Ole Richter, Luis Zungia and Rajit Manohar
2026 Free Silicon Conference (FSiC) 6-8 July 2026

A more modern way to design (open source) (memory) macro compilers

Disclaimer: Work in progress

- It is work in progress
- Not a funded project – limited resources/time

- The source code is GPLv2+
- Public availability will be after a successful tapeout
- It will be part of the Asynchronous Circuit Toolkit (github.com/asyncvlsi/actflow)

Why are (memory) macros important for you

- Memory is in (almost) any chip
- Memory density, energy and performance is key

- Current open memory compiles do not use the same automation as digital logic
- With reasons

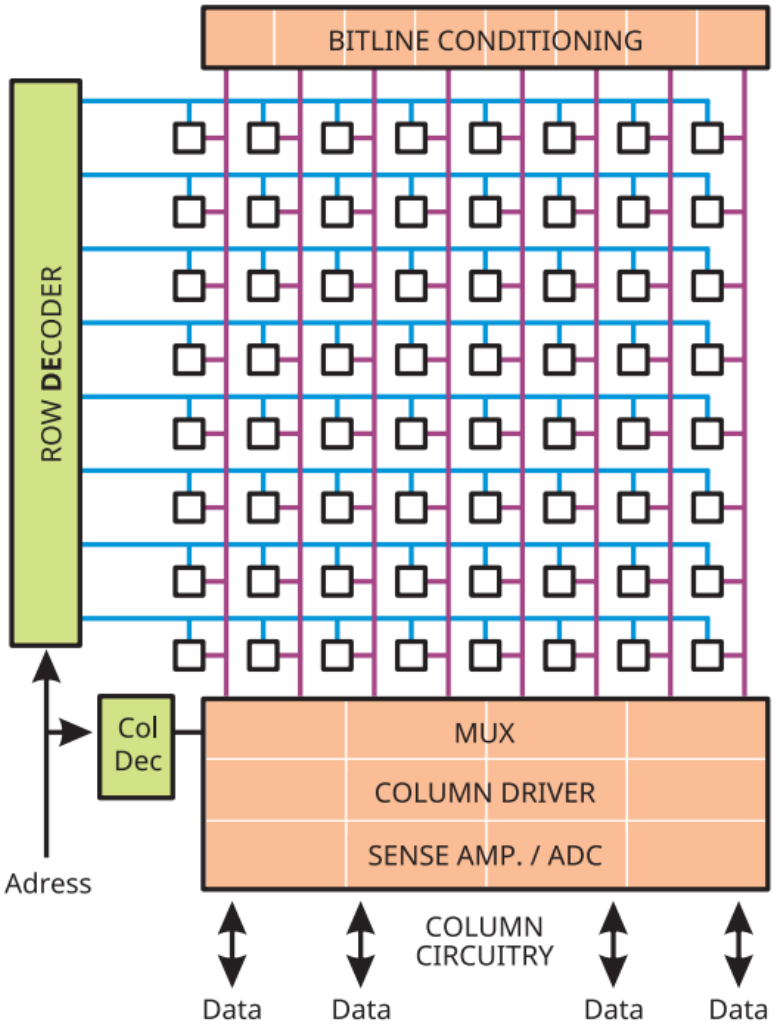
- What is missing is to bring more automation in the to macro compilers?

