Fine-grained hierarchical placement constraining for timing closure (and more)

Álvaro Navarro-Tobar

Ciemat

Plan de Recuperación, Transformación y Resiliencia

Funded by the European Union
NextGenerationEU

Proyecto PID2020-116262RB-C42 financiado por:

MINISTERIO DE CIENCIA E INNOVACIÓN

AGENCIA ESTATAL DE INVESTIGACIÓN

Presentation and outline

Ciemat has been part of CMS since its construction (myself for “just” 15 years) doing, among many other things, FPGA stuff

- Drift Tubes construction: readout frontend and backend (Altera, Spartan-IIe)
- Phase1 upgrade: readout backend (Spartan 6, Virtex 7)
- Phase2 upgrade: detector frontend (Polarfire) and trigger backend (Virtex UltraScale+)

Outline

CMS Muon Trigger Primitive generation Phase-2 upgrade

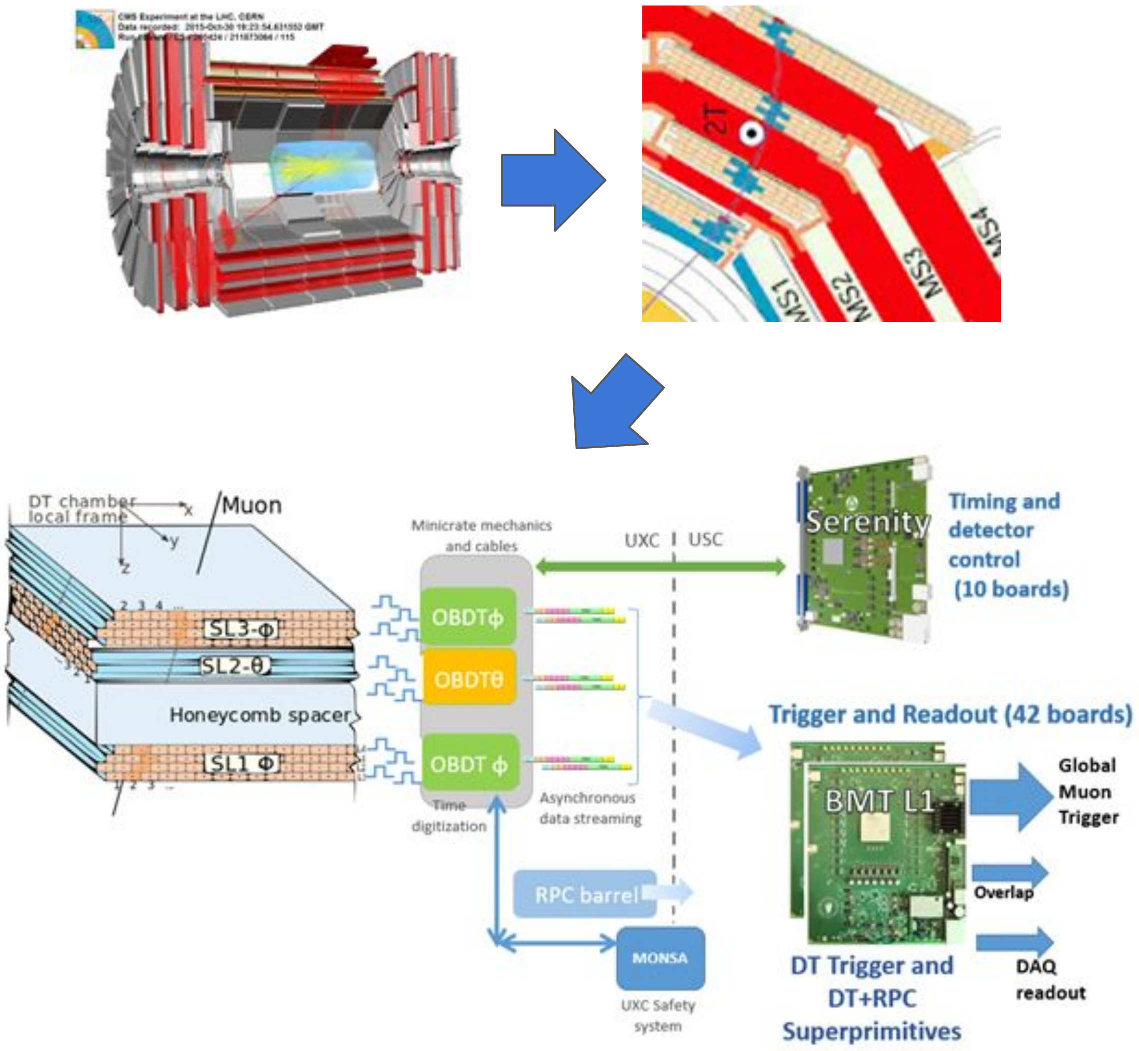
•**Analytical Method** (proposed by CIEMAT) has been developed to implement reconstruction of the barrel (DT+RPC) trigger primitives in HL-LHC

•Drift Tube chambers hits received asynchronously and need to be combined on each Superlayer to form a track.

