(O)TMB Pattern Logic

10 April 2018

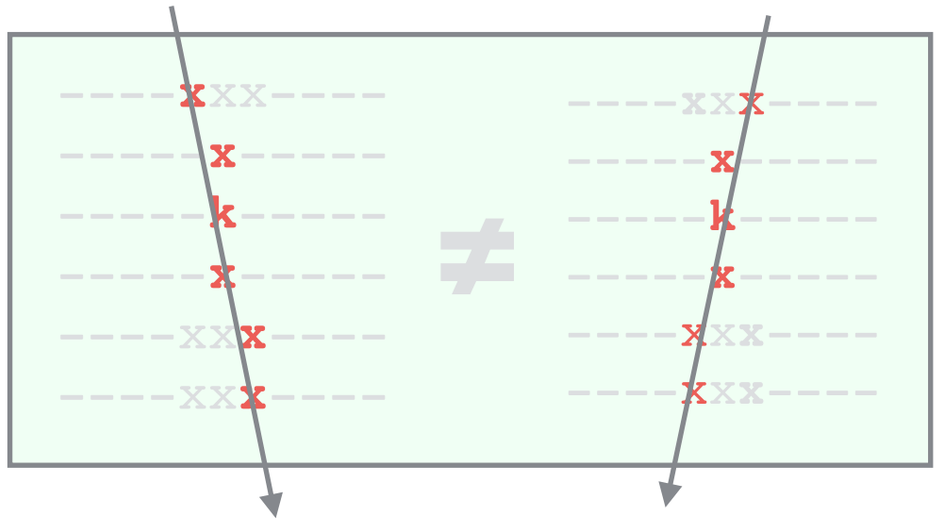
Andrew Peck

Introduction

- Work by TAMU team showed that good improvement in position and angular resolution is possible by fitting comparator hits
- Some difficulty in implementation
  - Fitting directly in firmware is slow, complicated, and inflexible
  - Looked into possibility of implementing a lookup-table approach to “fitting” inside of the OTMB

- Initial work looks very promising
  - Fits well, very easy to work into firmware

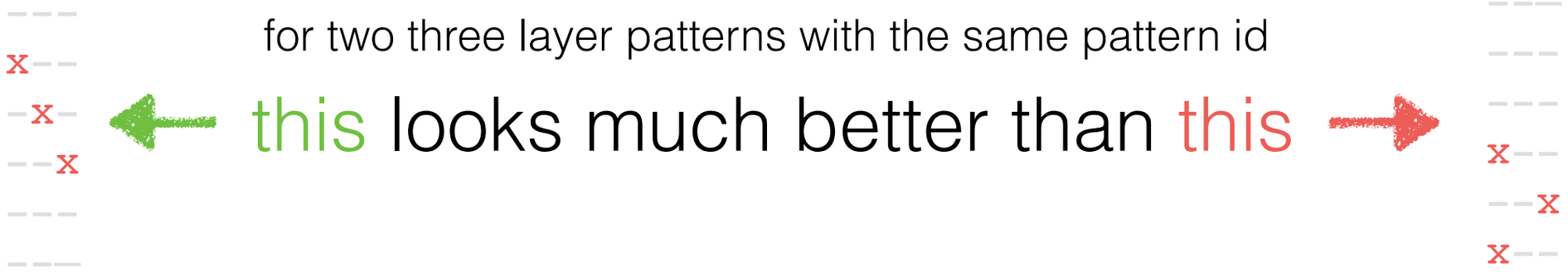
Clearly these two segments are different but present definitions cannot distinguish



patA: $\phi \approx -6^{\circ}$ $\phi \approx +6^{\circ}$

Introduction

- In addition to better resolution achieved by lookups…
  Lookup tables may also have benefits for noise reduction by rejecting segments w/ bad fits
  - Seems especially useful with 3 layer (pre)CLCT requirement




```
        0123456789A
// ly0 ---xxx-----
// ly1 ----xx-----
// ly2 -----k-----
// ly3 -----xx----
// ly4 -----xxx---
// ly5 -----xxx---
```

Presently these are both equal quality 3 layer segments in pattern #9 (2nd best)