(hard) Macros

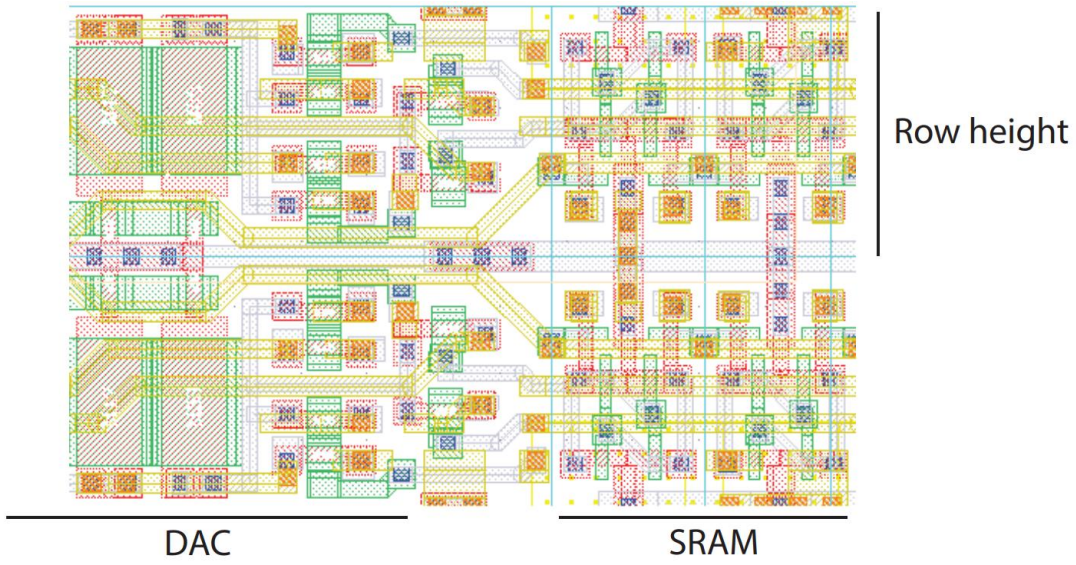
- Don’t participate in P&R beyond being a black box with pins
- Mostly memories, often mixed-signal
- Often repeating patterns, tilable -> macro compiler

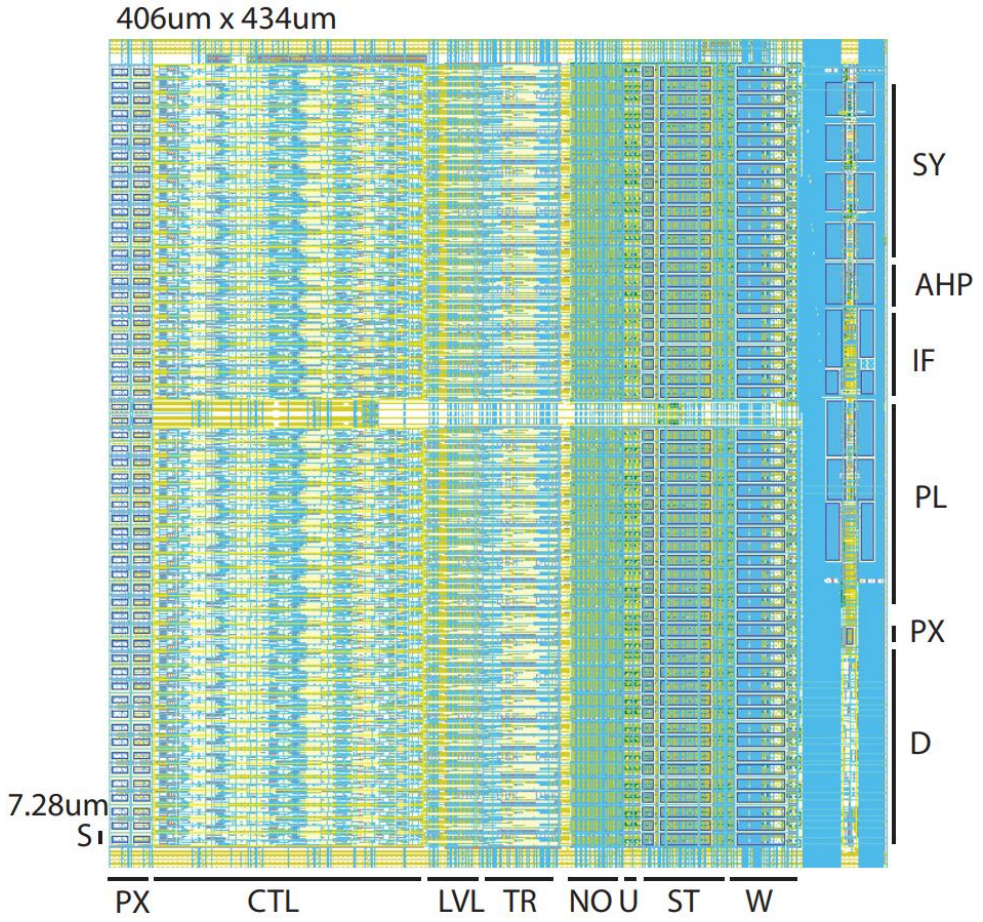
Memory

- SRAM
  - Cache, Register Files
- CAM
- DRAM
- FLASH/EEPROM
- MRAM
- ReRAM



Compute-in-memory, (analog) compute arrays, emerging device memories, eFPGAs





Why are memory / macro compilers so hard

To build, port and to extend?