•Laterality uncertainty and time drift uncertainty (400 ns)

•**Exploits maximum achievable resolution**, bringing the hw system closer to **offline performance capabilities**. [10.1016/j.nima.2023.168103]

•In collaboration with UAM and Univ. Oviedo. Uni Ovi also participates in OMTF



Need for speed

Throughput, *throughput*, **throughput**: our algorithm suffers from combinatorial explosion. Squeezing the last %'s of efficiency requires increasing substantially the number of hit combinations that can be analyzed in a fixed latency

Due to system constraints (experiment, hardware, firmware framework), the most "natural" frequencies to use are 240, 360 and 480 MHz (6x, 9x and 12x LHCclk)

480 MHz seemed like a good idea, challenging but achievable. Target 2.079 ns

Not so fast

The DT AM algorithm can be split in ~15 relatively small sub-units (entities)

My bottom-up approach

1. Design each entity so that it is able to run **very fast**, close timing with a big margin (1.7-1.8 ns) → challenging, 100 ps at this scale is a big achievement
    - out of context runs
    - 1.7 ns = 588 MHz, global clock buffers and BRAM and DSPs performance in the range 650-750 MHz
2. Integrate. With proper piping between modules, the margin on first step **should be enough** to absorb the difficulty to put the whole design together
3. **Fail**: I ended up thanking closure at 360 MHz

The Usual Suspects

Problems identified in my design and workflow

- SLR boundary crossings
- Not enough piping between modules (e.g. some modules output directly from unregistered output RAM)
- High fanout control signals not sufficiently piped (reset, BX, enables…). Sometimes not so easy to detect, as the violation may not appear in the high-fanout signal but in the entity's inner RTL which is being pulled too tightly.

Placement helps identifying/fixing

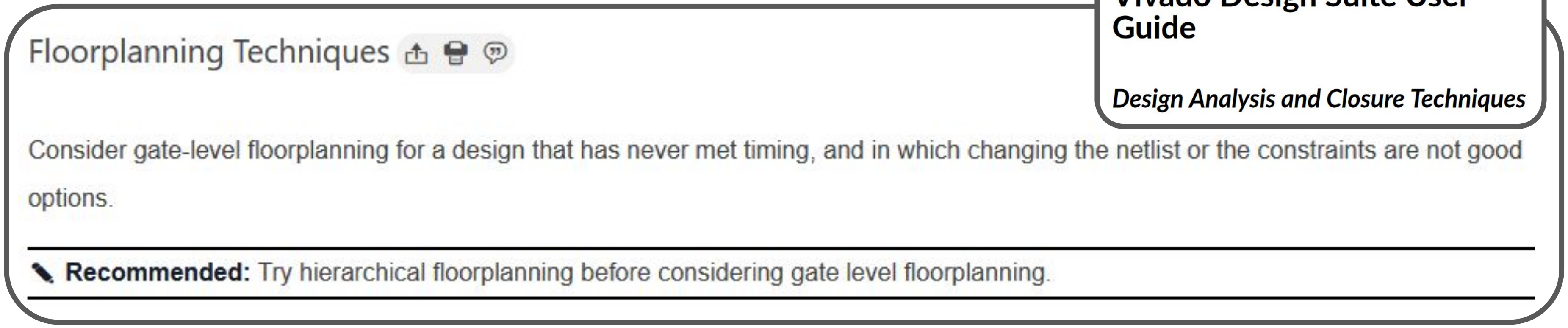
- Vivado doesn't make a good enough job in placing big design (500k LUTs/FFs) with many paths close to critical timing in big area; complexity is too high. Rolling dice a million times almost guarantees you get a bunch of violations. They don't appear consistently in the same paths, or the same instances of the same entities.

