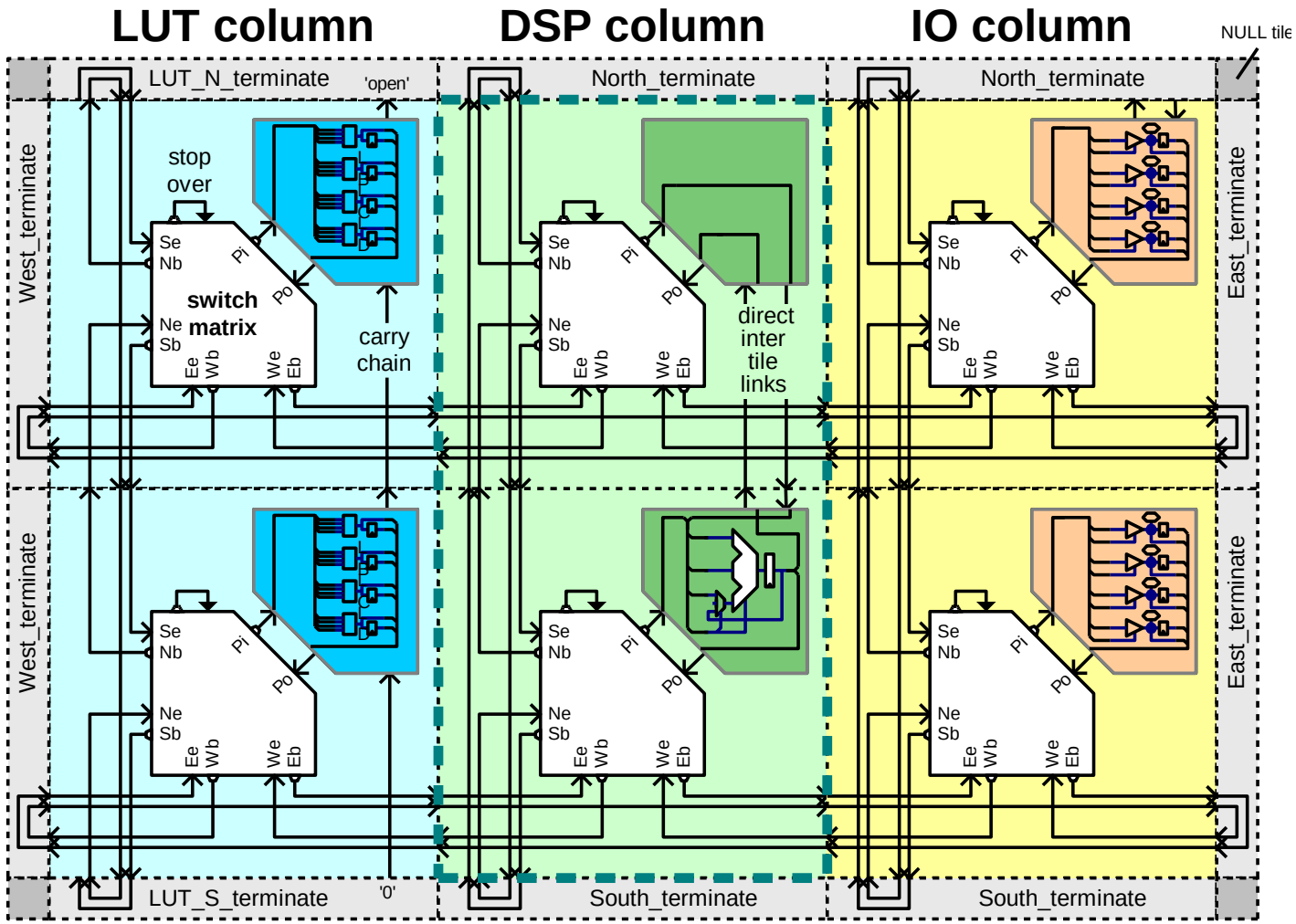


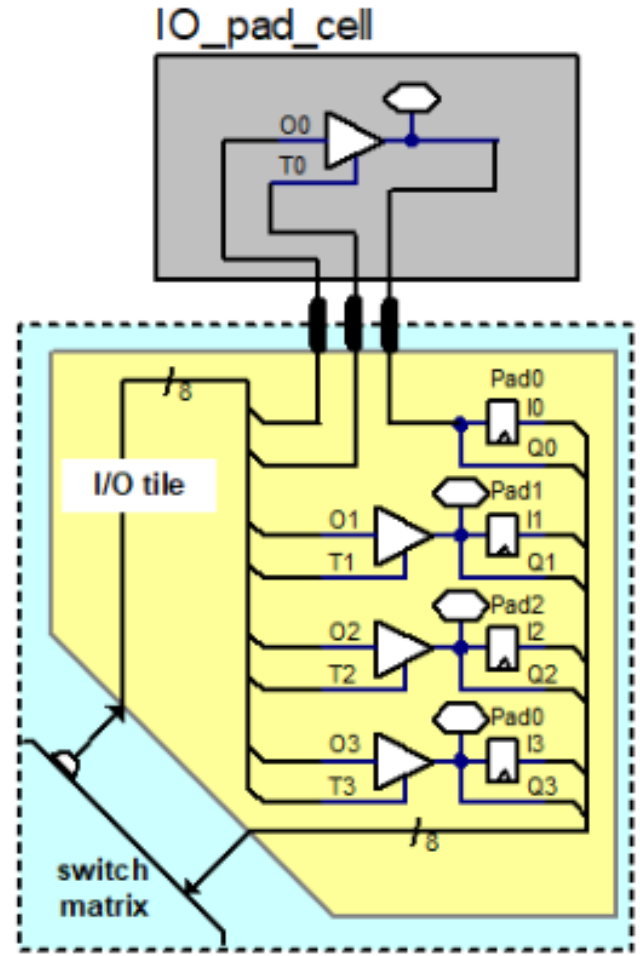
Basic concepts



- Basic tiles have same height, but type-specific width (for logic tiles, DSPs, etc.)
- Adjacent tiles can be fused for more complex blocks (see the DSP example) →Supertile

