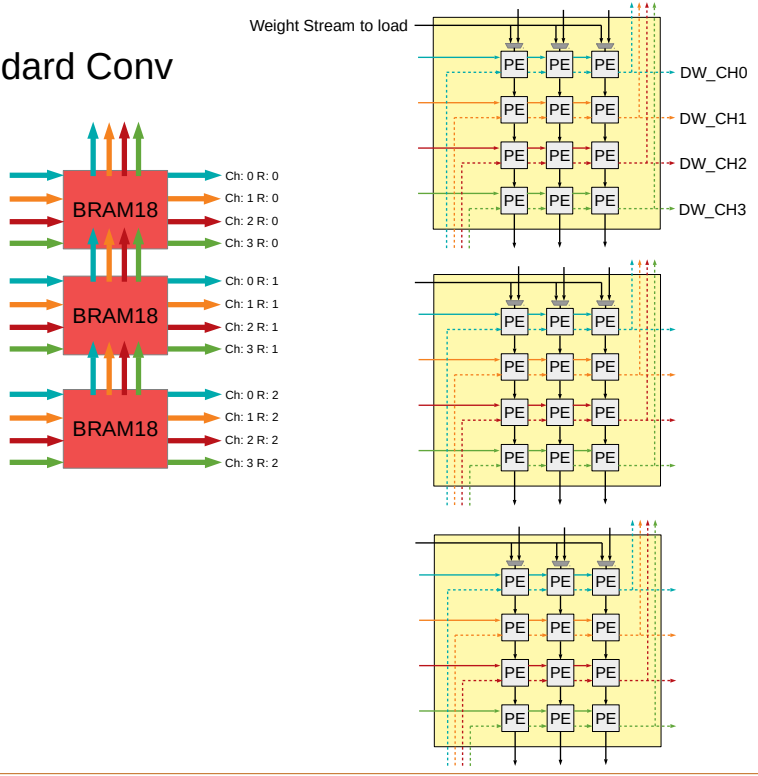
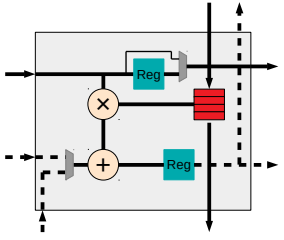
MLBlocks: Arming FPGA architectures with Dense & Low Precision units in classic column based manner