Placement makes a big difference

Approaching the placement problem

Vivado Design Suite User Guide

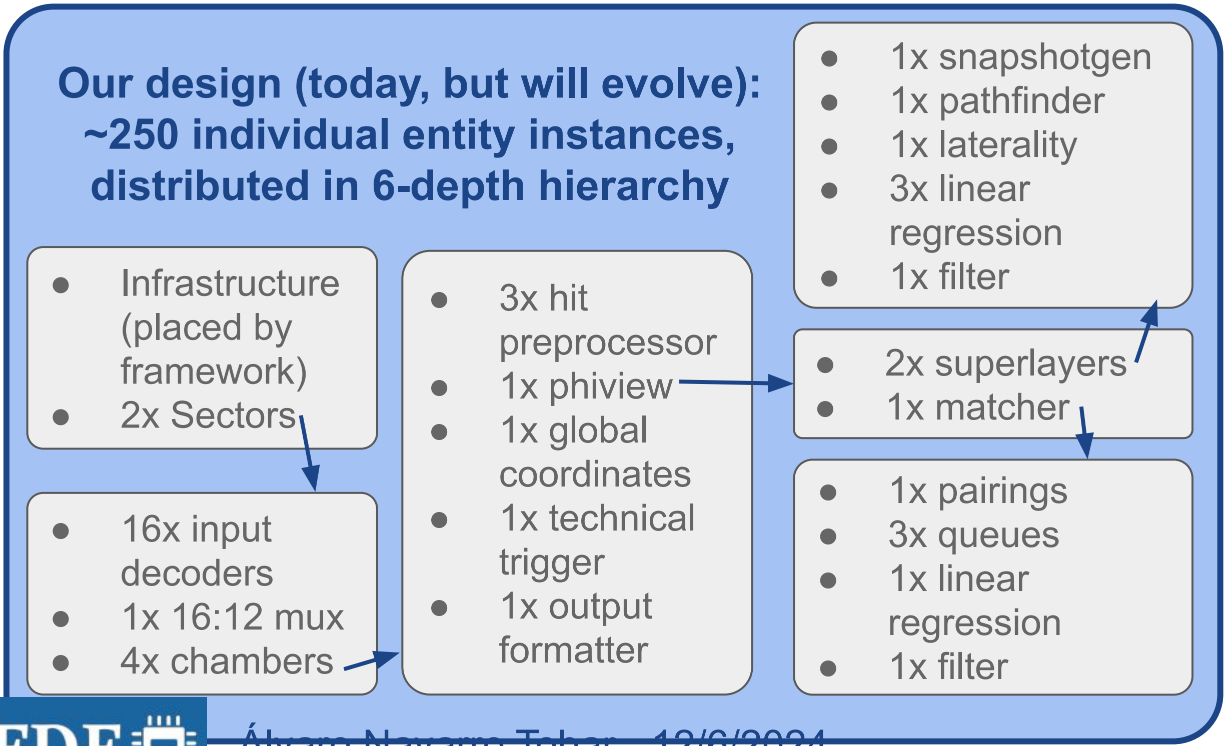
Design Analysis and Closure Techniques



Floorplanning Techniques

Consider gate-level floorplanning for a design that has never met timing, and in which changing the netlist or the constraints are not good options.

Recommended: Try hierarchical floorplanning before considering gate level floorplanning.



Our design (today, but will evolve): ~250 individual entity instances, distributed in 6-depth hierarchy

- Infrastructure (placed by framework)
- 2x Sectors

- 16x input decoders
- 1x 16:12 mux
- 4x chambers

- 3x hit preprocessor
- 1x phiview
- 1x global coordinates
- 1x technical trigger
- 1x output formatter

- 2x superlayers
- 1x matcher

- 1x snapshotgen
- 1x pathfinder
- 1x laterality
- 3x linear regression
- 1x filter

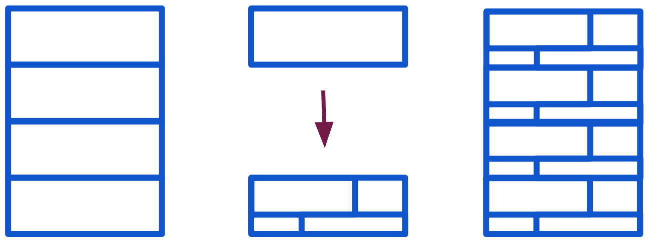
- 1x pairings
- 3x queues
- 1x linear regression
- 1x filter

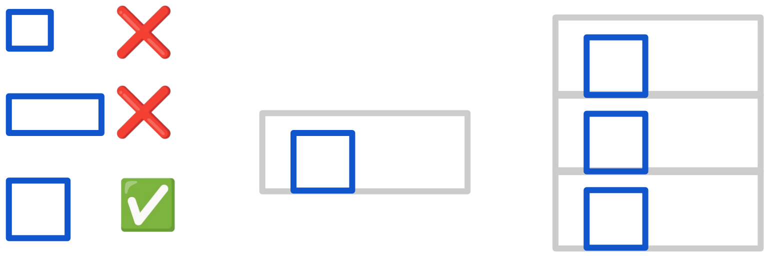
Device AMD VU13P (4 SLRs) ⇒ try 2 chambers/SLR (2nd-level depth) → **Fail!!** → need finer grained placement, but…

- How deep will I end up needing? All the way down?
- How would my constraints file look like, iterating over repeated blocks with 6-levels-deep inner structure?
- How maintainable will it be when the algorithm evolves?
- How easy is it to quickly draft and test pblocks for small instances?

I wish I had a placement helper framework which…

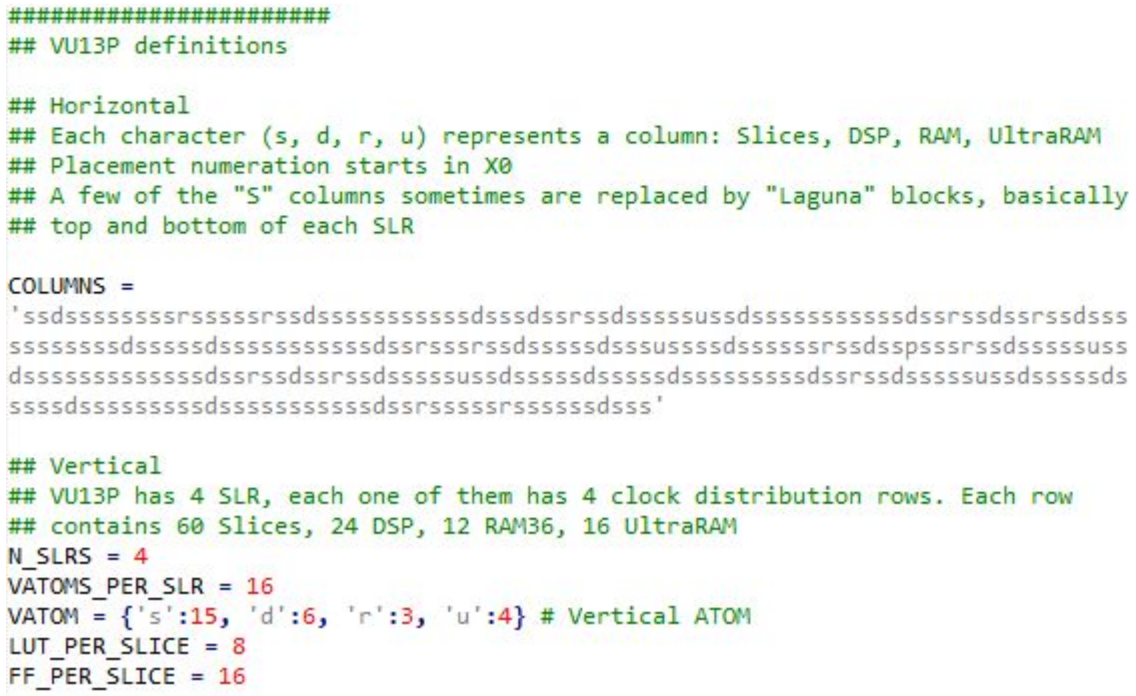
- Is **user-friendly**, backend does the heavy lifting
- Is **aware of resources and utilization**, automatically resize pblocks if needed, and reports on utilization with configurable safety margins
- Allows for **top-down workflow**: start with coarse placement, when it doesn't work, just subdivide a pblock to have inner structure, and result is offset to be placed where the original pblock was instantiated
- Allows for **bottom-up workflow**: when need arises to repeatedly test smaller, critical portions of design, it can generate constraints only for it, then run OOC, then progressively integrate on bigger portions of the design
- Reasonably easy to evolve with the design, **maintainable**, not a pain to remake everything when one of the modules in the middle of the algorithm is redesigned and grows or shrinks