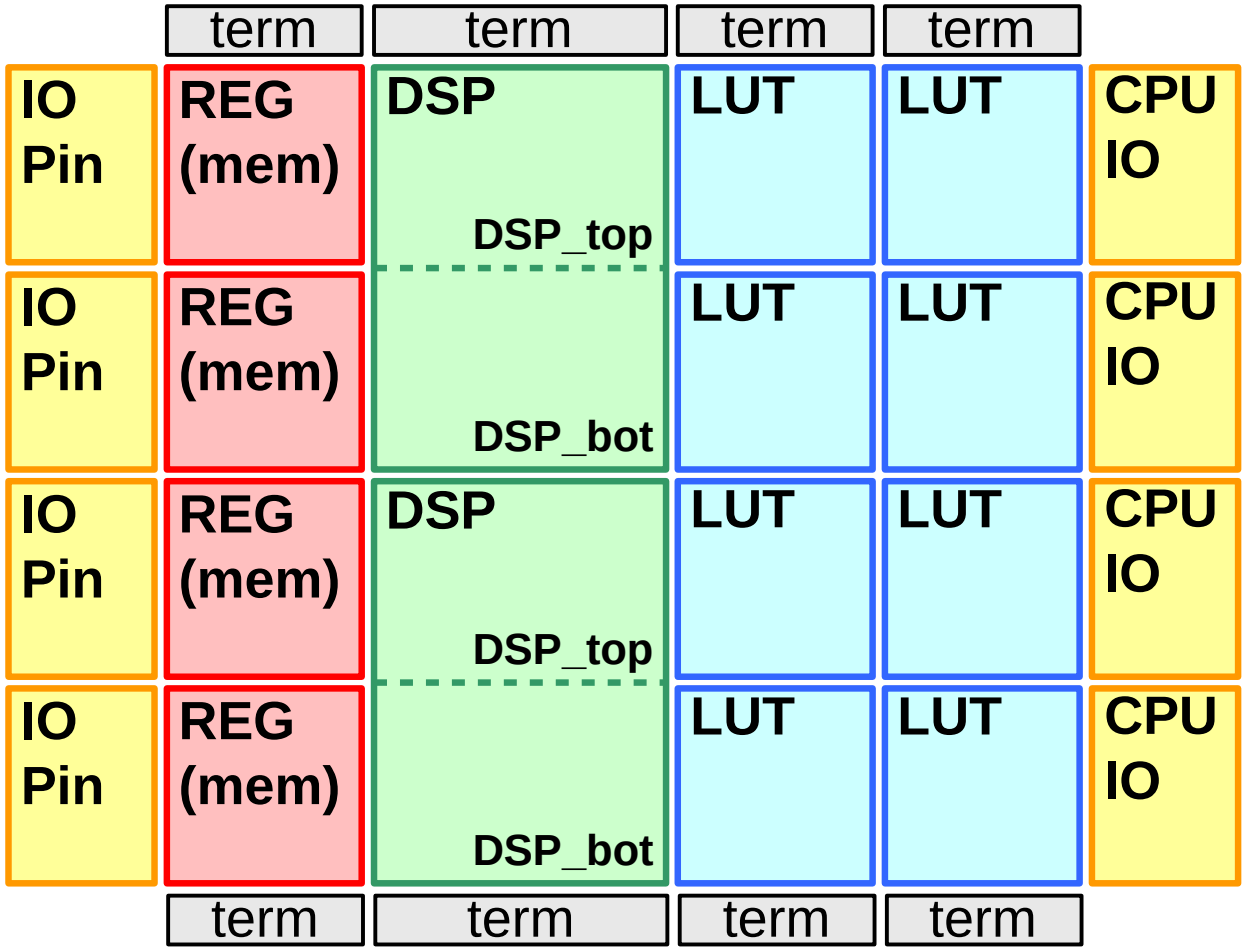
Basic concepts





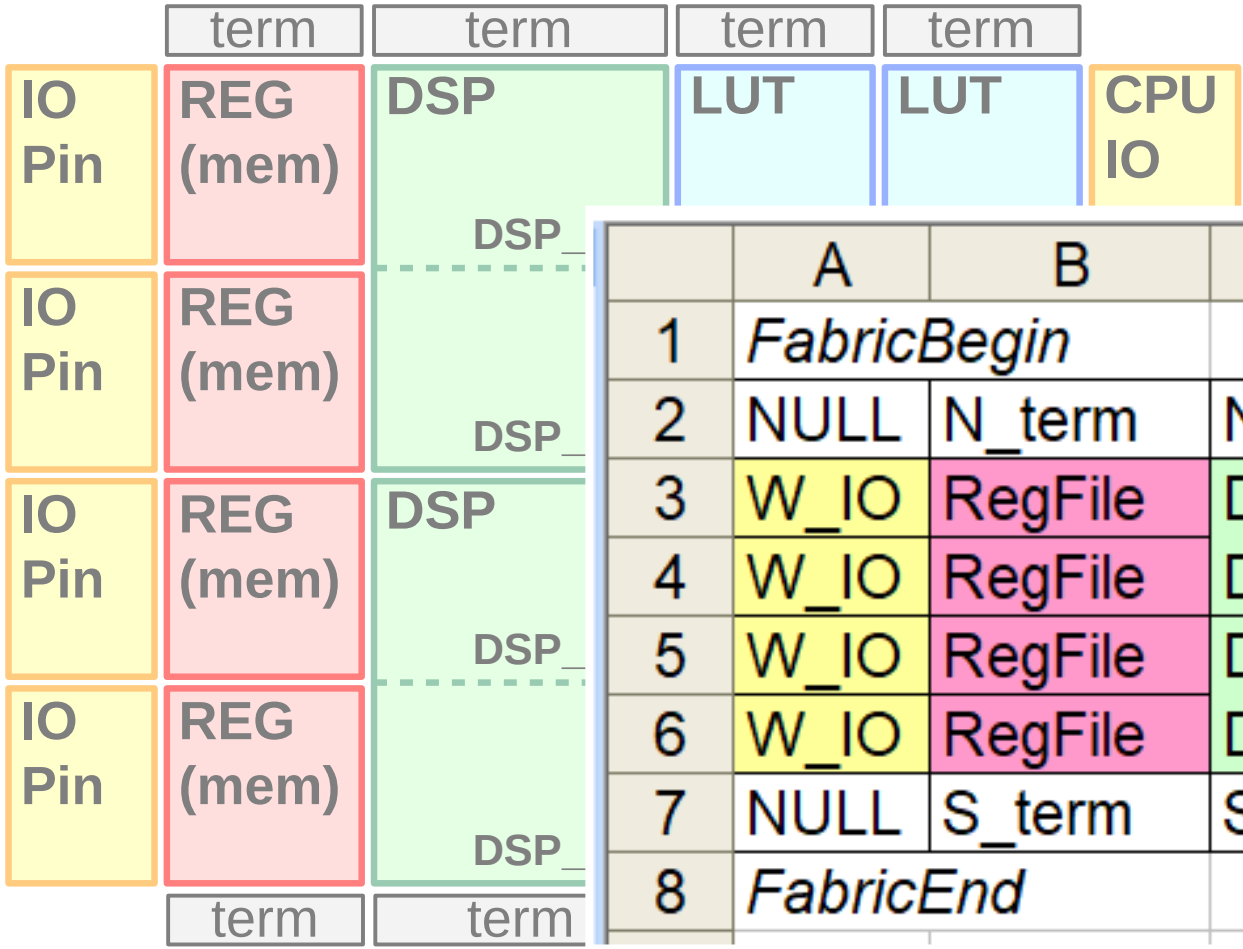
- I/Os belong logically to the fabric but are physically routed to the surrounding
- Internal wires, buses, etc. are „just“ wires at the border of the fabric

Let‘s build a small eFPGA: Fabric Definition



- 4 x register file, 2 x DSPs, 4 x LUTs (CLB), I/Os left and right,

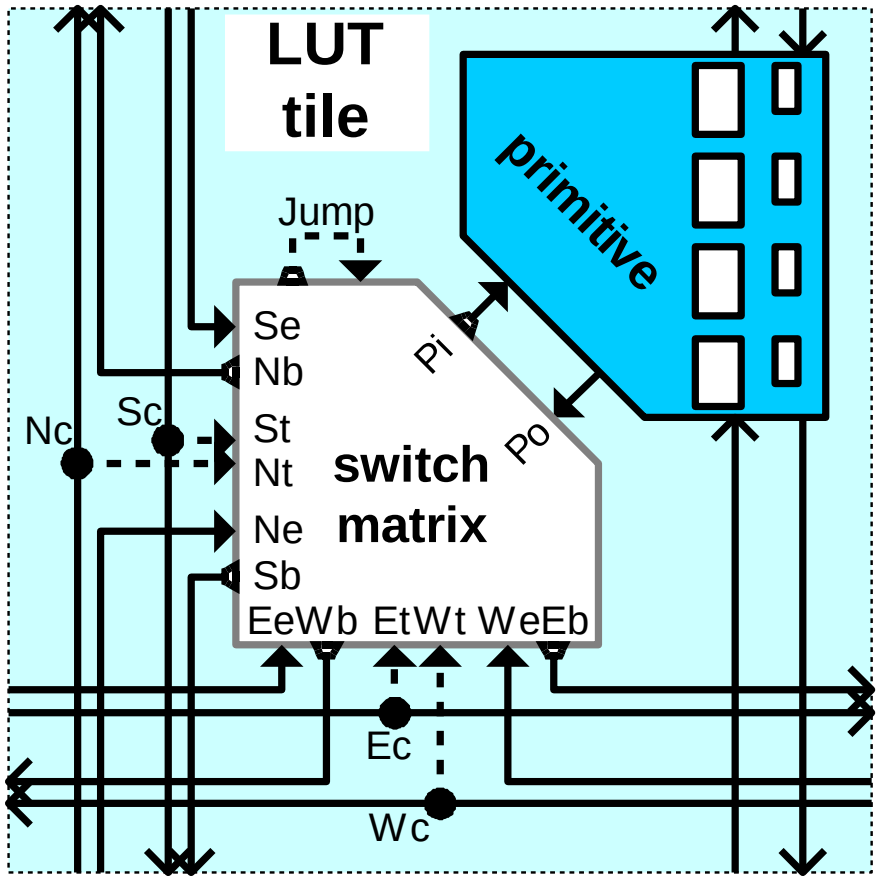
Let‘s build a small eFPGA: Fabric Definition



	A	B	C	D	E	F
1	*FabricBegin*					
2	NULL	N_term	N_term	N_term	N_term	NULL
3	W_IO	RegFile	DSP_top	LUT4AB	LUT4AB	CPU_IO
4	W_IO	RegFile	DSP_bot	LUT4AB	LUT4AB	CPU_IO
5	W_IO	RegFile	DSP_top	LUT4AB	LUT4AB	CPU_IO
6	W_IO	RegFile	DSP_bot	LUT4AB	LUT4AB	CPU_IO
7	NULL	S_term	S_term	S_term	S_term	NULL
8	*FabricEnd*					

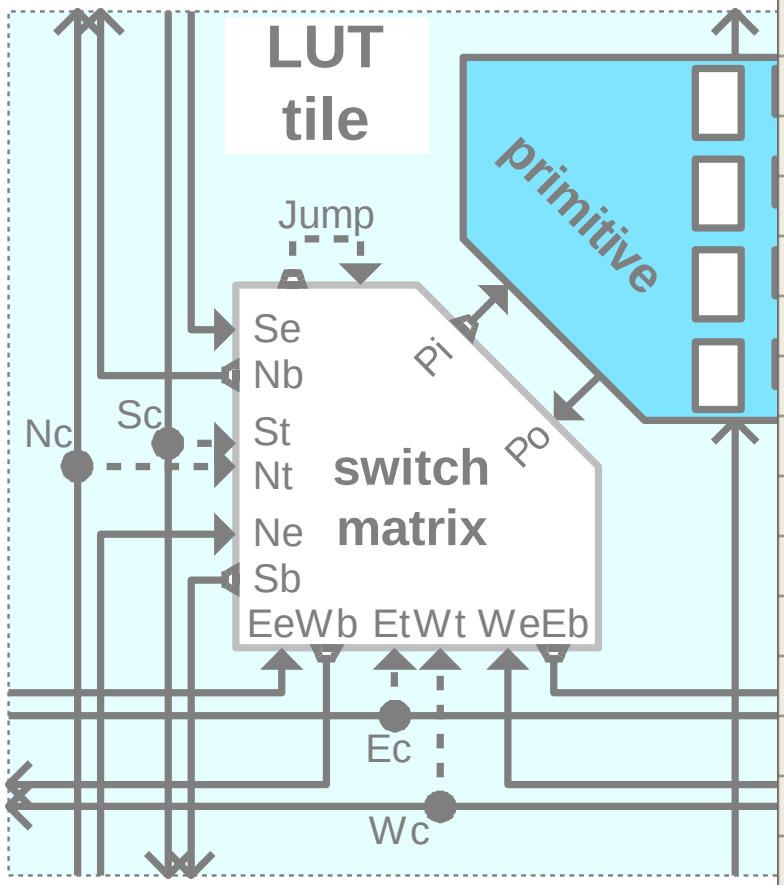
- 4 x register file, 2 x DSPs, 8 x LUT-tiles (CLB), I/Os left and right,
- A fabric is modelled as a spreadsheet (tiles are references to tile descriptors)