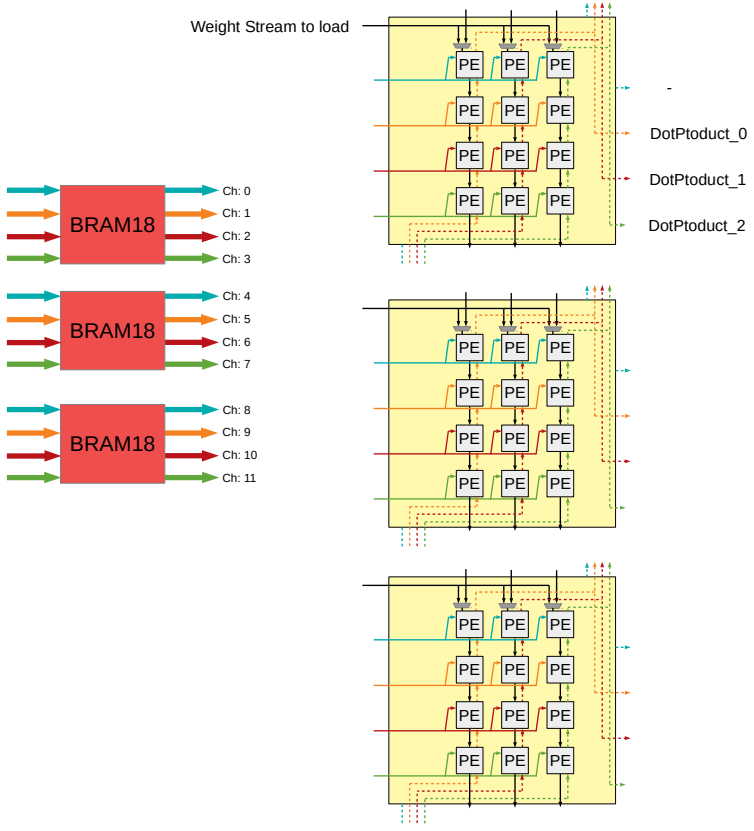
DWConv & Standard Conv



PE



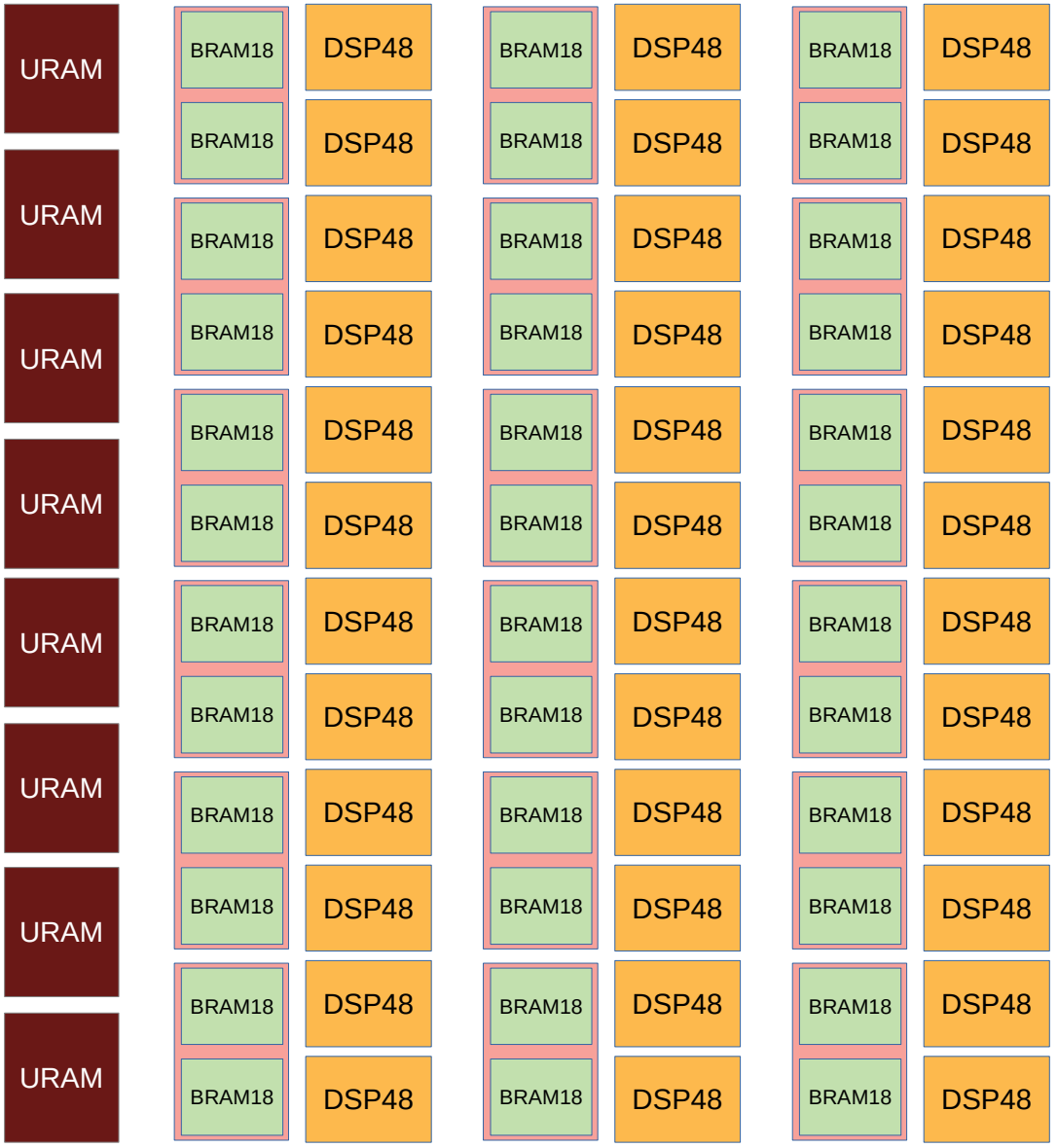
PWConv & Matrix-Matrix Multiplication



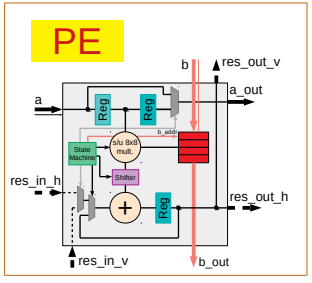
Contributions:

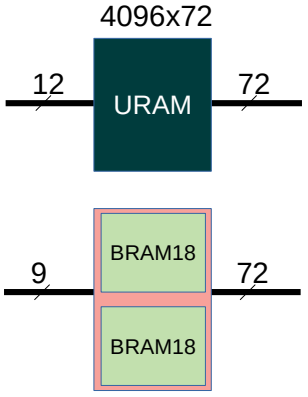
1- Reconfigurable PE (**DSP size**, Systolic array, **Column based**)
6 times more 8x8 multiplier comparing to a DSP Block (two 8x8),
RS Data flow, High frequency, flexible data movement. Great for **SConv, DWConv, PWConv, Matrix-Matrix Multiplication**
DPS – BRAM ratio 1/1 (same as Ultrascale+ arch), Low number of intermediary outputs in practice
Parameterized (for any budget limitation) – can integrate multi precision idea
2- Compare with cascade paper (Prof. Nachiket)
3- new suggestion to use each 18KBRAM as 36bit streamer using external controler circuit (delivering 662MHz) (in cascade paper: 18bit)

UltraScale+ architecture distribution:

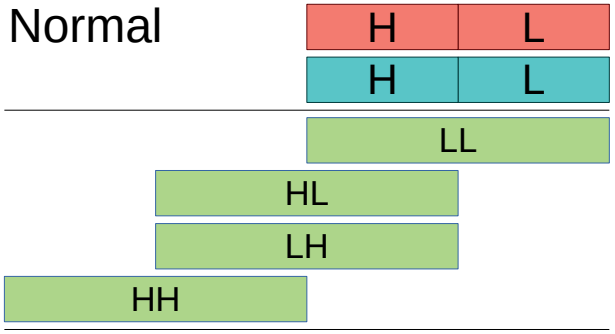


Virtex 7 → **28**nm DSP48**E1** Fmax=742MHz
Virtex U → 20nm
Virtex UP → 16nm

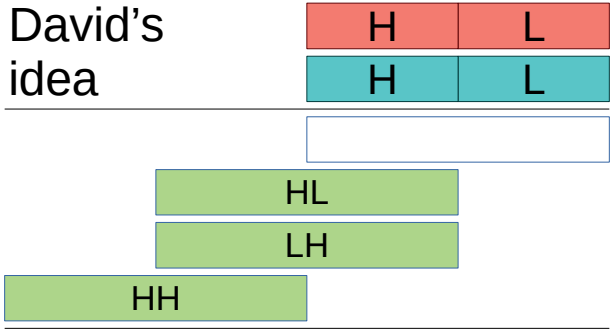




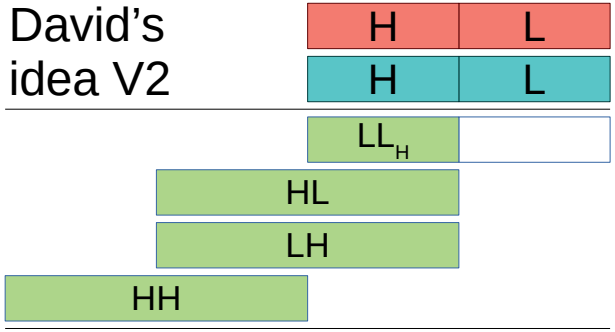
Normal

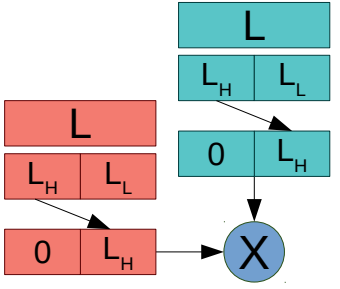


David's idea

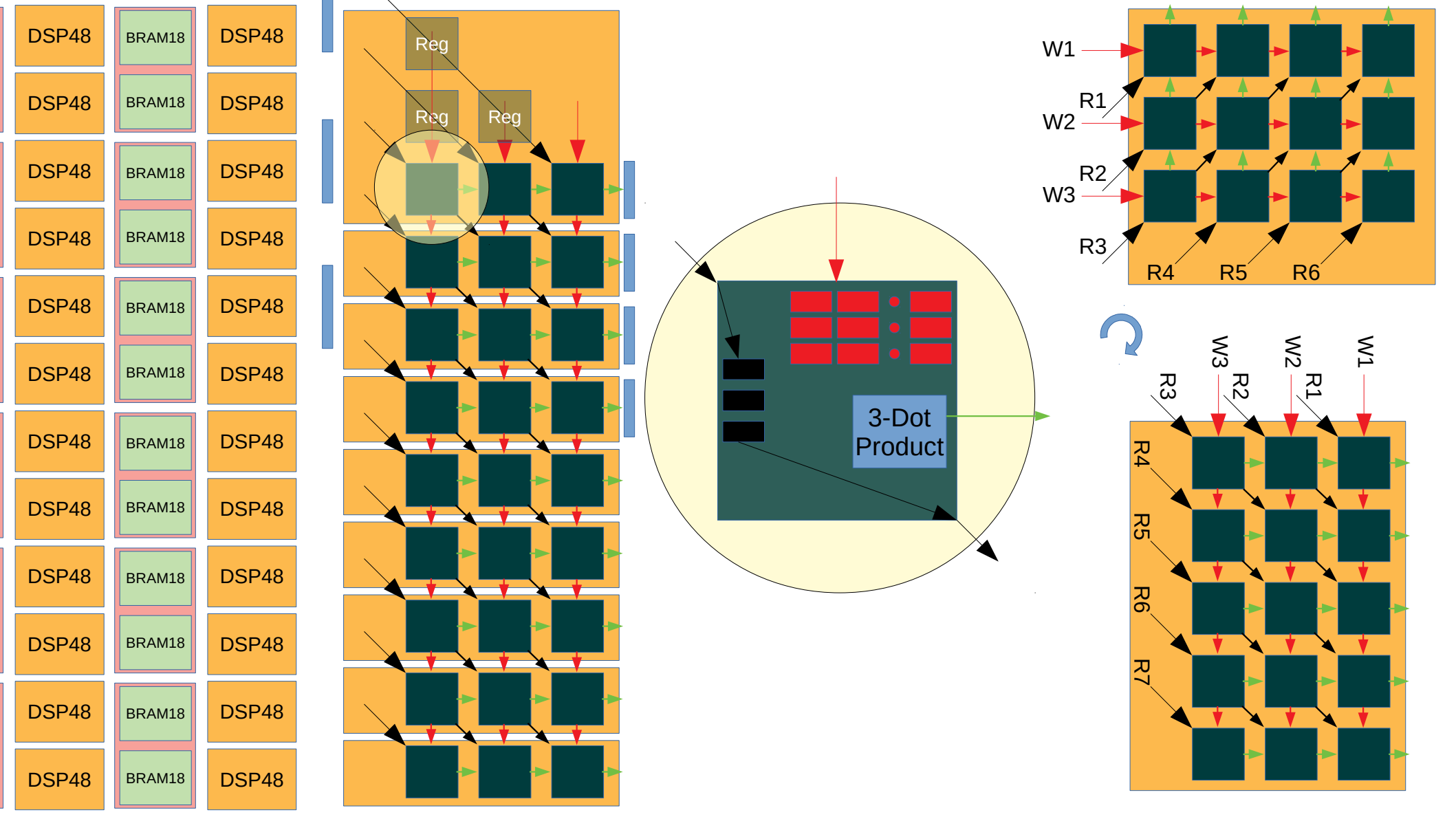


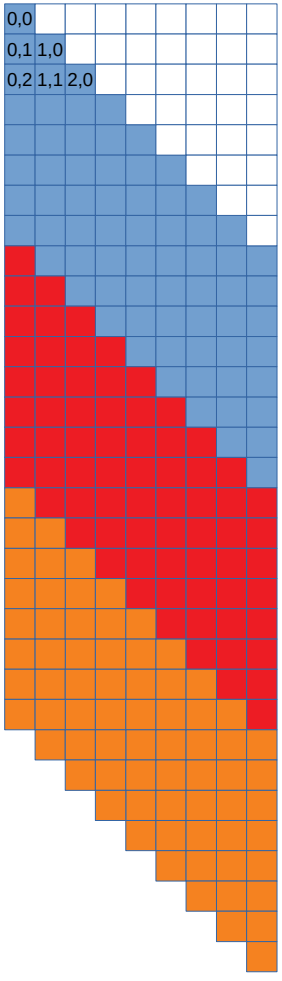
David's idea V2

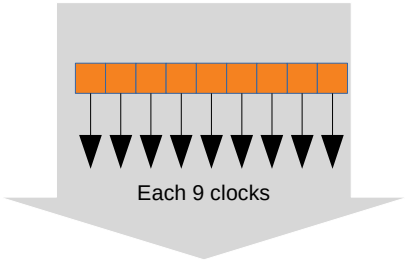


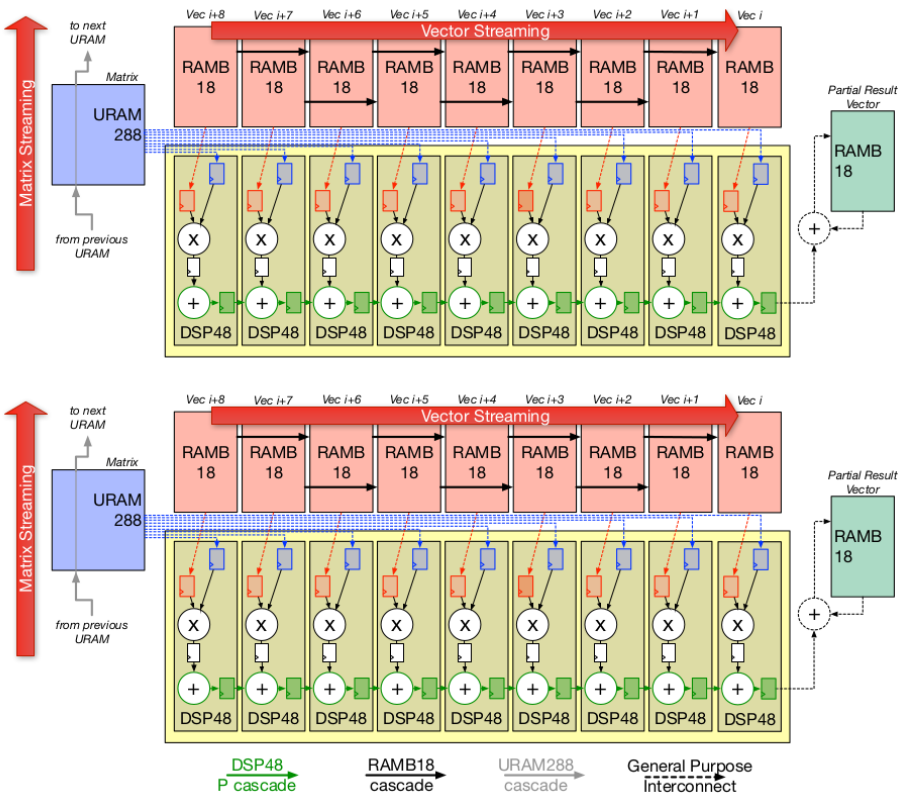


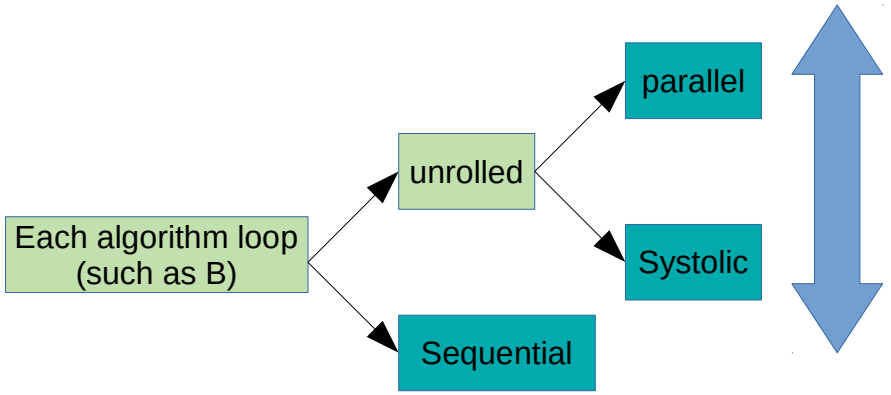
My amassing MLBlocks world


DSP48
BRAM18
Reg
Reg
Reg
3-Dot
Product
W1
R1
W2
R2
W3
R3
R4
R5
R6
W3
W2
W1
R3
R2
R1
R4
R5
R6
R7


0,0
0,1 1,0
0,2 1,1 2,0


Each 9 clocks


Matrix Streaming
to next URAM
Matrix
URAM 288
from previous URAM