Backend: X-Y coordinates and FPGA definition

Work in progress



```
########################
## VU13P definitions

## Horizontal
## Each character (s, d, r, u) represents a column: Slices, DSP, RAM, UltraRAM
## Placement numeration starts in X0
## A few of the "S" columns sometimes are replaced by "Laguna" blocks, basically
## top and bottom of each SLR

COLUMNS =
'ssdssssssssrsssssrssdsssssssssssdsssdssrssdsssssussdsssssssssssdssrssdssrssdsss
ssssssssdsssssdsssssssssssdssrsssrssdsssssdsssussssdssssssrssdsspsssrssdsssssuss
dsssssssssssssdssrssdssrssdsssssussdsssssdsssssdsssssssssdssrssdsssssussdsssssds
ssssdsssssssssdsssssssssssdssrssssssrssssssdsss'

## Vertical
## VU13P has 4 SLR, each one of them has 4 clock distribution rows. Each row
## contains 60 Slices, 24 DSP, 12 RAM36, 16 UltraRAM
N_SLRS = 4
VATOMS_PER_SLR = 16
VATOM = {'s':15, 'd':6, 'r':3, 'u':4} # Vertical ATOM
LUT_PER_SLICE = 8
FF_PER_SLICE = 16
```

Horizontally, each character in the COLUMNS string represents a fabric column

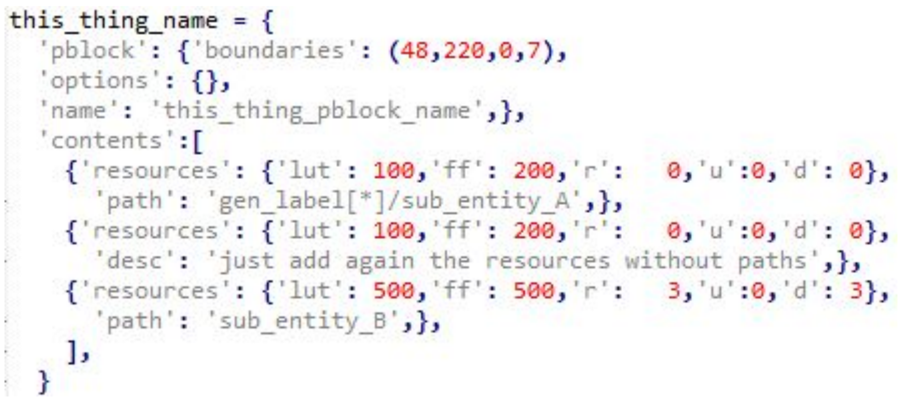
- Slices, RAM, DSP
- Others are still missing (PCIE, CFGIO…), they affect routability
- Non uniform distribution, pblocks that use RAM or DSPs cannot really be offset horizontally
- Slice-only pblock with care

Vertically, divided in “vertical atoms”, the minimum vertical step without fractional number of RAM and/or DSP

- Then stacked up to form SLR, then the full FPGA
- Turns out to be too coarse, favors generation of high-aspect-ratio, vertical pblocks, which is bad for routing.

User interface: pblocks and boxes
(bonus: top-down workflow example)

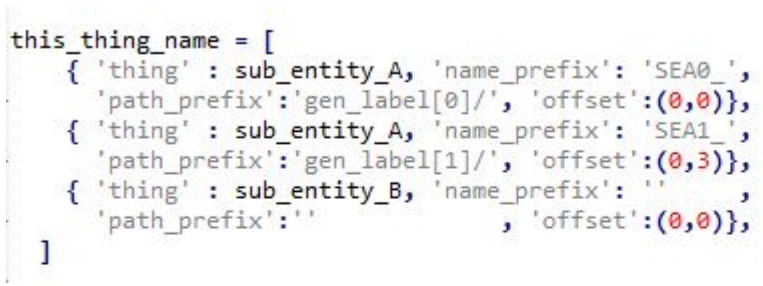
Work in progress



```
this_thing_name = {
  'pblock': {'boundaries': (48,220,0,7),
  'options': {},
  'name': 'this_thing_pblock_name',},
  'contents':[
    {'resources': {'lut': 100,'ff': 200,'r':   0,'u':0,'d': 0},
      'path': 'gen_label[*]/sub_entity_A',},
    {'resources': {'lut': 100,'ff': 200,'r':   0,'u':0,'d': 0},
      'desc': 'just add again the resources without paths',},
    {'resources': {'lut': 500,'ff': 500,'r':   3,'u':0,'d': 3},
      'path': 'sub_entity_B',},
    ],
  }
```

pblock

- Define boundaries, if one edge left undefined, auto-size
- Options: IS_SOFT, CONTAIN_ROUTING, other features..
- Contents, each item:
  - Defines resource utilization
  - Defines paths (if path field present, otherwise just add the resources)