Let‘s build a small eFPGA: Tile Definition



- Wires
- Primitives (basic elements)
- Switch matrix
- Configuration storage

Let‘s build a small eFPGA: Tile Definition

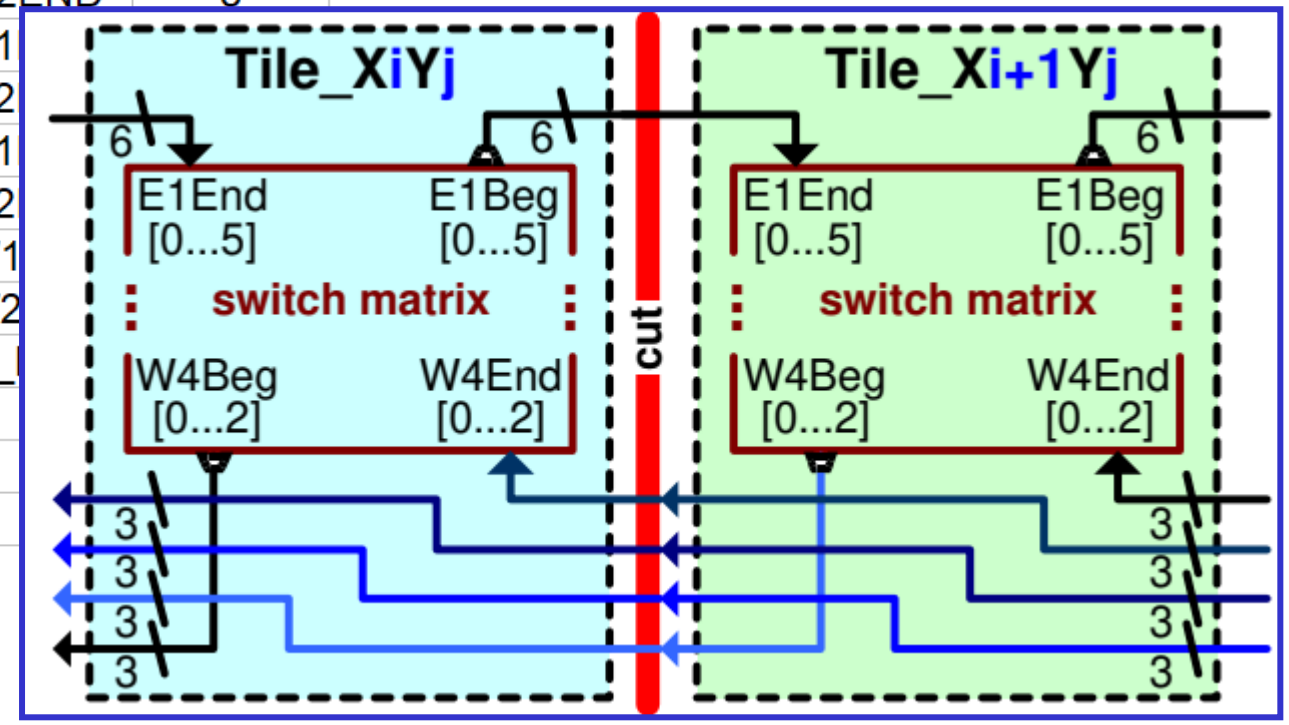


121	*TILE*	LUT4AB					
122	#direction	source	X-offset	Y-offset	destination	wires	
123	NORTH	N1BEG	0	1	N1END	8	
124	NORTH	N2BEG	0	2	N2END	4	
125	NORTH	Co	0	1	Ci	1	# carry
126	EAST	E1BEG	1	0	E1END	8	
127	EAST	E4BEG	4	0	E4END	2	
128	SOUTH	S1BEG	0	-1	S1END	8	
129	SOUTH	S2BEG	0	-2	S2END	4	
130	WEST	W1BEG	-1	0	W1END	8	
131	WEST	W4BEG	-4	0	W4END	2	
132	JUMP	J_BEG	0	0	J_END	42	
133	BEL	LUT4.vhdl	LA_				
134	BEL	LUT4.vhdl	LB_				
135	BEL	LUT4.vhdl	LC_				
136	BEL	LUT4.vhdl	LD_				
137	BEL	MUX8LUT.vhdl					
138	MATRIX	LUT4AB_switch_matrix.vhdl					
139	*EndTILE*						

- Wires
- Primitives (basic elements)
- Switch matrix
- Configuration storage

eFPGA Ecosystem – Tile/Wire Definitions

10	TILE	CLB				
11	#direction	source_name	X-offset	Y-offset	destination	wires
12	NORTH	N1BEG	0	1	N1END	3
13	NORTH	N2BEG	0	2	N2END	3
14	EAST	E1BEG	1	0	E1	
15	EAST	E2BEG	2	0	E2	
16	SOUTH	S1BEG	0	-1	S1	
17	SOUTH	S2BEG	0	-2	S2	
18	WEST	W1BEG	-1	0	W1	
19	WEST	W2BEG	-2	0	W2	
20	JUMP	J_BEG	0	0	J_	
21	BEL	clb_slice_4xLUT4.vhdl				
22	MATRIX	CLB_switch_matrix.VHDL				
23	EndTILE					



- Wires are defined by
  <direction> <symbolic begin|end names> <target offset> <# wires>
- Jump wires for hierarchical routing (Intel/Altera and Xilinx UltraScale style)

eFPGA Ecosystem – Switch Matrix Definition

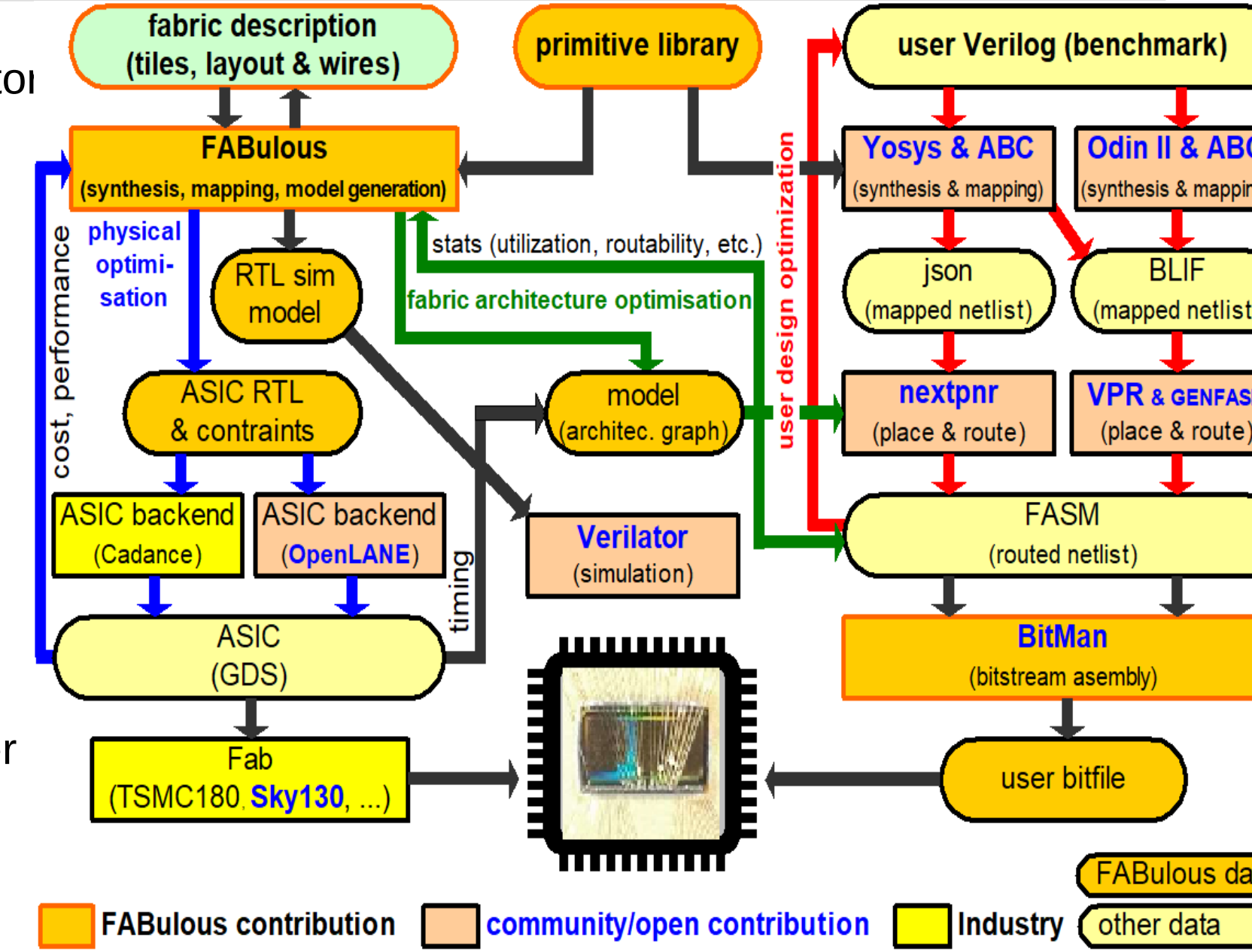
```
# LUT4AB
# double with MID cascade : [N,E,S,W]2BEG --- [N,E,S,W]2MID -> [N,E,S,W]2BEGb --- [N,E,S,W]2END (
[N|E|S|W]2BEGb[0|1|2|3|4|5|6|7],[N|E|S|W]2MID[0|1|2|3|4|5|6|7]

############## LUT Inputs ##############
############## LUT Inputs ##############
############## LUT Inputs ##############

# shared double MID jump wires
J2MID_ABa_BEG[0|0|0|0],[JN2END3|N2MID6|S2MID6|W2MID6]
J2MID_ABa_BEG[1|1|1|1],[E2MID2|JE2END3|S2MID2|W2MID2]
J2MID_ABa_BEG[2|2|2|2],[E2MID4|N2MID4|JS2END3|W2MID4]
J2MID_ABa_BEG[3|3|3|3],[E2MID0|N2MID0|S2MID0|JW2END3]


# Carry chain  Ci -> LA_Ci-LA_Co -> LB_Ci-LB_Co -> ... ->
LA_Ci,Ci0
L[B|C|D|E|F|G|H]_Ci,L[A|B|C|D|E|F|G]_Co
Co0,LH_Co
```

- Describes the adjacency in a symbolic way
  <mux_output>,<mux_input>
- Alternatively adjacency matrix

	A	B	C	D	E	F
1	CLB	N1END0	N1END1	N1END2	N2END0	N2EN
2	N1BEG0	0	1	1	1	1
3	N1BEG1	1	0	1	0	0
4	N1BEG2	1	0	1	0	1
5	N2BEG0	0	1	0	1	0
6	N2BEG1	1	0	0	0	0
7	N2BEG2	1	1	1	0	0
8	N4BEG0	0	1	0	1	1
9	N4BEG1	1	1	1	1	1
10	E1BEG0	1	0	1	0	1
11	E1BEG1	1	1	0	1	1

The FABulous eFPGA Ecosystem

- FABulous eFPGA generator
  - ASIC RTL and constraints generation
  - Generating models for nextpnpr/VPR flows
  - FPGA emulation
- Virtex-II, Lattice clones (patent-free!)
- See our FPGA 2021 paper „FABulous: An Embedded FPGA Framework"