LUT would allow rejection of one while keeping the other

```
        0123456789A
// ly0 ---xxx-----
// ly1 ----xx-----
// ly2 -----k-----
// ly3 -----xx----
// ly4 -----xxx---
// ly5 -----xxx---
```

- Thresholds don't have to be simply # of layers hit
- On pattern by pattern basis can study and tune the OTMB response
- Can reduce pattern threshold to only 3 layers, and use a reduced subset of 3 layer patterns
- Studies would be critical to optimize these capabilities

Implementation

- Simplest idea: Follow the lead of the track finder
  - Use coarse pattern or “roads” to identify segment candidates
  - Instead of just stopping at the roads, we feed the specific hits on the road to a lookup-table
    - ✦ Lookup-table allows software defined response to each possible set of hits
- With post-processing on the patterns, the difficulty of optimizing patterns is significantly reduced
    - ✦ Pattern definitions to focus on efficiency; don’t need to have fine resolution patterns
    - ✦ Lookup-tables can suppress noise created by widening patterns
    - ✦ Potentially can reduce the number of patterns without losing efficiency
      - Reducing logic by eliminating patterns is the only foreseeable path to a possible implementation in the standard TMBs

```
          0123456789A                    0123456789A
// ly0 ----xxx----          // ly0 ----xxx----
// ly1 -----x-----          // ly1 ----xxx----
// ly2 -----k-----    →     // ly2 ----xxx----
// ly3 -----x-----          // ly3 ----xxx----
// ly4 ----xxx----          // ly4 ----xxx----
// ly5 ----xxx----          // ly5 ----xxx----
```

Implementation

- (O)TMB pattern finding is based around cascaded priority encoders
  - 1) Each (D)CFEB is sorted to produce one or zero CLCT "candidates"
    - "best1of32"
  - The CLCT "candidates" are sorted to choose the best CLCT from the whole chamber
    - "best1of7" (OTMB) / "best1of5" (TMB)
- Lookup-table implementation uses the existing sorting logic to sort candidates based on the same algorithm (# of layers, pattern ID)
  - But along the way, has to carry a compressed version of the comparator hits
    - Dubbed "comparator codes"
    - Carried comparator codes from pre-sorted CLCT candidates are fed into lookup table to produce final CLCT output w/ better resolution

Implementation

- Compression of comparator hits into comparator codes uses a very simple scheme:
  - Take advantage of the idea that neighboring half-strips should not fire simultaneously
  - We can:
    - 1) Compress three halfstrips into 2-bits
    - 2) Concatenate together 6 layers into a single 12 bit number ==> “comparator code”

Example:

Compress each layer to 2 bits

Concatenate together all 6 layers

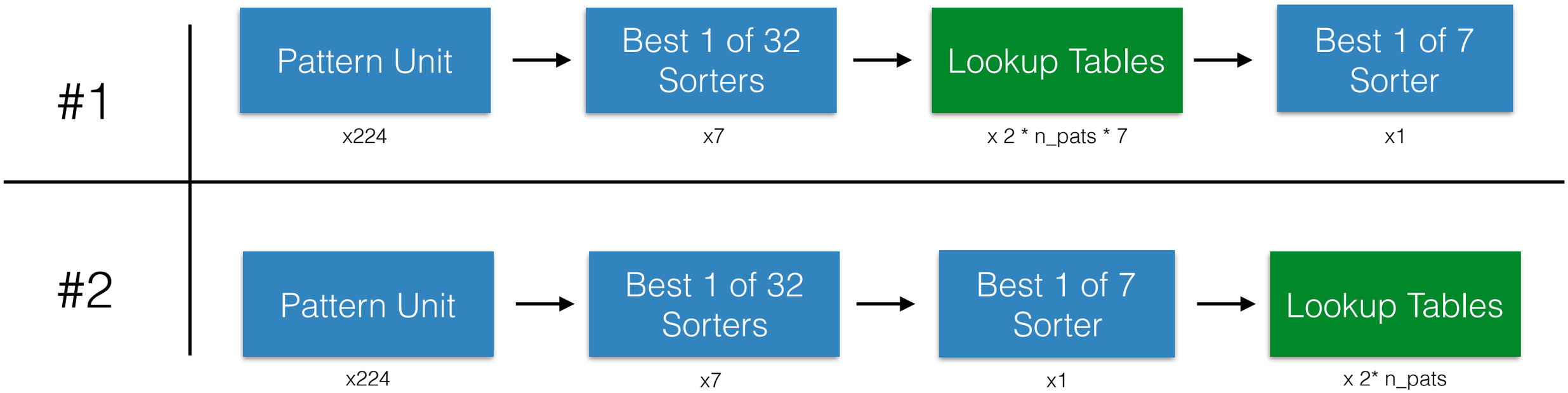
Form a single 12 bit number

```
        0123456789A
// ly0 ----xxx----        // ly0 0x0
// ly1 ----1xx----        // ly1 0x1
// ly2 ----x1x----        // ly2 0x2
// ly3 ----xx1----        // ly3 0x3
// ly4 ----xx1----        // ly4 0x3
// ly5 ----xx1----        // ly5 0x3

comparator code =
        (0 << 0*2)
      | (1 << 1*2)
      | (2 << 2*2)
      | (3 << 3*2)
      | (3 << 4*2)
      | (3 << 5*2)

0xFE4
```