- Its not just building a new cell library
- It is not just building a fixed size custom memory bock

You need to guarantee functionality for every generation -> determinism and/or verification

Why are memory / macro compilers so hard?

Solutions used so far:

- Hardcoded, magic number scripting of placement and routing -> determinism
- Alt. run multi corner extracted sims or validation on every run
- Scripted control logic – front and backend

Our aim: if you are designing a full custom (mixed-signal) macro: we want to give you the tools to build a compiler out of it, with little extra overhead and much more automation.

But why is memory so different from logic?

Memory Macros are mixed-signal design

- To be compact and efficient:
  - The layout / technology defines everything

- To be performant
  - Memories have to be self timed (asynchronous)

⇒ To get competitive designs, requires being layout driven

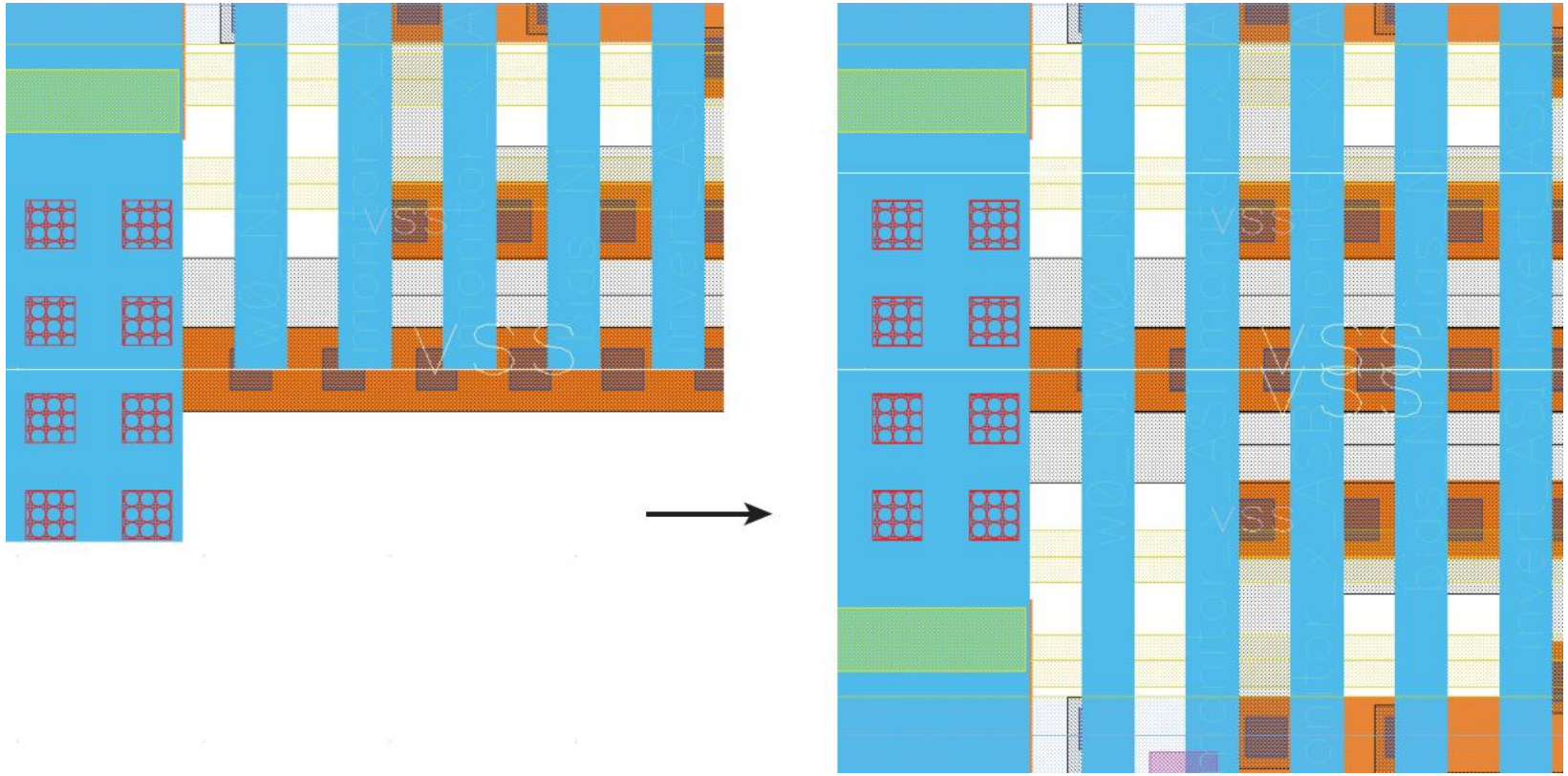
⇒ every design is different enough to prohibit a top-down design