Vec i+8
Vec i+7
Vec i+6
Vec i+5
Vec i+4
Vec i+3
Vec i+2
Vec i+1
Vec i
Vector Streaming
RAMB 18
DSP48
Partial Result Vector
DSP48 P cascade
RAMB18 cascade
URAM288 cascade
General Purpose Interconnect



Dis Parallel:

1- more fan in and outs (since we are talking about small Pes it is fine)

Dis Systolic:

1- tougher scheduling, rythmic scheduling

2- prevent circuit fusions (less optimization)

$B = B_{Seq} \times B_{par} \times B_{Sys}$

# of Physical MAC:	$\times B_{par} \times B_{Sys}$	
# of Input:	$\times B_{par}$	(without internal serial to parallel)
# of Output:	$\times B_{par}$	

Params = {right side indexes}

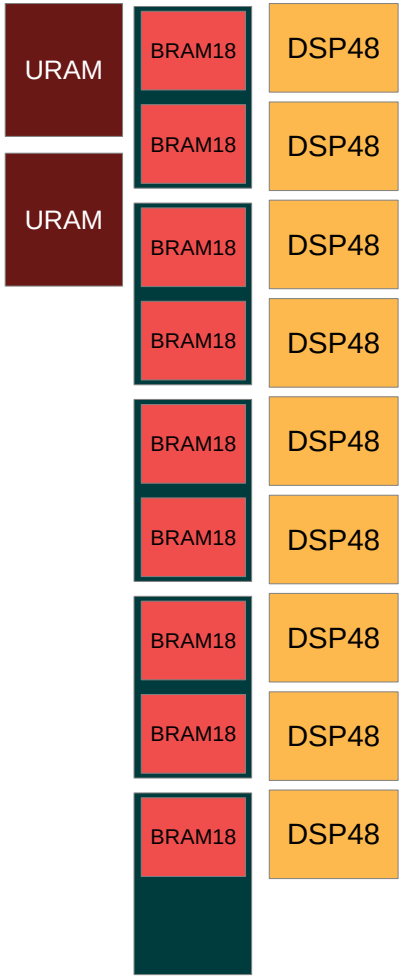
for $param^{i}_{sch\_0}$: $0 \rightarrow sch^{i}\_0$