- 12 bits comparator code fits perfectly into the address size of a 36k block ram in the Virtex-6 FPGA
  - For each (O)TMB pattern, need to generate and save this number for input to the lookup table
  - Get 9 bits output per 36k block RAM ==> use 2 BRAMs per pattern to get 18 bits of output

Implementation

- Two places to insert the lookup tables:

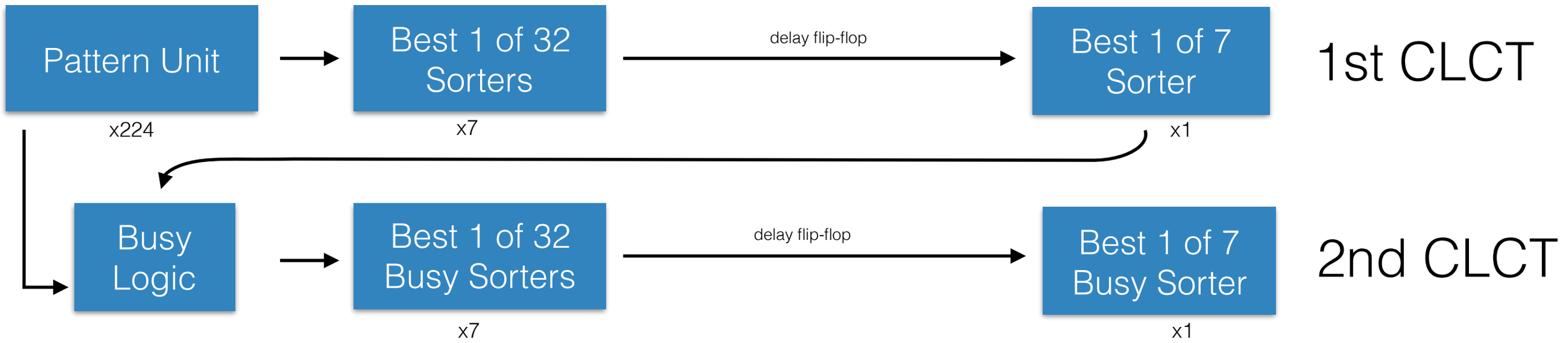


- Choice #1
  - Lets the final sorting step be done based on the Output of the lookup table
    - Can use higher quality fit as final sorting metric, good for very high rates (?)
    - Uses more block rams
- Choice #2
  - Only does the lookup once… all prior sorting is based on just # of layers, pattern ID
- Both are possible in the OTMB… #2 is the only possible choice in standard TMB (if it fits at all)