The first open-everything FPGA

- Built using open tools

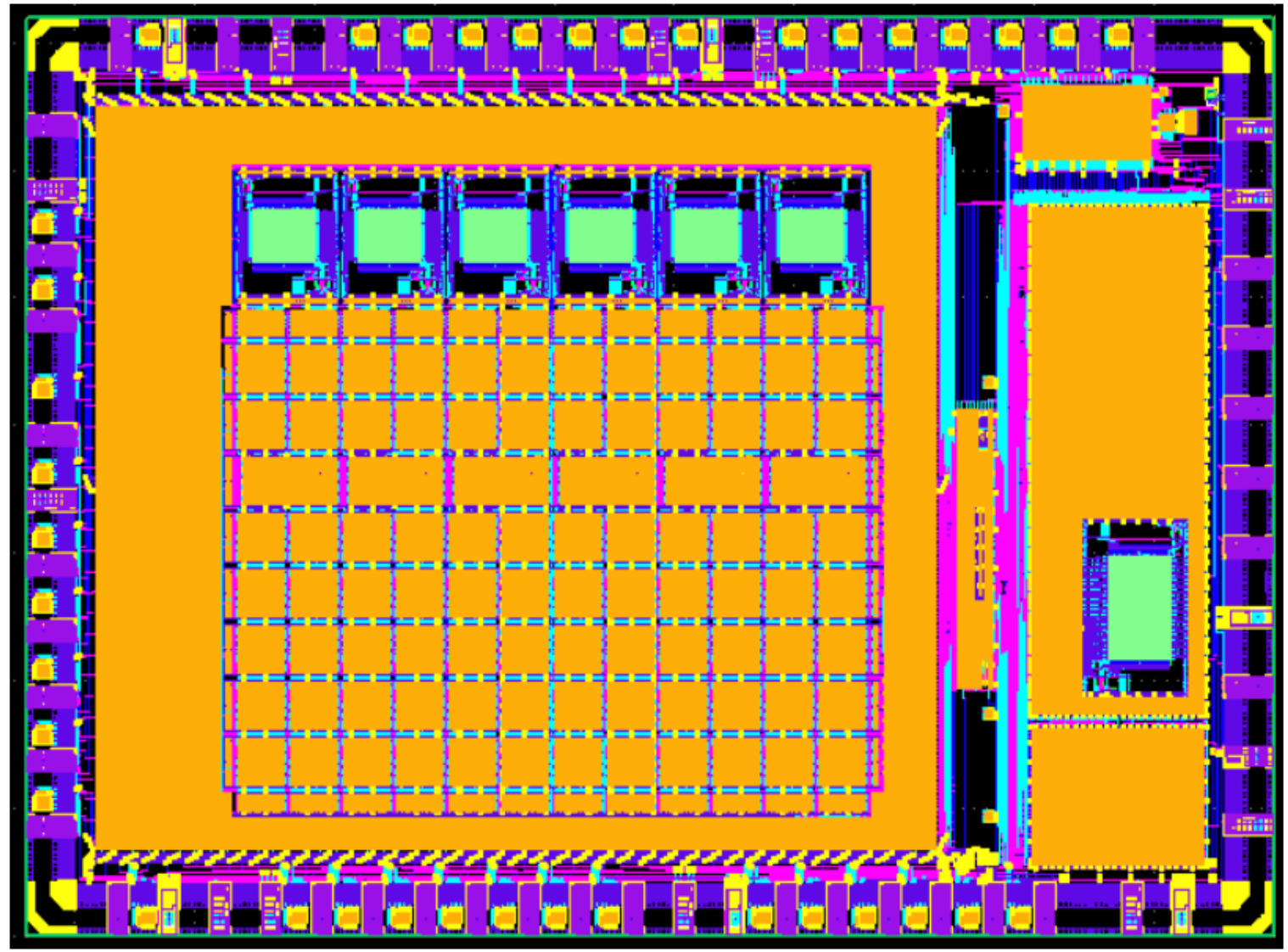
  (Yosys, OpenLane, Verilator…)

- Open PDK

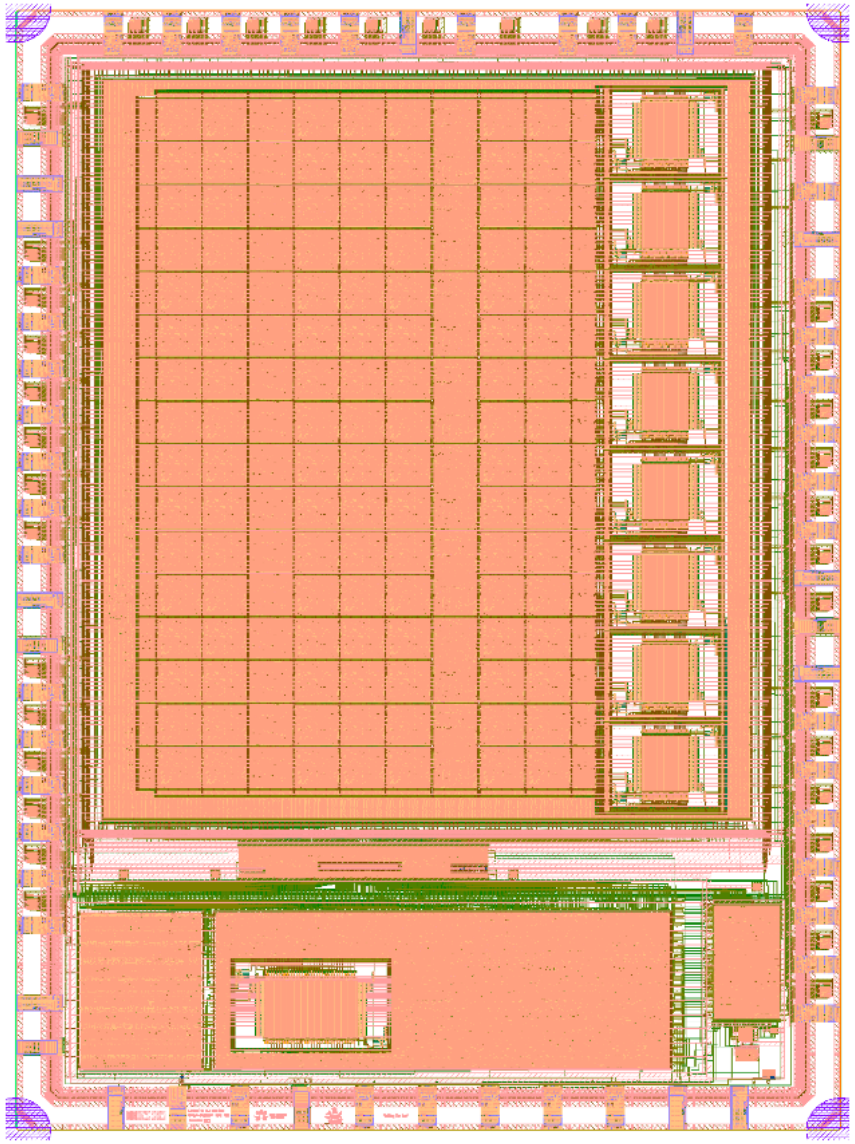
  (Skywater 130 process)

- Google Shuttle (MPW5):

  https://github.com/nguyendao-uom/open_eFPGA



FABulous Chip Gallery



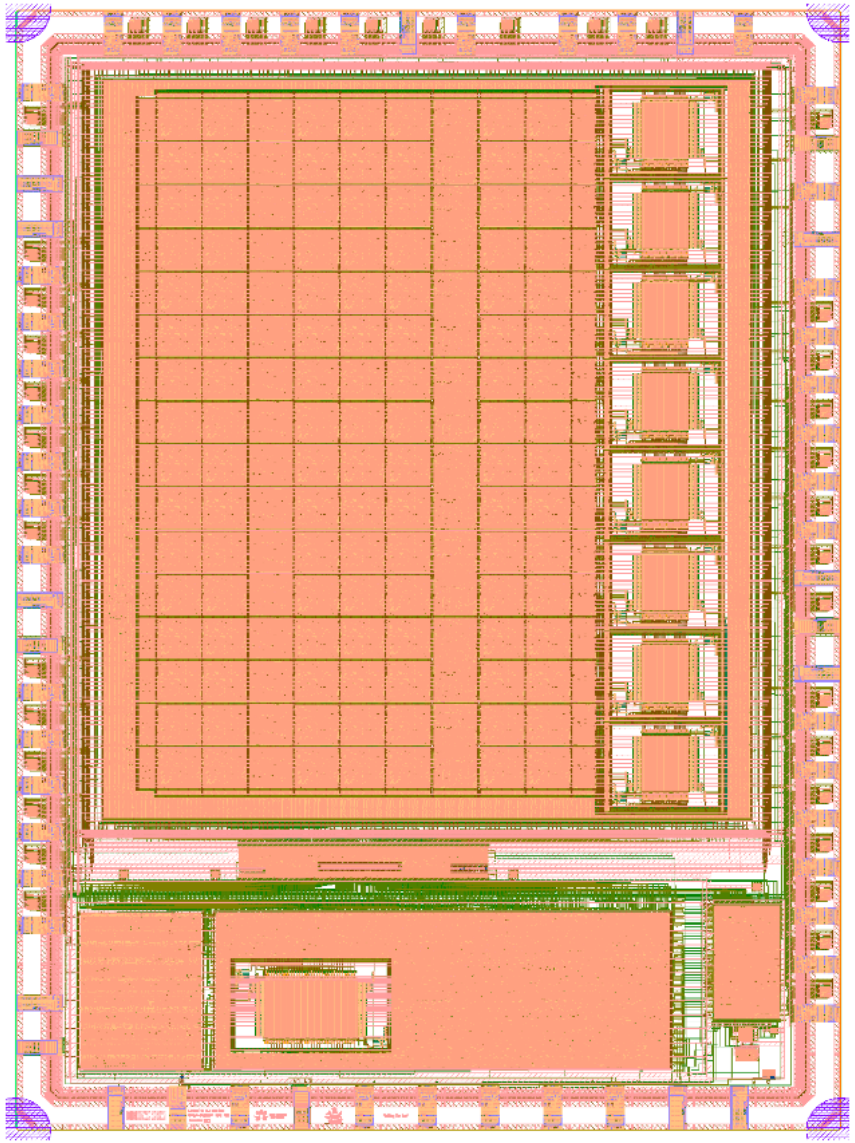
Sky130 with CLBs, DSPs, RegFiles, BRAMs

Google Shuttle - MPW-2

(can implement RISC-V)