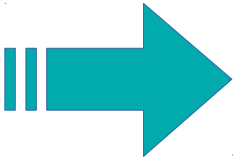
for $param^{i}_{sch\_1}$: $0 \rightarrow sch^{i}\_1$

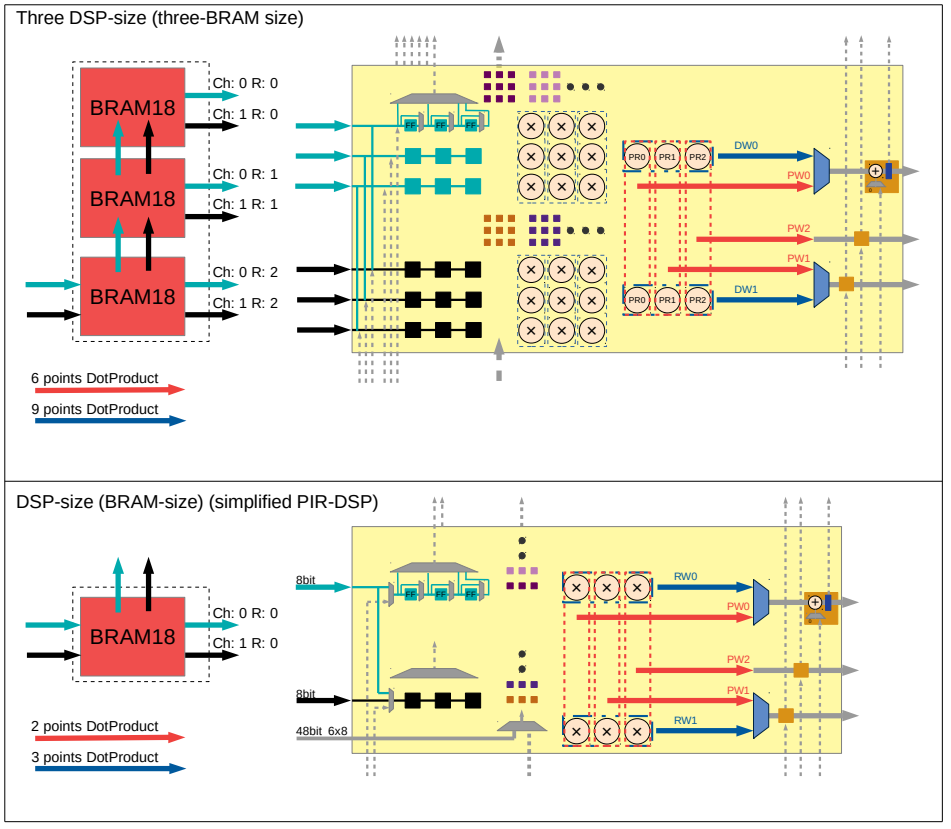
for $param^{i}_{sch\_2}$: $0 \rightarrow sch^{i}\_2$

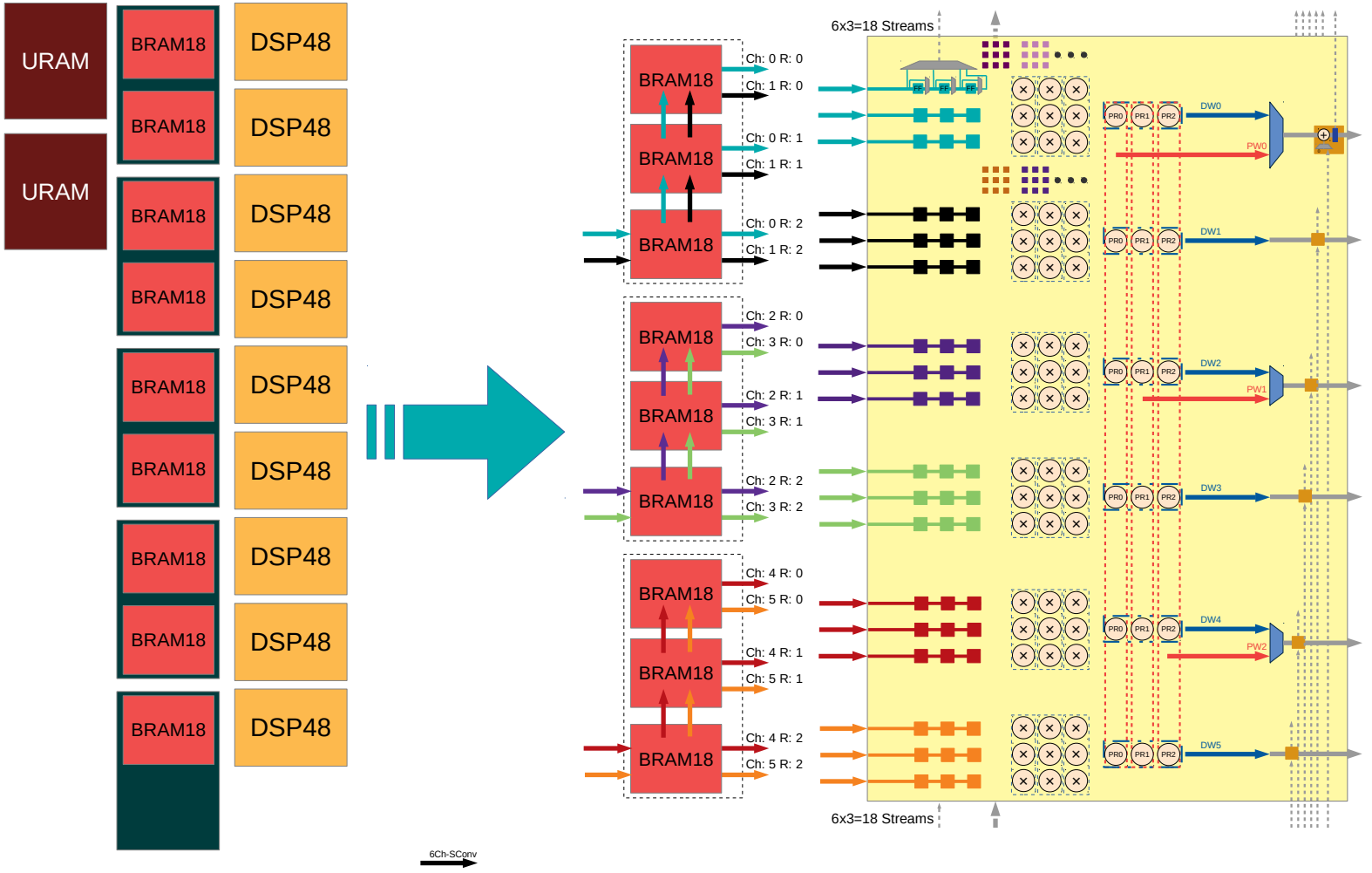
for $param^{i}_{seq}$: $0 \rightarrow comp\_seq^{i}$