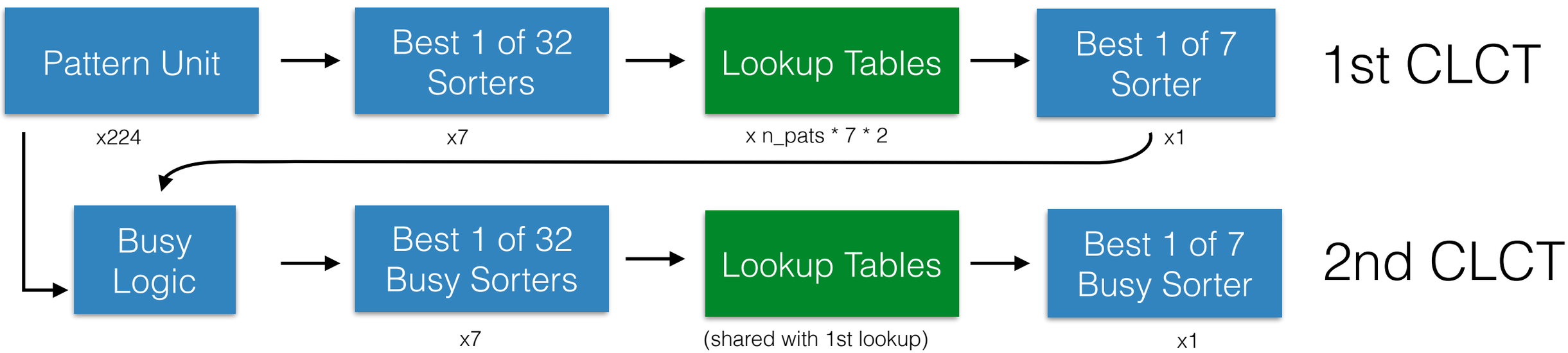
OTMB Pattern Logic

No added latency for lookup… use an existing "extra" flipflop (added to maintain legacy timing)

Current OTMB Pattern Finding Logic:



Modified OTMB Pattern Finding Logic:



OTMB Pattern Finding Logic

- Most of the OTMB logic remains identical, even with new scheme
  - Very simple implementation
  - No timing changes, no complex rewrites, no huge changes in firmware, maintains legacy functionality as compile-time flag
  - Latency does not increase
    - ✦Pattern finder contains an unnecessary flip-flop, delaying CLCT production to “maintain timing with legacy sequencer”

- OTMB Implementation:
  - No latency change, ~reduced logic usage, increased block RAM usage
  - Wrote prototype firmware as proof of concept
    - ✦Tested logic in simulator
    - ✦Compiles OK
  - Block RAM usage:
    - ✦12 bit inputs fit perfectly into block 36k block RAMs
    - ✦2 RAMs per pattern per DCFEB give 18 output bits per lookup (more than enough)
    - ✦Occupies ~70 RAMs currently.. reducible to 42 by reusing left/right bends (should be easy)

DCFEB Active FEB Flags

- At first glance… widening the pattern roads and/or decreasing layer threshold would increase pretrigger rates
  - Don't want to increase DCFEB bandwidth usage

- But DCFEBs no longer require early pretrigger
  - DCFEB ring buffer is sufficient for a delayed active_feb_flag
  - On OTMB, one of the lookup-table output bits can be an active_feb_flag
    - Lookup-table can be programmed to implement any desired active_feb_flag thresholds
      - ✦ e.g. Can require 4 layers, or can require high quality fit, etc, fully defined by software

- Implementation of this adds some complication:
  - No apparent showstopper but it will require care
  - Also changes DCFEB pipeline depth (just change a VME setting to accommodate)
  - Software studies are critical

<u>This approach probably not possible in standard TMB due to added pretrigger latency</u>

OTMB Firmware Plans

- Wrote demonstrator during free time last summer
  - Initial check in simulation → **OK**
  - Meant only as a quick proof of concept to check the idea, feasibility, logic usage, but should be a good basis to start working on production quality firmware
  - Backwards compatible (no changes to data format)

- Next steps:
  - Finish and first implementation of LUT firmware
    - ✦ Target fully backwards compatible implementation
      - Same data format, trigger format
      - Should be “plug-and-play” with current OTMB firmware
    - ✦ Test on bench at UCLA (using “emulator board”, and check at b904)
  - Coordinate w/ trackfinder team to understand options and goals for a modified data format to encode more position / angular information
  - Update firmware, unpacker, online software, etc. with new implementation