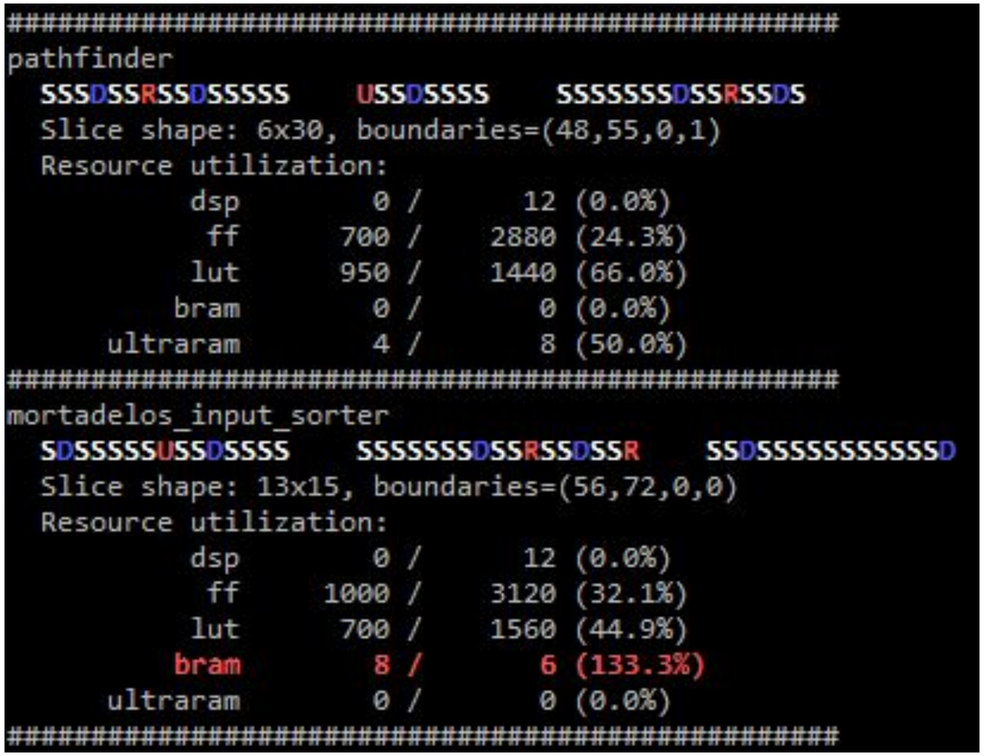
```
this_thing_name = [
    { 'thing' : sub_entity_A, 'name_prefix': 'SEA0_',
      'path_prefix':'gen_label[0]/', 'offset':(0,0)},
    { 'thing' : sub_entity_A, 'name_prefix': 'SEA1_',
      'path_prefix':'gen_label[1]/', 'offset':(0,3)},
    { 'thing' : sub_entity_B, 'name_prefix': ''     ,
      'path_prefix':''             , 'offset':(0,0)},
  ]
```

box

- Organizer, contains a list of “things” (pblocks or other boxes)
- 3 alterations recursively applied down the hierarchy for each thing:
  - Prefix the name of inner pblocks
  - Prefix each path in all inner pblocks
  - Offset the fabric placement of each inner pblock

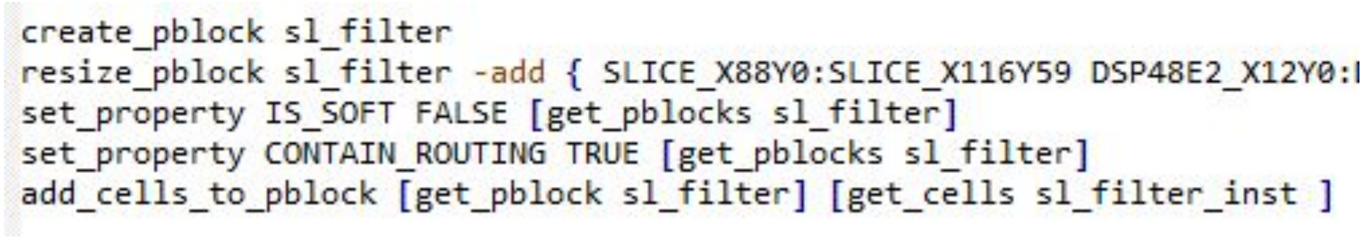
pblock output on console and tcl

- Shows columns for current pblock and neighboring towards left and right
- Shows slice shape and boundaries
- Shows used vs available resources
  - Warns when over-utilized or above safety margins
- Outputs pblocks to constraints file