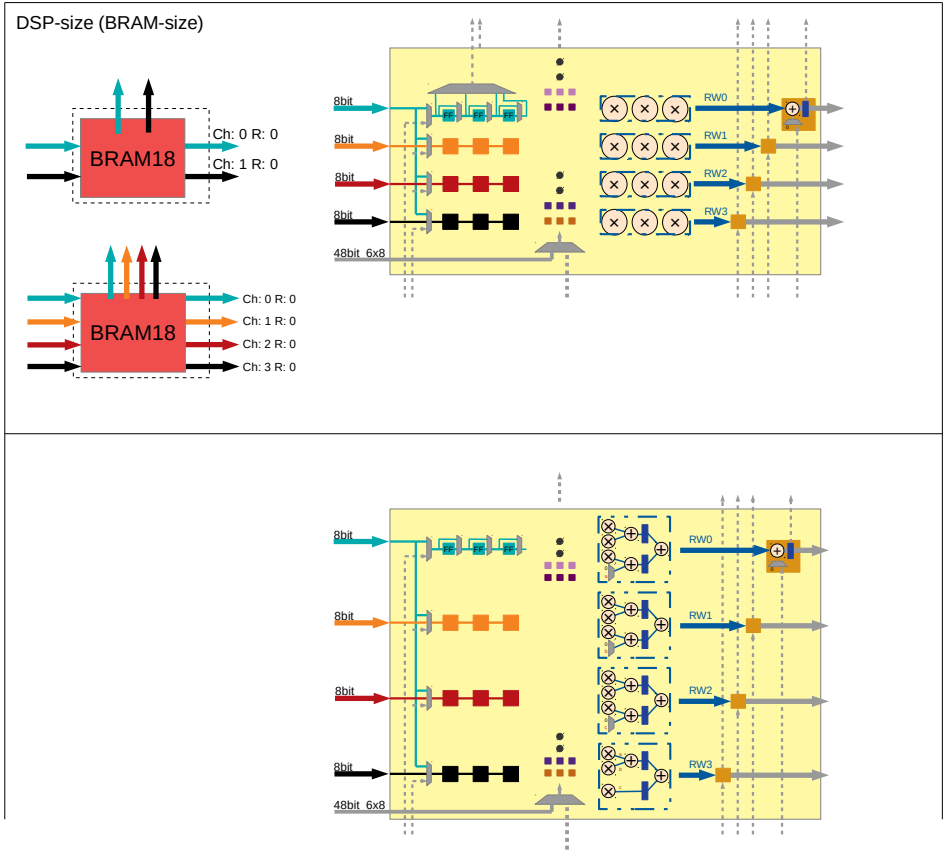
for $param^{i}_{un}$: $0 \rightarrow comp\_un^{i}$


URAM
URAM
BRAM18
BRAM18
BRAM18
BRAM18
BRAM18
BRAM18
BRAM18
BRAM18
BRAM18
BRAM18
DSP48
DSP48
DSP48
DSP48
DSP48
DSP48
DSP48
DSP48
DSP48

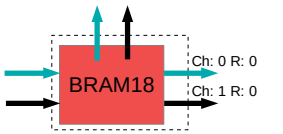


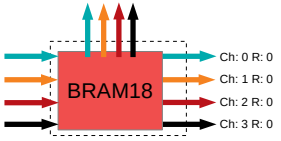

Three DSP-size (three-BRAM size)
BRAM18
BRAM18
BRAM18
Ch: 0 R: 0
Ch: 1 R: 0
Ch: 0 R: 1
Ch: 1 R: 1
Ch: 0 R: 2
Ch: 1 R: 2
6 points DotProduct
9 points DotProduct
PR0
PR1
PR2
DW0
PW0
PW2
PW1
DW1
DSP-size (BRAM-size) (simplified PIR-DSP)
BRAM18
Ch: 0 R: 0
Ch: 1 R: 0
2 points DotProduct
3 points DotProduct
8bit
8bit
48bit 6x8
RW0
PW0
PW2
PW1
RW1