Firmware Usage Comparison

Standard OTMB firmware

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	28,446	301,440	9%	
Number used as Flip Flops	28,401			
Number used as Latches	19			
Number used as Latch-thrus	0			
Number used as AND/OR logics	26			
Number of Slice LUTs	57,681	150,720	38%	
Number of RAMB36E1/FIFO36E1s	54	416	12%	
Number using RAMB36E1 only	51			
Number using FIFO36E1 only	3			
Number of RAMB18E1/FIFO18E1s	137	832	16%	
Number using RAMB18E1 only	137			
Number using FIFO18E1 only	0			

ROM_LUT implementation w/ reduced pattern set (5) & wider patterns

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	24,742	301,440	8%	
Number used as Flip Flops	24,711			
Number used as Latches	5			
Number used as Latch-thrus	0			
Number used as AND/OR logics	26			
Number of Slice LUTs	41,962	150,720	27%	
Number of RAMB36E1/FIFO36E1s	5	416	1%	
Number using RAMB36E1 only	2			
Number using FIFO36E1 only	3			
Number of RAMB18E1/FIFO18E1s	137	832	16%	
Number using RAMB18E1 only	137			
Number using FIFO18E1 only	0			

ROM_LUT implementation w/ current patterns

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	28,974	301,440	9%	
Number used as Flip Flops	28,894			
Number used as Latches	54			
Number used as Latch-thrus	0			
Number used as AND/OR logics	26			
Number of Slice LUTs	48,356	150,720	32%	
Number of RAMB36E1/FIFO36E1s	40	416	9%	
Number using RAMB36E1 only	37			
Number using FIFO36E1 only	3			
Number of RAMB18E1/FIFO18E1s	137	832	16%	
Number using RAMB18E1 only	137			
Number using FIFO18E1 only	0			

LCT Data Format

Current Format

Data Field	Description
alct0 key [6:0]	ALCT key layer wiregroup
clct0 pat [3:0]	CLCT bend pattern
lct0 quality [3:0] * lct0_vpf	LCT quality
lct0 vpf	LCT valid pattern flag
clct0 bend	CLCT bend direction
clct0 cfeb/halfstrip [7:0]	CLCT (D)CFEB
clct0 sync err	CLCT0 Sync Error Flag
alct0 bxn [0]	ALCT BX flag
clct bx0 * lct0_vpf	CLCT BX flag
csc_id [3:0] * lct0_vpf	Trigger ID

Idea for modified format

remove dedicated (redundant)
bend and vpf bits

Data Field	Description
alct0 key [6:0]	ALCT key layer wiregroup
clct0 pat [4:0]	CLCT bend pattern
lct0 quality [3:0] * lct0_vpf	LCT quality
clct0 cfeb/halfstrip [8:0]	CLCT (D)CFEB
clct0 sync err	CLCT0 Sync Error Flag
alct0 bxn [0]	ALCT BX flag
clct bx0 * lct0_vpf	CLCT BX flag
csc_id [3:0] * lct0_vpf	Trigger ID

Standard TMB Implementation (?)

- TMB is extremely resource starved…
  - Slices are 100% occupied, RAM is 100% full
  - Implementation will be difficult

- Implementation may be possible… but very uncertain right now
  - Can perform lookup only on final 2 LCT outputs (instead of 1 lookup per CFEB)
    - Would need to reduce number of patterns to 3 total to free logic
      - Straight/left/right.. apparently little efficiency loss.. but needs to be investigated further
    - Requires 8 18 kb rams (120 total—currently ALL are used)
      - Need to identify candidates for removal, e.g. RPC buffers. Not sure yet
    - Increases latency by at least 1 bx
    - No way to use LUTs for pretrigger (due to CFEB SCA requirements)
    - Needs more work to know if this will work

Summary

- Improvement of CLCT resolution in firmware is obtainable
  - Studies already show the benefit
  - Need to work on firmware implementation and details
  - Need to understand how this can be used in the trackfinder
  - Higher resolution CSC trigger primitives also more useful for GEM matching
  - Need to emulate firmware changes

- Firmware implementation in progress
  - Needs more work, especially on active feb flag formation

- Software studies and training of the lookup tables in progress
  - Loaded via text file, easy to change

- Standard TMB implementation may be possible… but not straightforward