https://github.com/nguyendao-uom/eFPGA_v3_caravel

FABulous Chip Gallery

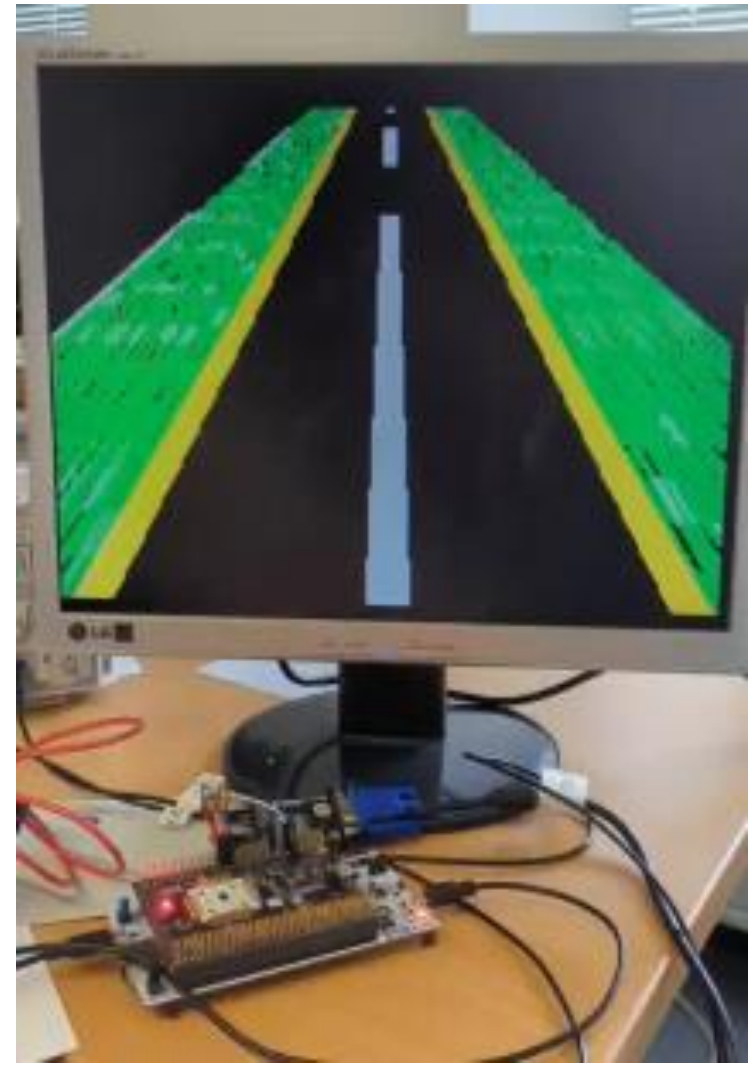


Sky130 with CLBs, DSPs, RegFiles, BRAMs

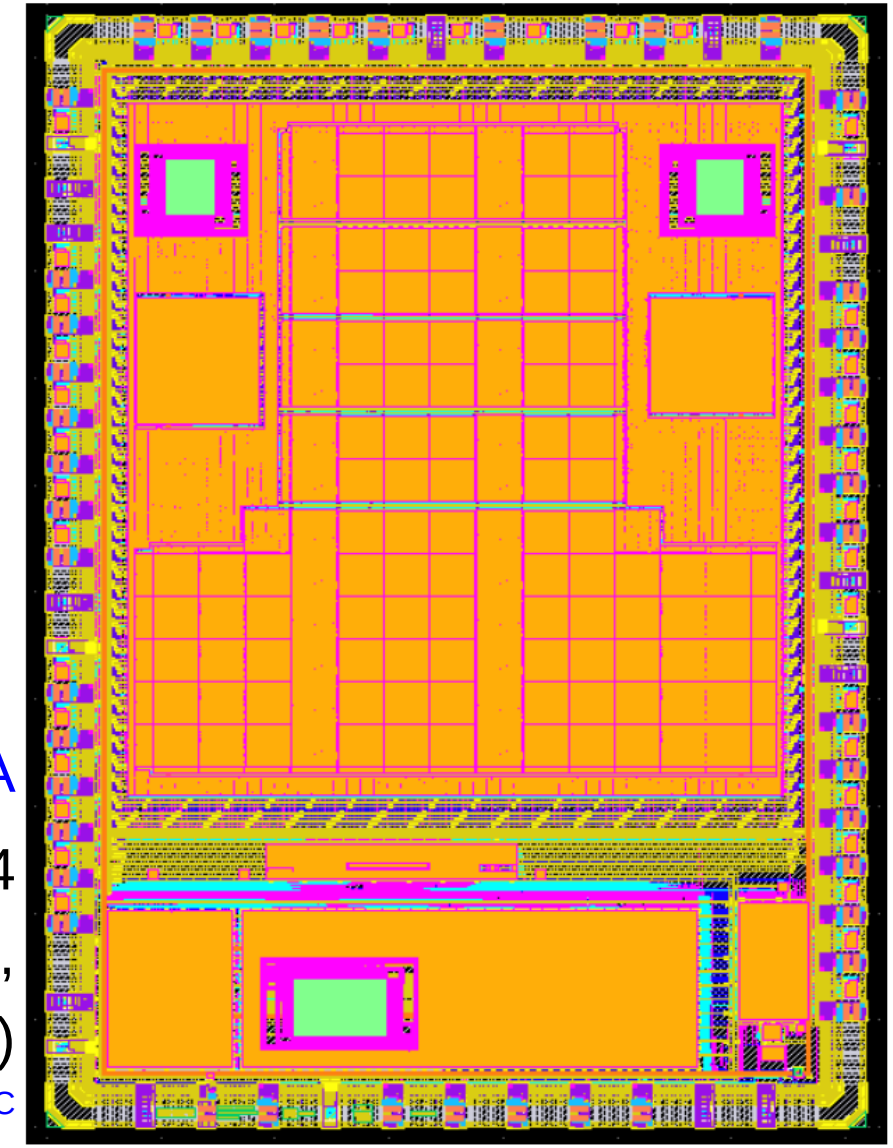
Google Shuttle - MPW-2

(can implement RISC-V)

https://github.com/nguyendao-uom/eFPGA_v3_caravel



FABulous Chip Gallery



Dual-Ibex-Crypto-eFPGA

Google Shuttle - MPW-4

(custom instructions,

T-shaped fabric)

https://github.com/nguyendao-uom/ICESOC

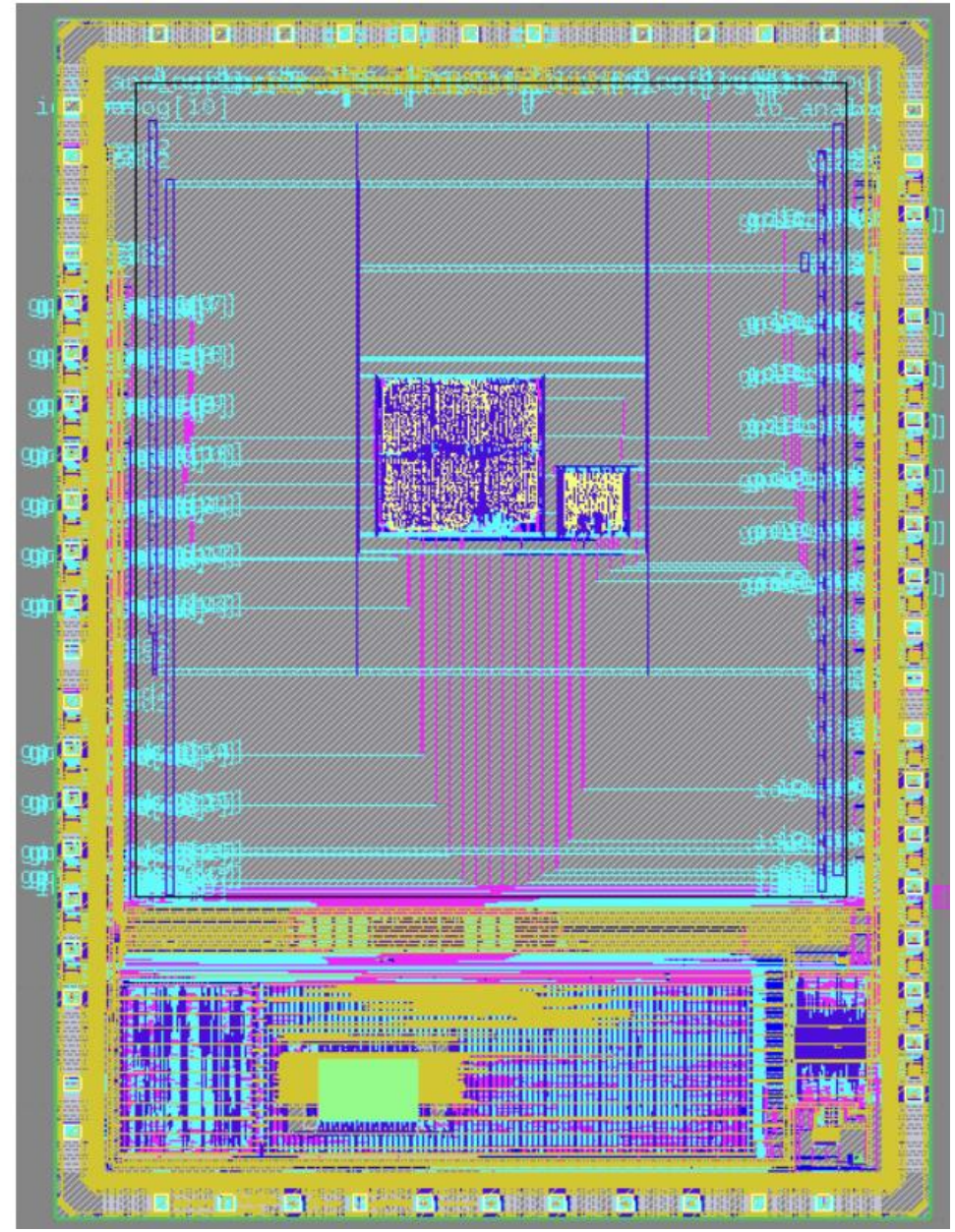
FABulous Chip Gallery

Open ReRAM FPGA test chip

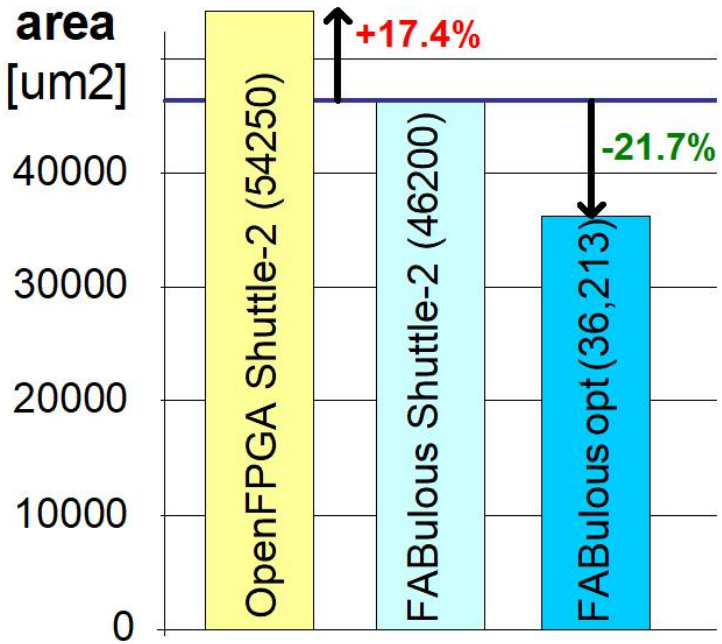
- Sky130, Google Shuttle MPW4https://github.com/nguyendao-uom/rram_testchip
- Just enough logic to send „Hello World“ to a UART
- Different configuration modes