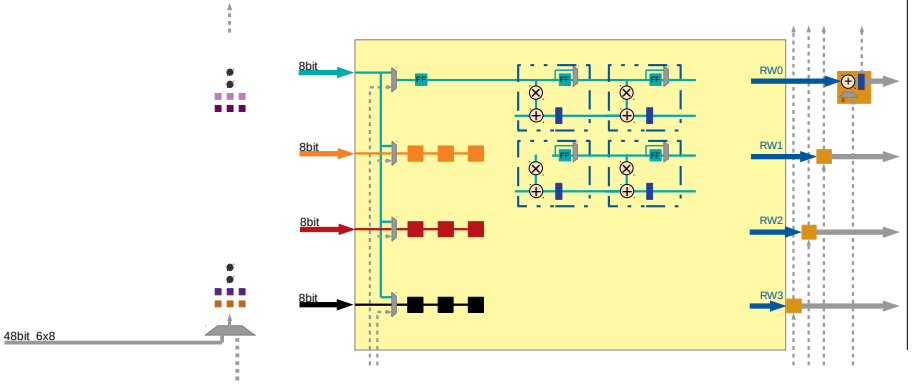

URAM
URAM
BRAM18
DSP48
6x3=18 Streams
Ch: 0 R: 0
Ch: 1 R: 0
Ch: 0 R: 1
Ch: 1 R: 1
Ch: 0 R: 2
Ch: 1 R: 2
Ch: 2 R: 0
Ch: 3 R: 0
Ch: 2 R: 1
Ch: 3 R: 1
Ch: 2 R: 2
Ch: 3 R: 2
Ch: 4 R: 0
Ch: 5 R: 0
Ch: 4 R: 1
Ch: 5 R: 1
Ch: 4 R: 2
Ch: 5 R: 2
PR0
PR1
PR2
DW0
DW1
DW2
DW3
DW4
DW5
PW0
PW1
PW2
6Ch-SConv


DSP-size (BRAM-size)
BRAM18
Ch: 0 R: 0
Ch: 1 R: 0
BRAM18
Ch: 0 R: 0
Ch: 1 R: 0
Ch: 2 R: 0
Ch: 3 R: 0
8bit
8bit
8bit
8bit
48bit 6x8
RW0
RW1
RW2
RW3
8bit
8bit
8bit
8bit
48bit 6x8
RW0
RW1
RW2
RW3

DSP-size (BRAM-size)





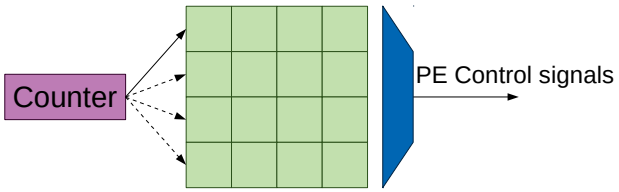


README.md

Primary Results

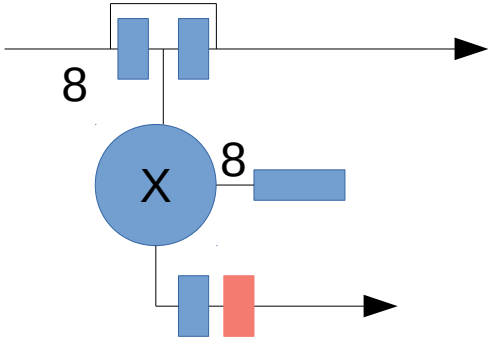
Size	Experiment	Computations (A,B)	A_D	B_D	ACC_D	SHIFTER	Area
DSP48		Dual8x8,27x18					7958
4x3	HP_all+R	8x8,8x16,16x8,24x8,8x24,16x16	6	6	2	HP	18340
4x3	HP_all	8x8,8x16,16x8,24x8,8x24,16x16	6	3	1	HP	13797
4x3	HP_most+R	8x8,8x16,16x8,24x8,16x16	6	4	2	HP	16778
4x3	HP_most	8x8,8x16,16x8,24x8,16x16	6	2	1	HP	13107
4x3	HP_semi+R	8x8,8x16,16x8,16x16	4	4	2	HP	16232
4x3	HP_semi	8x8,8x16,16x8,16x16	4	2	1	HP	12536
4x3	HP_apx+R	8x8,8x16,16x8,16x16apx	4	4	2	HP_apx	14318
4x3	HP_apx	8x8,8x16,16x8,16x16apx	4	2	1	HP_apx	10786
4x3	BYPASS+R	8x8+reuse	2	2	2	BYPASS	10721
4x3	BYPASS	8x8	2	1	1	BYPASS	**6445**
3x3	HP_all+R	8x8,8x16,16x8,24x8,8x24,16x16	6	6	2	HP	13760
3x3	HP_all	8x8,8x16,16x8,24x8,8x24,16x16	6	3	1	HP	10346
3x3	HP_most+R	8x8,8x16,16x8,24x8,16x16	6	4	2	HP	12641
3x3	HP_most	8x8,8x16,16x8,24x8,16x16	6	2	1	HP	9570
3x3	HP_semi+R	8x8,8x16,16x8,16x16	4	4	2	HP	12206
3x3	HP_semi	8x8,8x16,16x8,16x16	4	2	1	HP	9140
3x3	HP_apx+R	8x8,8x16,16x8,16x16apx	4	4	2	HP_apx	10686
3x3	HP_apx	8x8,8x16,16x8,16x16apx	4	2	1	HP_apx	**8161**
3x3	BYPASS+R	8x8+reuse	2	2	2	BYPASS	**8062**
3x3	BYPASS	8x8	2	1	1	BYPASS	**4825**
2x3	HP_all+R	8x8,8x16,16x8,24x8,8x24,16x16	6	6	2	HP	9246
2x3	HP_all	8x8,8x16,16x8,24x8,8x24,16x16	6	3	1	HP	**6944**
2x3	HP_most+R	8x8,8x16,16x8,24x8,16x16	6	4	2	HP	**8460**
2x3	HP_most	8x8,8x16,16x8,24x8,16x16	6	2	1	HP	**6447**
2x3	HP_semi+R	8x8,8x16,16x8,16x16	4	4	2	HP	**8173**
2x3	HP_semi	8x8,8x16,16x8,16x16	4	2	1	HP	**6160**
2x3	HP_apx+R	8x8,8x16,16x8,16x16apx	4	4	2	HP_apx	**7186**
2x3	HP_apx	8x8,8x16,16x8,16x16apx	4	2	1	HP_apx	**5544**
2x3	BYPASS+R	8x8+reuse	2	2	2	BYPASS	**5440**
2x3	BYPASS	8x8	2	1	1	BYPASS	**3267**