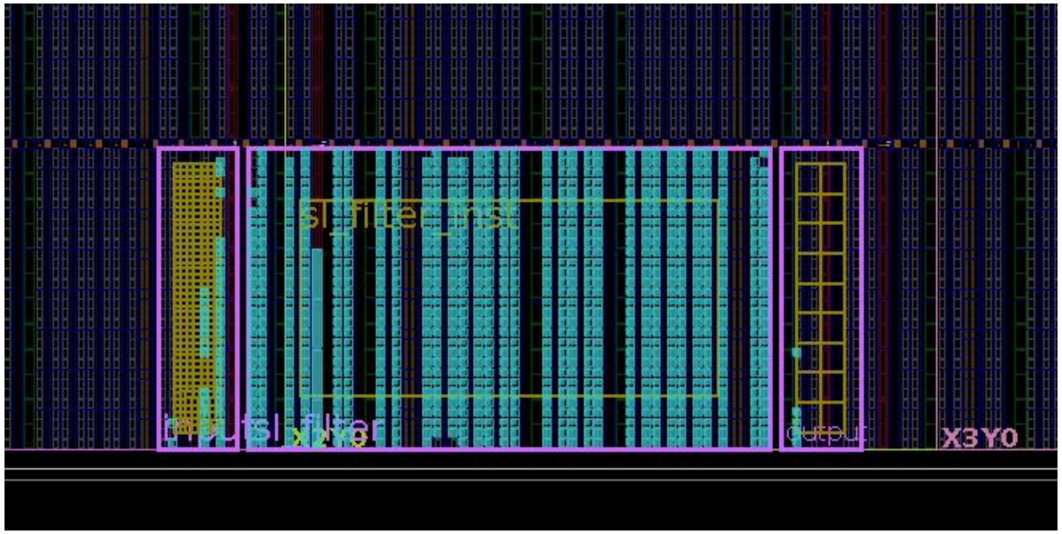
```
##############################################################
pathfinder
  SSSDSSRSSDSSSSS    USSDSSSS    SSSSSSSDSSRSSDS
  Slice shape: 6x30, boundaries=(48,55,0,1)
  Resource utilization:
           dsp        0 /       12 (0.0%)
            ff      700 /     2880 (24.3%)
           lut      950 /     1440 (66.0%)
          bram        0 /        0 (0.0%)
      ultraram        4 /        8 (50.0%)
##############################################################
mortadelos_input_sorter
  SDSSSSSUSSDSSSS    SSSSSSSDSSRSSDSSR    SSDSSSSSSSSSSSD
  Slice shape: 13x15, boundaries=(56,72,0,0)
  Resource utilization:
           dsp        0 /       12 (0.0%)
            ff     1000 /     3120 (32.1%)
           lut      700 /     1560 (44.9%)
          bram        8 /        6 (133.3%)
      ultraram        0 /        0 (0.0%)
##############################################################
```

Work in progress

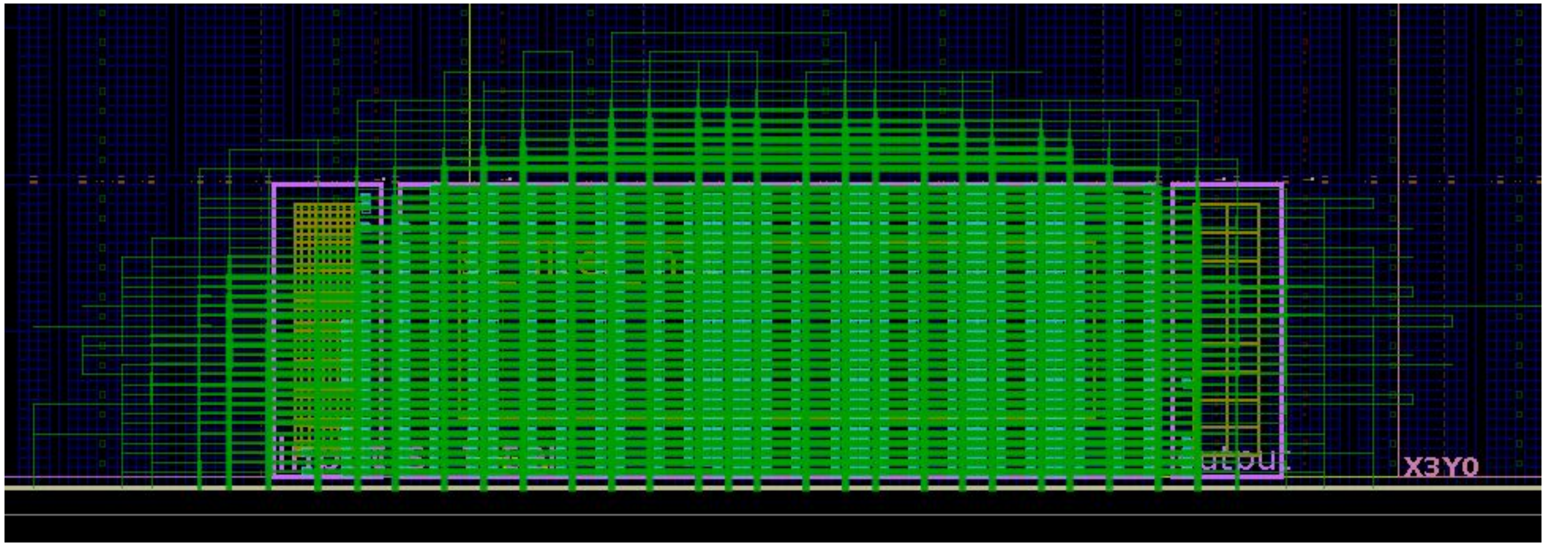


```
create_pblock sl_filter
resize_pblock sl_filter -add { SLICE_X88Y0:SLICE_X116Y59 DSP48E2_X12Y0:
set_property IS_SOFT FALSE [get_pblocks sl_filter]
set_property CONTAIN_ROUTING TRUE [get_pblocks sl_filter]
add_cells_to_pblock [get_pblock sl_filter] [get_cells sl_filter_inst ]
```