Two parts: tileable macro and logic

Building the puzzle - abutment

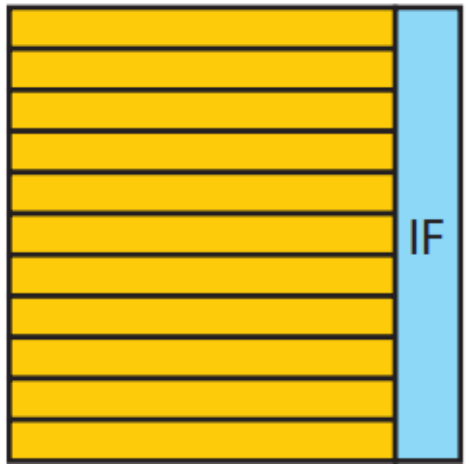
- XY-flip
- Connect by abutment

- Designer define: markers and P&R boundaries



Tiling squares

- Rectangles
- Connecting rectangles to create new rectangles
- Eg. SRAM cell to array
- Eg. 2 SRAM cell columns to mux cell



tileing directives

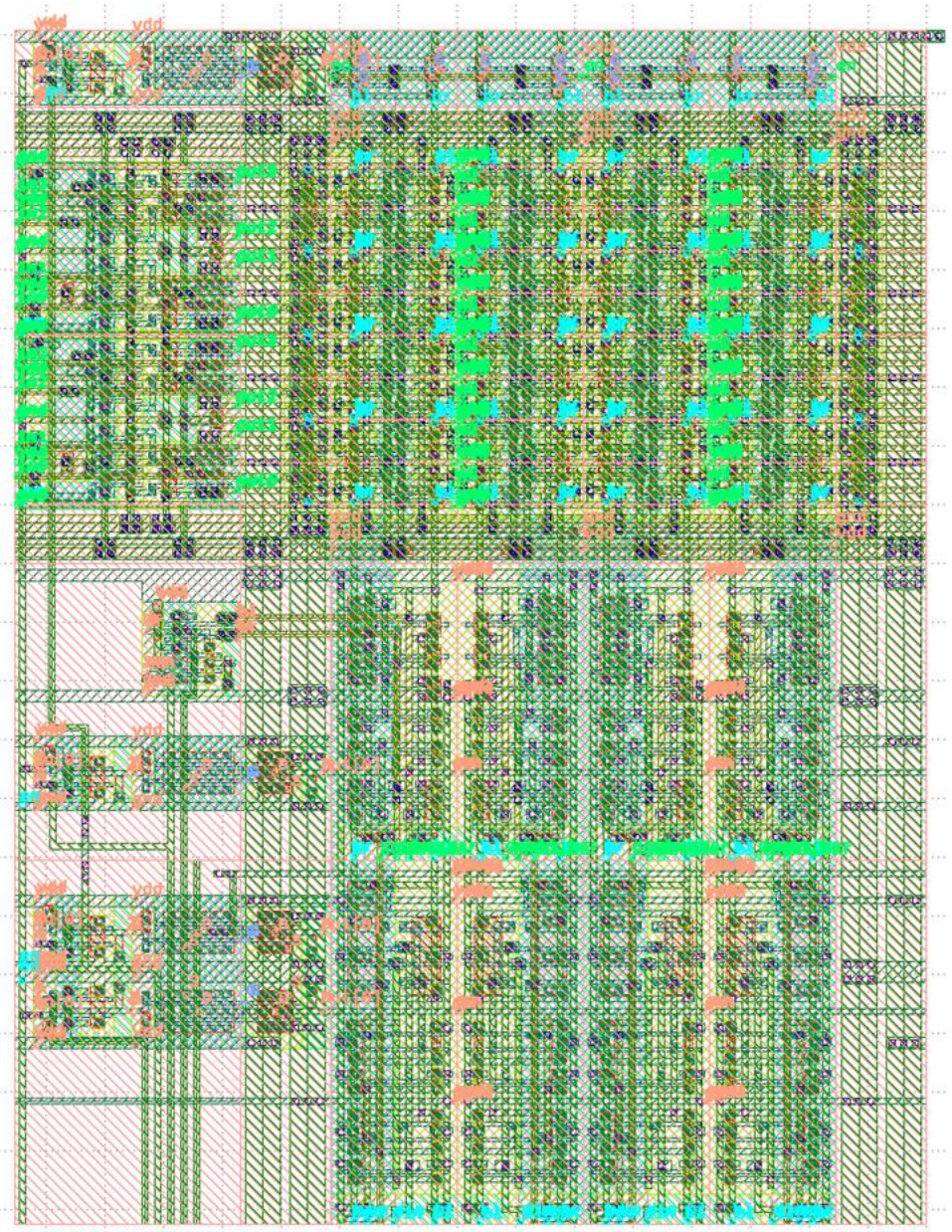
```
abut <left, top, right, bottom> { [flip] cell1, [flip] cell2, [cell3 …]}
```

When abutting a cell the port or port list needs to match for both cells

A cell can also be a nested abut statement cellX = abut ...

Tileing directives

```
defproc array (bool wl[8], sl[4], slb[4]){
        sram6t cell[8][4];
        extern (<tileinglang>) {
                abut top { (,i:4:
                        flip abut right { (,j:4: cell[2*i][j]) },
                        abut right { (,j:4: cell[2*i+1][j]) }    )
                }
        }
}
```



Tileing macro example
4x8 SRAM

Custom macros with digital P&R components

"tileble macro" and "logic"

- We need to integrate those blocks automatically
- We can't assume any top down architecture => otherwise result is not competitive
- We need to verify timing / functionality

=> build the tool chain parts

WIP – Mix and match P&R and macros

Automate a lot of connections and placement that are hard to script decoder / controller

What does this translate to