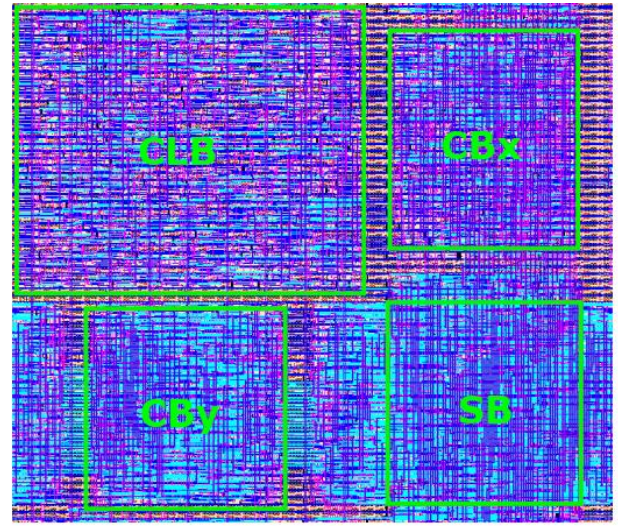
Posible advantages of ReRAM FPGAs

- Security (user circuit is encoded in resitsive states)
- Reliability (ReRAM is radiation hard)
- Probably density
- Instantanous on
- CMOS friendly

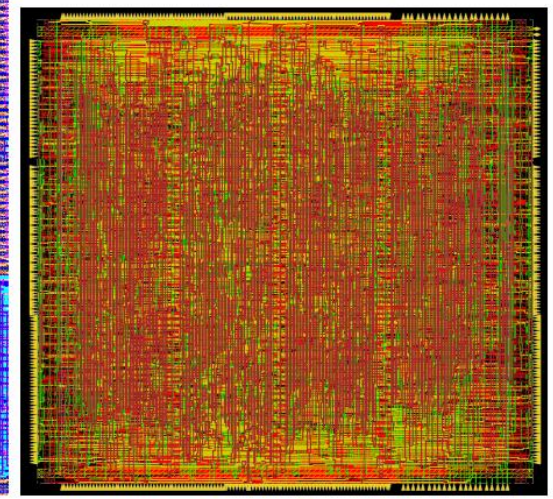


FABulous versus OpenFPGA (on Sky130)

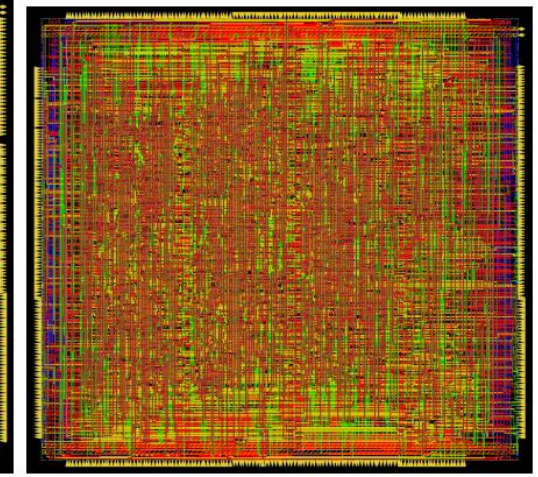




OpenFPGA Shuttle 2
250x217 um2
(+17.4%)



Fabulous Shuttle 2
220x210 um2
(+0%)



Fabulous Opt.
193x193 um2
(-21.7%)

- FABulous and OpenFPGA have a Google Shuttle2 submission
- ~ same physical impl. problem
- OpenFPGA CLBs are 17% bigger
- New optimizations gave us further 21.7% in density on the same netlist!

	resources
OpenFPGA	1300xMUX2 / 530xDFF
FABulous	376xMUX4 / 46xMUX2 / 8xFF / 586xlatch

~1200 MUX2

The FABulous eFPGA Framework – Wrap-up

- Heterogeneous (FPGA) fabric (DSBs, BRAMs, CPUs, custom blocks)
  - Multiple tiles can be combined for integrating more complex blocks
  - Custom blocks can be instantiated directly in Verilog and are integrated in Yosys, VPR/nextpnr CAD tools (Synthesis, Place&Route) (as primitive blocks)
- Support for dynamic partial reconfiguration (some elements of XC6200, like wildcard configuration)
- Configuration through shift registers or latches (or custom cells)
- Support for custom cell primitives (passtransistor multiplexers)
- Good performance / area / power figures (about 1.5x worse than Xilinx) (could be narrowed down through customization)
- Usable by FPGA users (you don‘t have to be an FPGA architect) → there are FPGA classics that we have/will clone
- ToDo: multiple clock domains, mixed-grained granularity, ...

FABulous Contributors

People:

Andrew Attwood	a.j.attwood@ljmu.ac.uk
Asma Mohsin	asma.mohsin@stud.uni-heidelberg.de
Bea Healy	tabitha.healy@student.manchester.ac.uk
Dirk Koch	dirk.koch@manchester.ac.uk
Gennadiy Knies	gennadiy.knis@stud.uni-heidelberg.de
Jakob Ternes	jakob.ternes@stud.uni-heidelberg.de
Jing Li	jing.li@manchester.ac.uk
Jonas Künstler	jonas.kuenstler@stud.uni-heidelberg.de
Kelvin Chung	king.chung@student.manchester.ac.uk
Marcel Jung	marcel.jung@stud.uni-heidelberg.de
Myrtle Shah	gatecat@ds0.me
Nguyen Dao	nguyen.dao@manchester.ac.uk

FORTE | Forming the future of nanoelectronics

EPSRC Engineering and Physical Sciences Research Council

Carl Zeiss Stiftung

GEFÖRDERT VOM

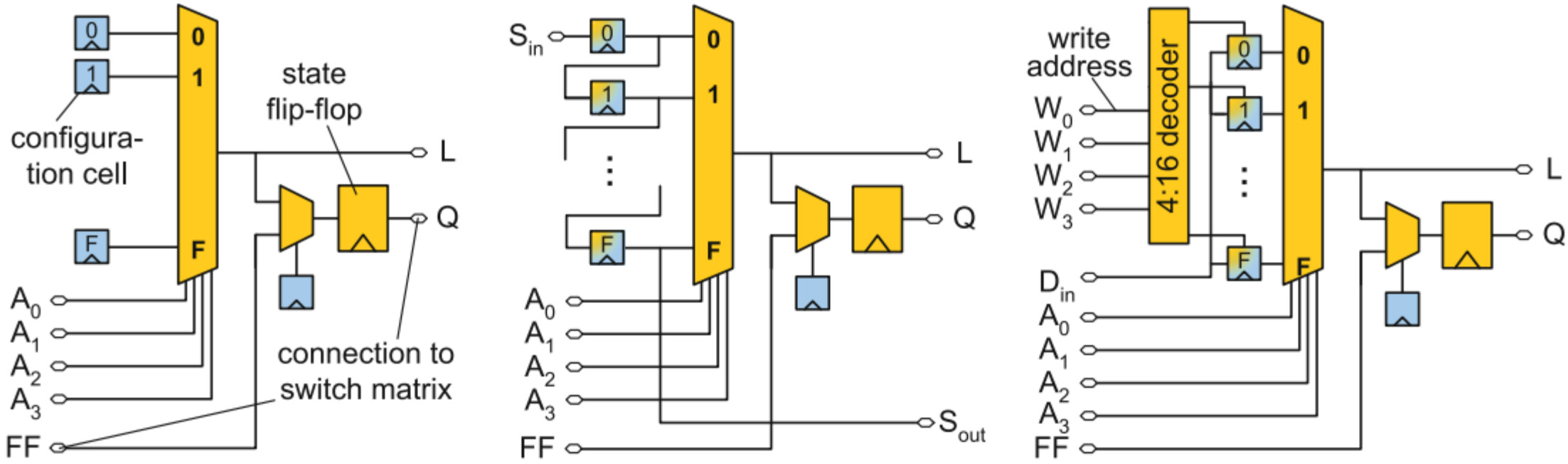
Bundesministerium für Bildung und Forschung

See our projects under: **https://github.com/FPGA-Research-Manchester**

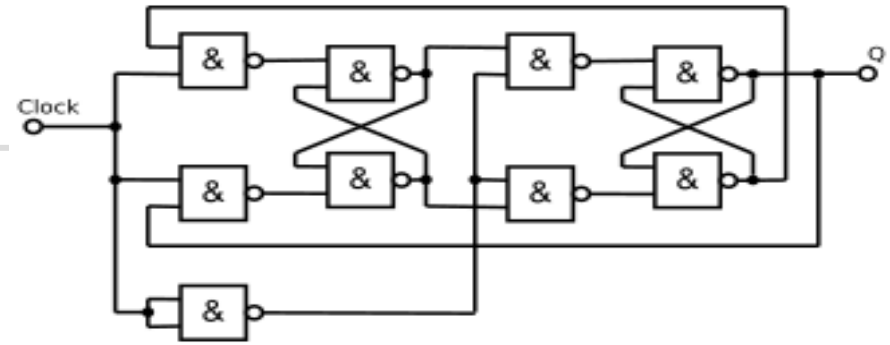
This work is kindly supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/R024642/1 and Carl-Zeiss-Stiftung