Bottom-up: single entity. w/o contain routing



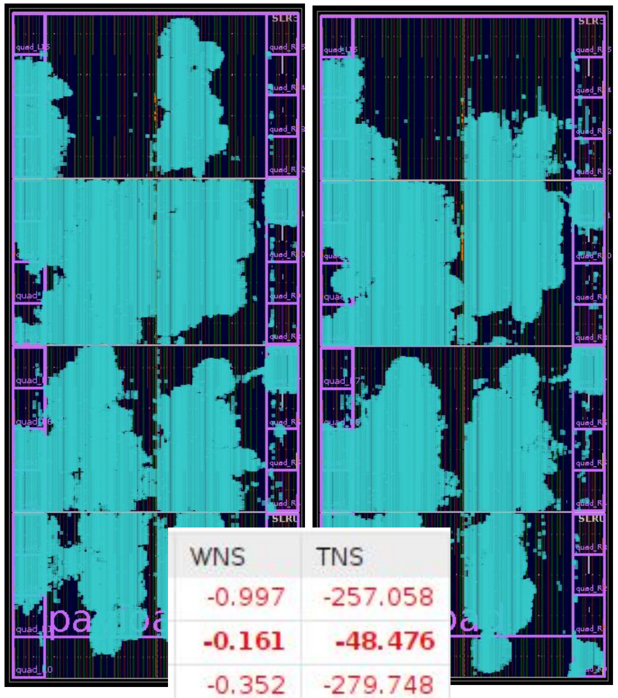
- Single entity plus input/output registers of wrapper
- Closes timing, seems ok
- But uses routing resources outside of pblock
- Will not work due to congestion in final design



Work in progress

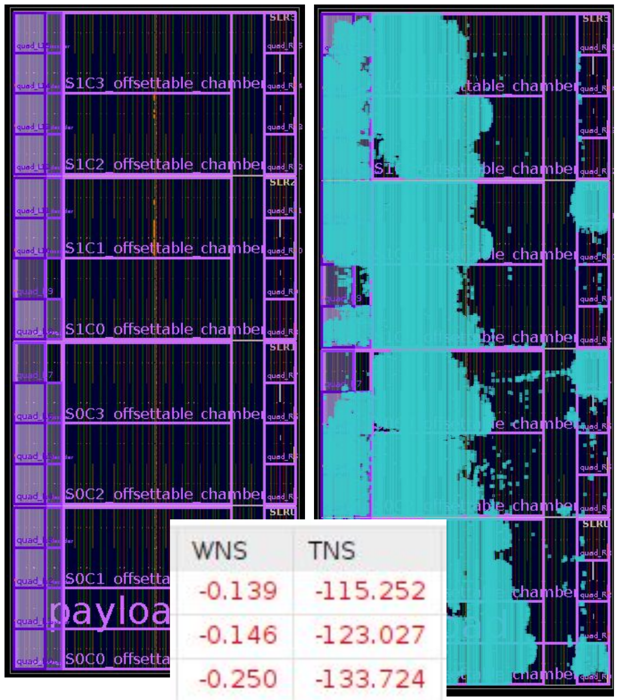
Work in progress

Attained results



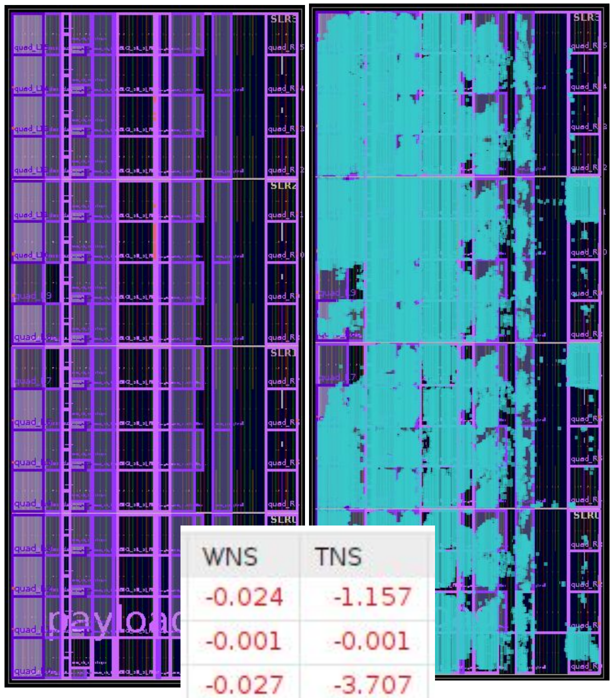
WNS	TNS
-0.997	-257.058
-0.161	**-48.476**
-0.352	-279.748
-1.031	-141.833
-1.302	-698.481
-1.109	-507.989

Only infrastructure placed algorithm → 0 pblocks (*)



WNS	TNS
-0.139	-115.252
-0.146	-123.027
-0.250	-133.724
-0.102	**-81.627**
-0.271	-827.066
-0.154	-208.935

SLR Xings + 8 chambers algorithm → 24 pblocks (*)



WNS	TNS
-0.024	-1.157
-0.001	-0.001
-0.027	-3.707
-0.025	-2.996
0.003	**0.000**
-0.019	-0.930

Latest draft placement algorithm → 208 pblocks

(*) RTL for SLR crossings and piping optimized thanks to fine-grained placement, then removed placement

[VHDL record serializer/deserializer methodology]