- Net tracing inside the macro -> automatic pin naming
- Pin location extraction for Place and Route
- Timing driven placement / Timing driven global routing
- Timing analysis for relative timing constraints

WIP – automated controller timing closure

- Custom self timing by adapted delay lines
- Initially based on a liberty file generated by sim with (pex) for the macro part (wip - xcell)
- Later STA timing and slope propagation could be extended to some types of macros like SRAM or eFPGA
- Emit a liberty file for the full design

So why build it with ACT

- Timing verification on relative timing - cyclone
- STA corner verification (wip) - cyclone
- Automatic delay line insertion (wip) - interact
- Mixed mode simulation (behavioural – gate level - xyce) - actsim
- Export spice (with or without layout parasitic) - interact
- Export Verilog netlists - interact
- Hierarchical mix and match between macro and P&R (wip) – Dali, PhyDB, layout

Note: Act only includes tools that are needed for async circuits, it does not duplicate any tools already available and sufficient for async design, it uses them.

Curious? – reach out to us!

Competitive layout first macro compilers, that lets you modify and automate what makes sense

Ole Richter – ojuri@dtu.dk
Luis Zuniga
Rajit Manohar – rajit.manohar@yale.edu

Build on prior work from:

- Samira Ataei
- Thomas Jagielski
- Matt Dobre


DTU