FPGA Basics – Logic

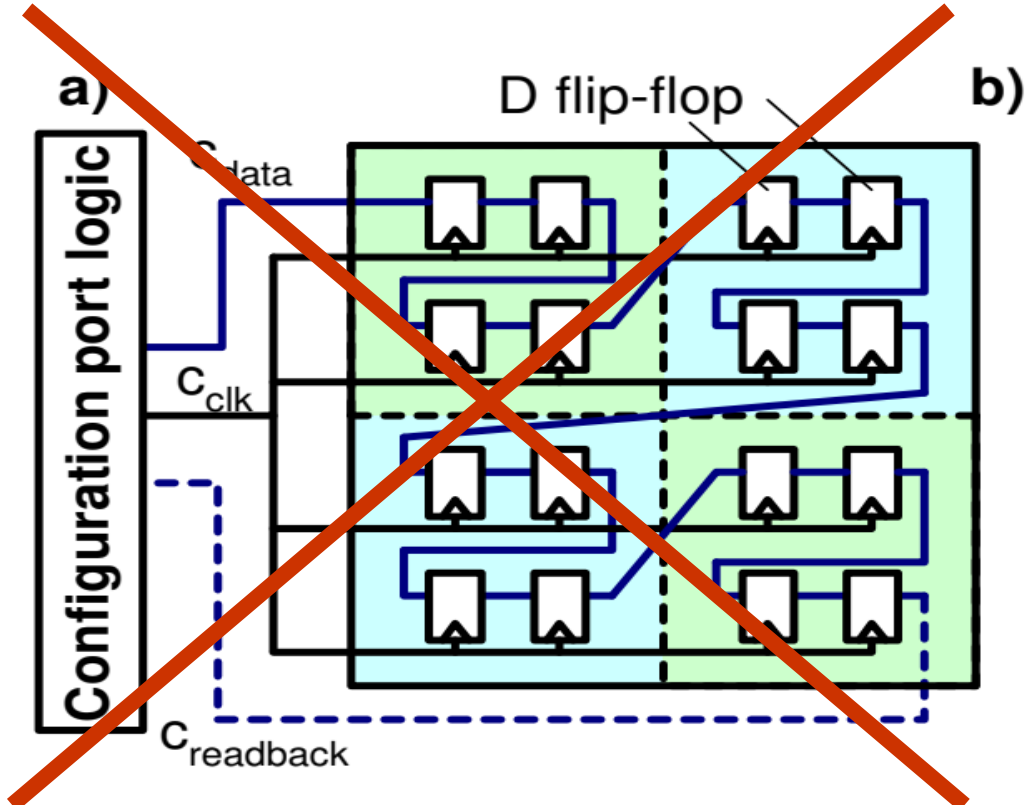
- Look-up tables (LUTs) are basically multiplexers selecting configuration latches storing a function as a simple truth table
- Configuration latches are usually written through the configuration port only
- In distributed memory options (LUT is used as a shift register or memory file, table is also writeable through the user logic)

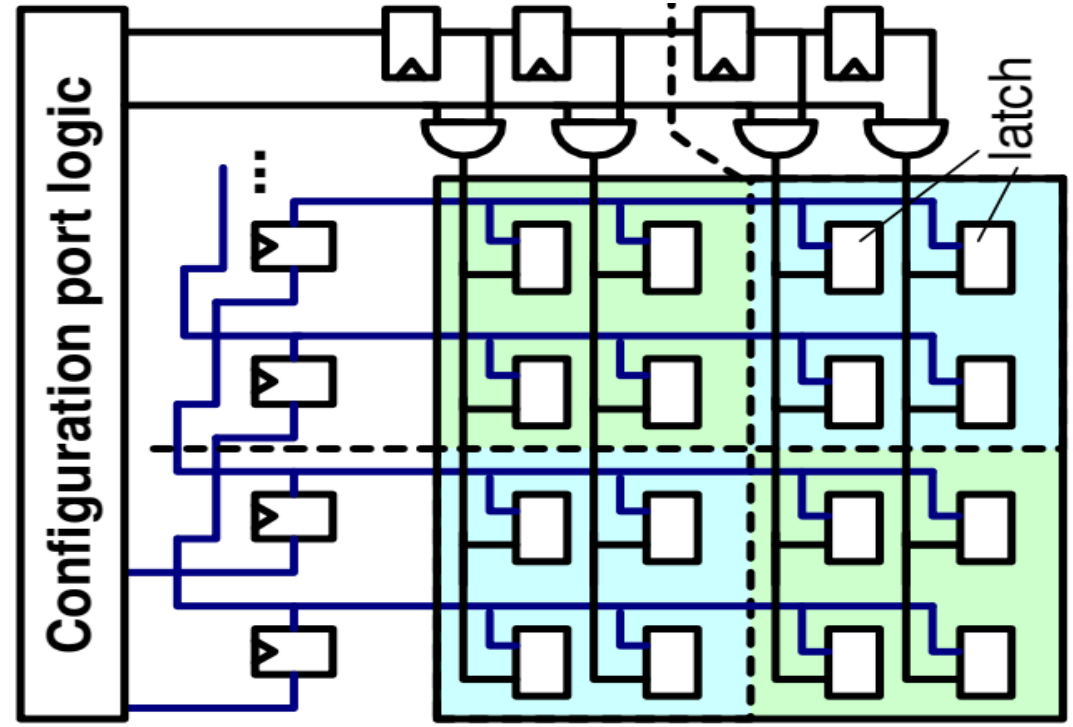


FPGA Configuration

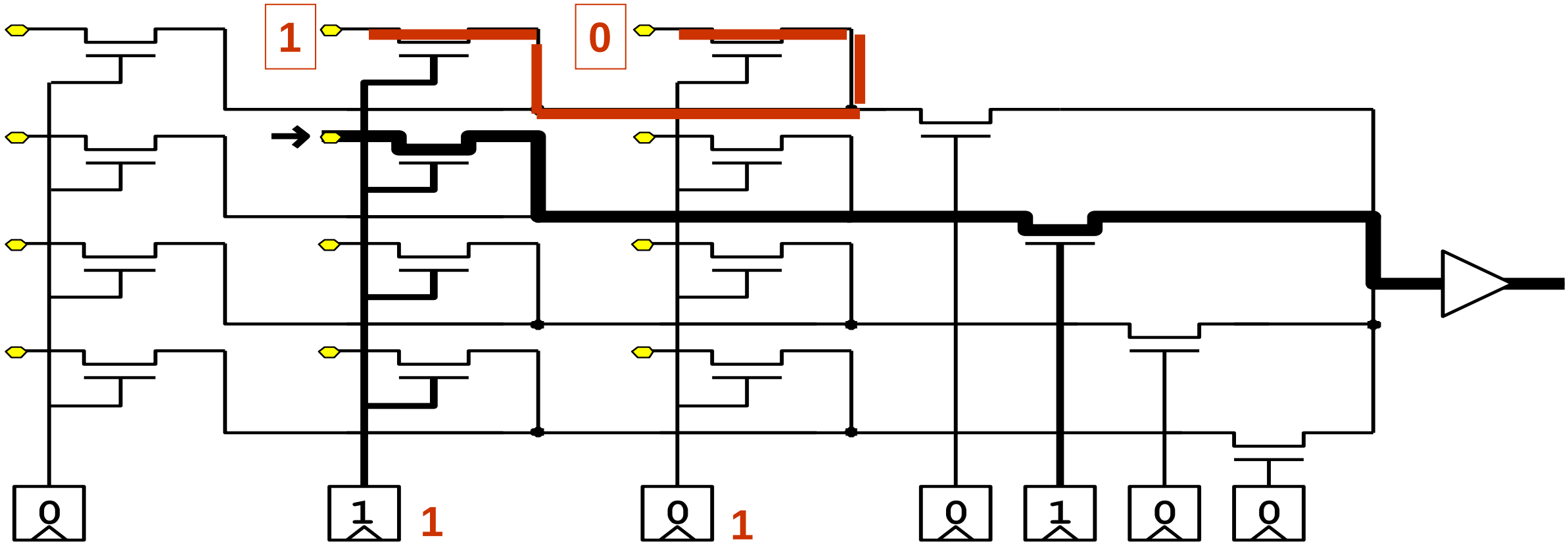


- Do not use shift register configuration
  - High power during configuration (thousands of bits)
  - Configuration only valid if completely shifted in (transient short-circuits or ring-oscillators)
  - Cannot do „real“ partial reconfiguration (static routes through reconfigurable regions)
  - Too expensive (shift registers need flip flops, frame-based configuration can do with latches)





FPGA Basics – Routing (Virtex-II style)



- **AMD/Intel use multiple levels of one-hot encoded routing with pass-transistors**
- **Multiple activated inputs can cause short-circuit situations**
  → this is why you should blank a region before overwriting it with a new module
  → less of a problem for encoded bitstreams (not one-hot encoded)

Tile-based Design in FABulous

- Replace standard cell multiplexers with custom mux-4

$A_{std\text{-}cell} - A_{c\text{-}mux4} \times N = (33.8\,\mu m^2 - 17.5\,\mu m^2) \times 376 = 6{,}116\,\mu m^2$

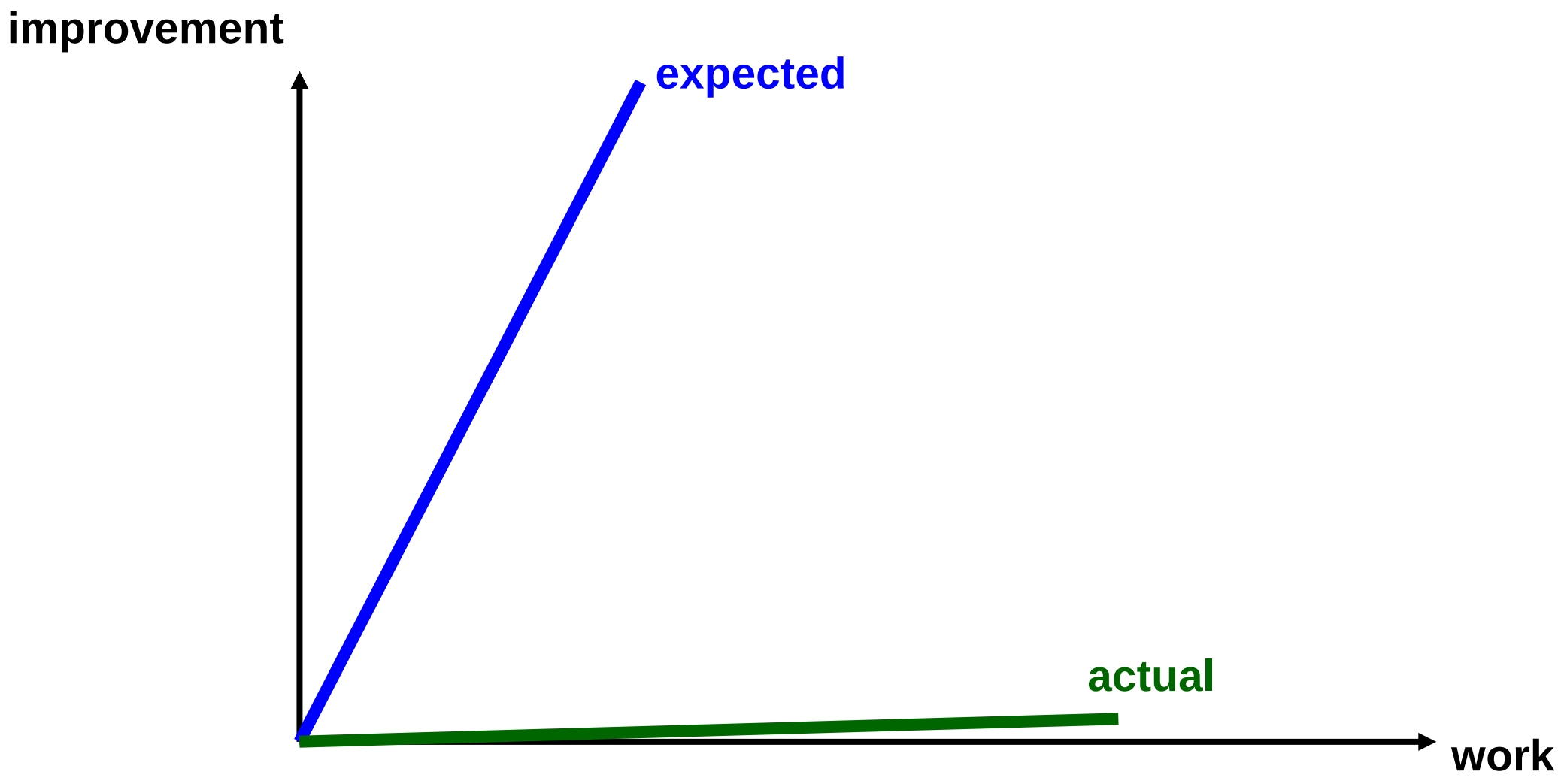
	Standard cell				Custom mux-4	
	height	width	**area**	util.	**area**	util.
CLB	219 µm	219 µm	**47,961**	81.8%	**46,225**	60.7%
REG	219 µm	214 µm	**46,866**	84.1%	**46,655**	64.3%
DSP	443 µm	185 µm	**81,955**	80.9%	**81,780**	56.7%