You've got your data nicely structured in a vhdl record and want to convert a to a std_logic_vector and back again (e.g. for a fifo or tx over serial link). You &-concatenate. You slice the output. Your change the record. You take a deep breath. You feel miserable. You *bug*.

```vhdl
-- Flatten/Unflatten mydata_t
```

```vhdl
type mydata_combo_t is record
  slv : std_logic_vector(MYDATA_SIZE-1 downto 0);
  obj : mydata_t;
end record;
```

```vhdl
function flatten_mydata(obj : mydata_t) return std_logic_vector is
begin
  return flatten_unflatten_mydata( obj => obj ).slv;
end function;

function unflatten_mydata(slv : std_logic_vector) return mydata_t is
begin
  return flatten_unflatten_mydata( slv => slv ).obj;
end function;
```

```vhdl
function flatten_unflatten_mydata(
  obj : mydata_t                                  := MYDATA_NULL      ;
  slv : std_logic_vector(MYDATA_SIZE-1 downto 0)  := (others => '0') )
return mydata_combo_t is
  variable combo : mydata_combo_t;
  variable cursor : integer range MYDATA_SIZE downto 0 := 0;
begin
  combo.obj.slv_field :=
        slv( WIDTH_SLV_FIELD + cursor - 1 downto cursor ) ;
  combo.slv( WIDTH_SLV_FIELD + cursor - 1 downto cursor ) :=
    obj.slv_field;
  cursor :=  WIDTH_SLV_FIELD + cursor;

  combo.obj.usgn_field := unsigned(
        slv( WIDTH_USGN_FIELD + cursor - 1 downto cursor ) );
  combo.slv( WIDTH_USGN_FIELD + cursor - 1 downto cursor ) :=
    std_logic_vector( obj.usgn_field );
  cursor :=  WIDTH_USGN_FIELD + cursor;

  combo.obj.bit_field :=
        slv( 1 + cursor - 1 ) ;
  combo.slv( 1 + cursor - 1 ) := obj.bit_field ;
  cursor :=  1 + cursor;

  for i in combo.obj.array_of_sub_records'range loop
    combo.obj.array_of_sub_records(i) := unflatten_sub_record(
        slv(  SUBRECORD_SIZE + cursor - 1 downto cursor ) );
    combo.slv(  SUBRECORD_SIZE + cursor - 1 downto cursor ) :=
      flatten_sub_record( obj.array_of_sub_records(i) );
    cursor :=   SUBRECORD_SIZE + cursor;
  end loop;

  return combo;
end function;
```

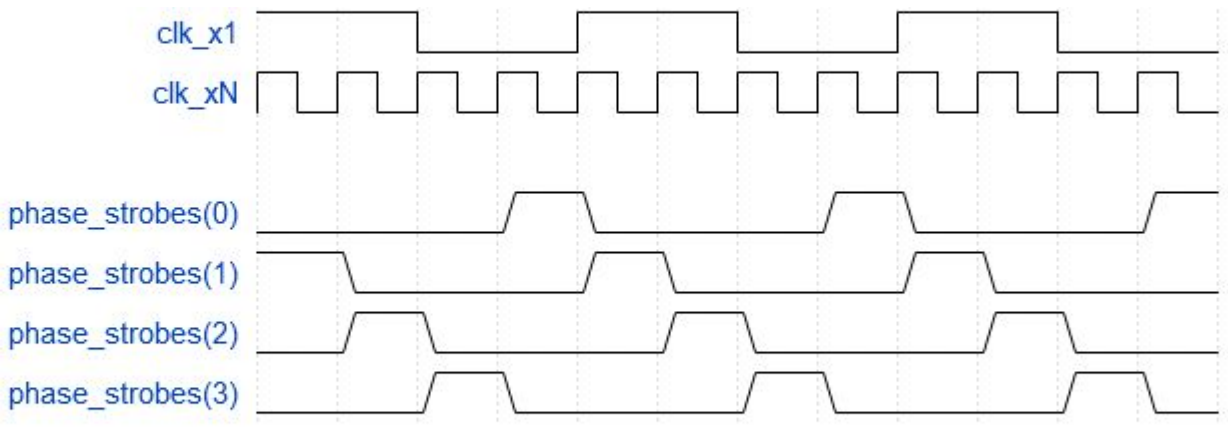
[Clock phase strobes]

Working family of related clocks (eg. LHC clock x1, x9, x12)

Sometimes code running in fast clock needs to know phase relationship to parent clock:

- data passing between parent and its derived clock
- data passing between sibling clocks, forcing write and read on edges known to be safe
- …

clock_strobes.vhd takes the two clocks and generates array of phase strobes (example N=4):



Occupies <5 Luts and <20 FFs ⇒ don't bother distributing strobes, just instantiate a copy wherever you need

Conclusions

- I thought it might be useful to have a framework to help with placement that makes it a little more friendly and I am working on it.
- It’s very beta but already making my life easier with placement
- Placement helps a lot with identifying problems with RTL
- Placement can make a difference in getting those last 100 ps
- Wishlist/future work in backup
- Record (de)serializaton and clock_strobes also save me time (and bugs)

Discussion

- Does it sound interesting?
- Any advice on how to make it better or what else to include?
- Or any tips on timing closure in general that you want to share?
- …?

```
end package body;
```

Wishlist/future work, sorted by increasing unlikeliness

- refactoring for better quality code
- non-rectangular pblocks, could improve RAM/DSP resources usage
- increase the vertical granularity
- nested pblocks: could make sense to give space to an entity, and hand place a particularly critical subset of its registers. Currently achieved by just doing 2 pblocks, one that includes the other
- some columns have either slices or laguna depending on their vertical position on the SLR. currently just assuming they're always slices
- LUTs used for RAM are not taken into account. makes no distinction on different kinds of slices or usages for LUTs
- Provide feedback on costly routing: some horizontal/vertical paths have more dramatic effects on timing than others (crossing columns of DSPs or IO blocks) Could this info be incorporated to provide more info to the user doing the placement?
- automatic reading of vivado outputs to calculate resource utilization
- other FPGAs? other vendors?
- a nice GUI