Observation:

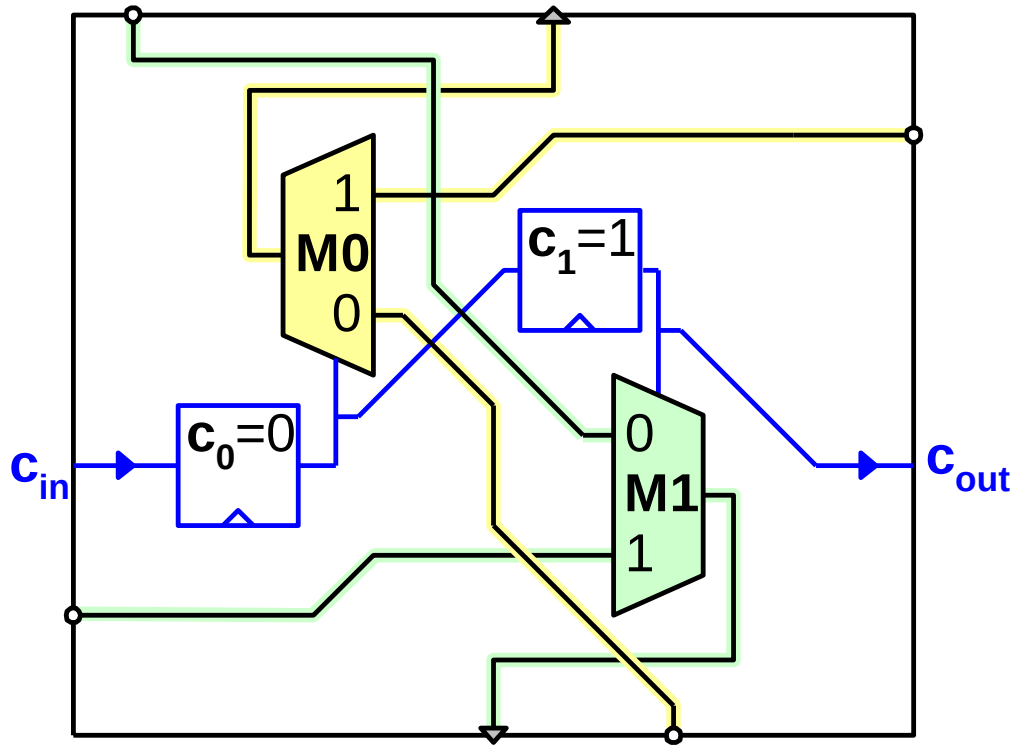
- No area improvement
- Instead: core utilization went down

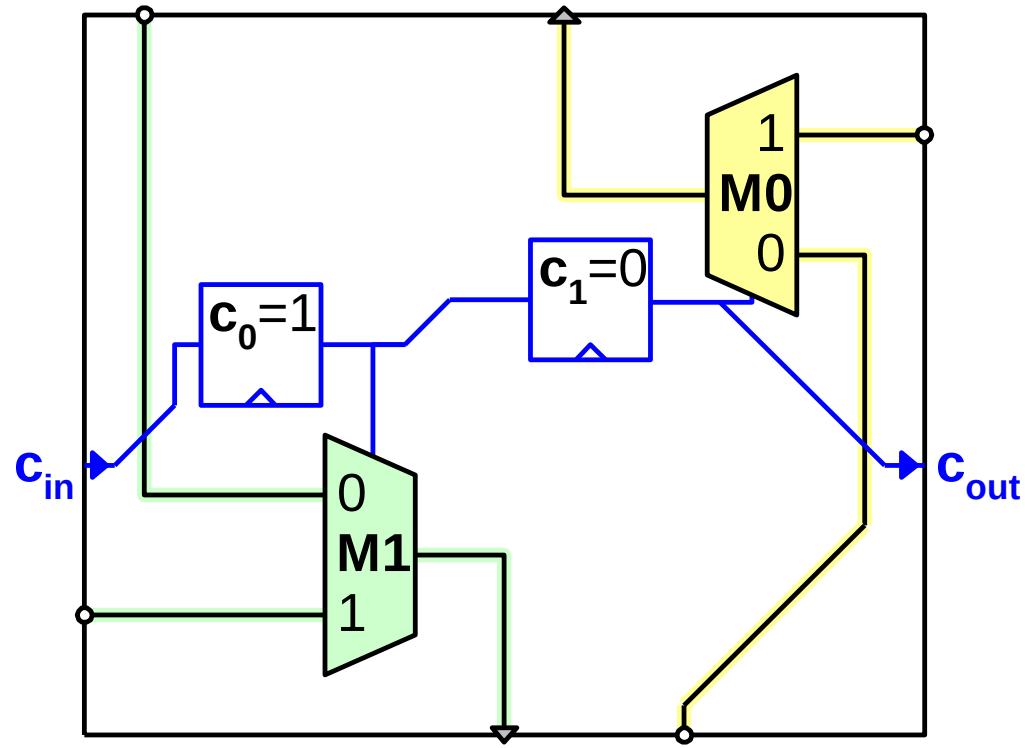
→ Congested tile routing

In short



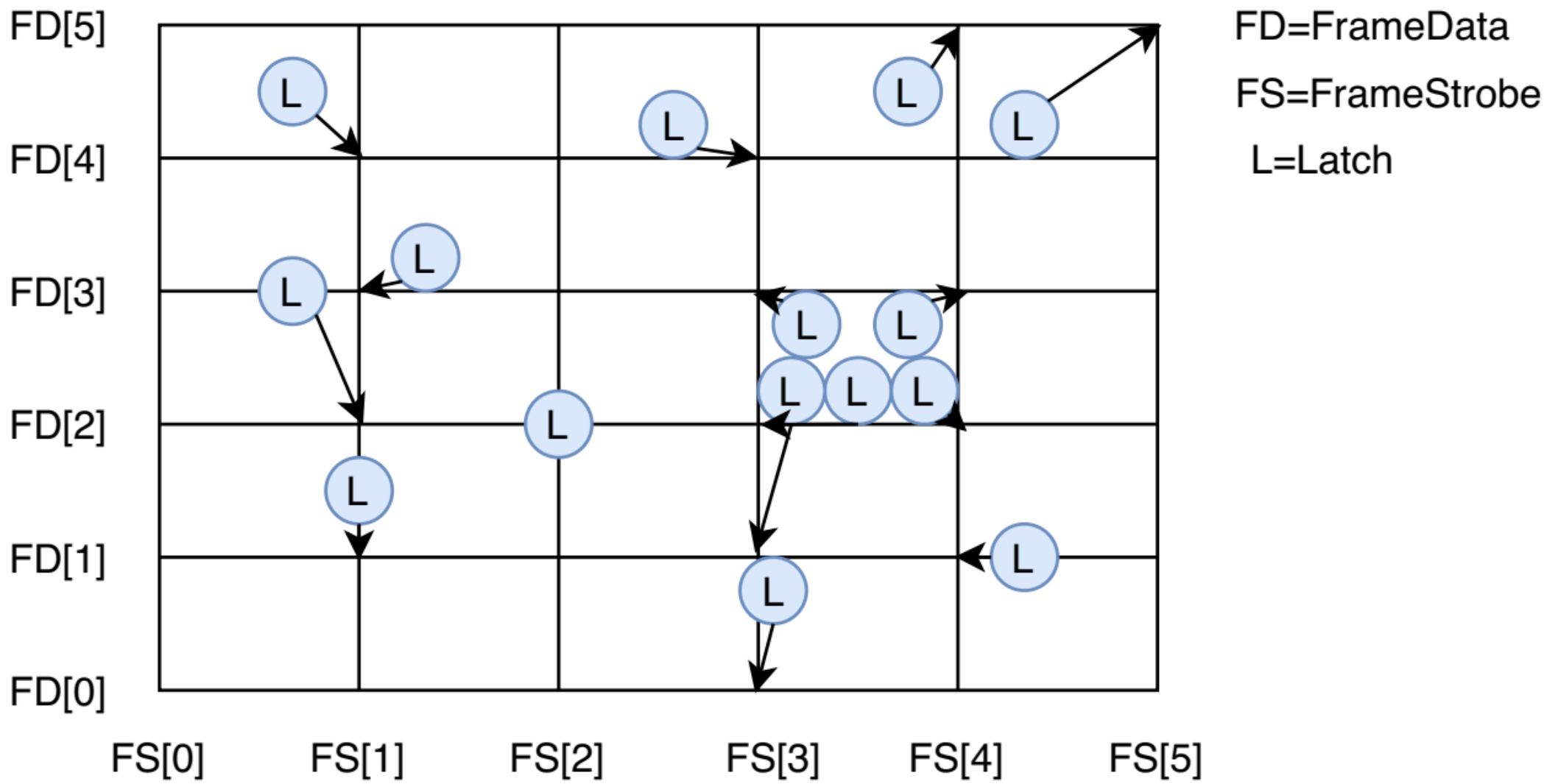
Optimization: Bitstream Remapping





- The configuration bit cells may induce inferior placement of multiplexers
- We can remap configuration bits → requires remapping of the bitstream (trivial)

Optimization: Bitstream Remapping



- We use Google's Operations Research tools to compute the grid points (https://github.com/google/or-tools)

Optimization: Bitstream Remapping