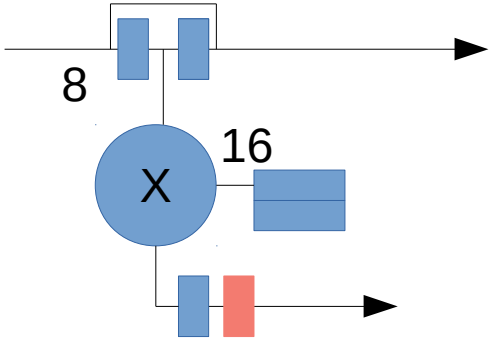
412MHz without pipeline

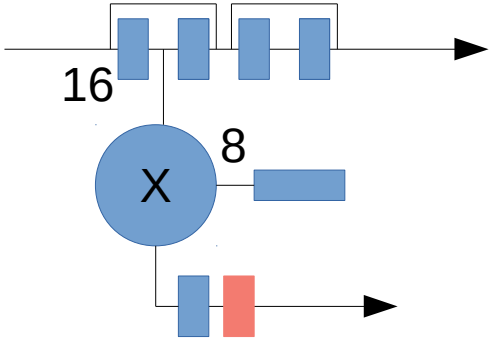


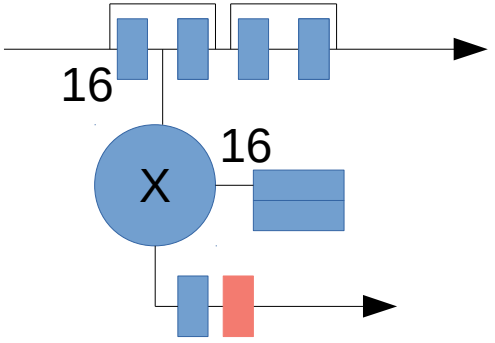
MLBlobk - PEFlex

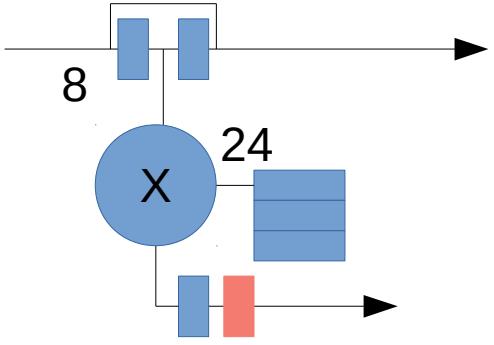
For reusement:
Each extra reusement requires one more results register

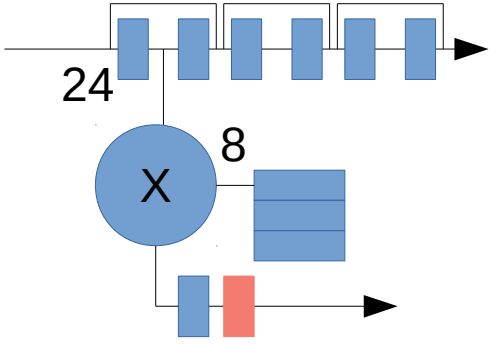












Why Dot product comparing to Systolic?

- Circuit fusion and optimisation
- Both have same unrolling factor
- Efficient pipelining rather than structured pipelines
- Register replacing and retiming
- Systolic are designed for better scaling (we are focusing on PE design which is small size)
  - If we talk for inside a PE ==> dot-product is better
  - If we explore PE-PE structure ==> Systolic manner
- Vector unit or systolic? Number of IO is much better in Systolic arrays
- My MLBlock benefit both Systolic-array and dot-product based accelerators.
  - Without using Systolic interconnections: MLBlock = a dot-product unit. Great for Supertile
  - Using interconnections: MLBlock = a column based systolic array structure Great for TPU/SeanFPT like

IO requirements

- Types:
  - Stream (Windowing) vs RAM:
    - Reasonable windowing
  - # of IOs
    - Less out
    - Maximum input

For every parameter in a given algorithm

$P = P_{Comp\_Sch} \times P_{Comp\_PE}$

$P_{Comp\_Sch} = \prod P_{Sch\text{-}i}$

$P_{Comp\_PE} = P_{Comp\_Un} \times P_{Comp\_Seq}$

- IO aspect of a PE

$P_{Comp\_Un} \times P_{Comp\_Seq} = P_{IO\_Un} \times P_{IO\_Seq}$

P_{IO_Seq} : dictate the clock cycles for fully recharge

P_{IO_Seq} should be

For every parameter in a given algorithm

Required Multipliers = Π P_{comp_PE}

$P_{Comp\_Un} \times P_{Comp\_Seq} = P_{IO\_Un} \times P_{IO\_Seq}$

- P_{IO_Seq} : clock cycles for fully recharge
- P_{Comp_Seq} :
-

Extracting from Algorithm

Algorithm 1 CONV layer: simple seven nested loops.

```
for b = 0 until B do
  for k = 0 until K do
    for c = 0 until C do
      for y = 0 until Y do
        for x = 0 until X do
          for f_y = 0 until F_Y do
            for f_x = 0 until F_X do
              O[b][k][x][y] += I[b][c][x+f_x][y+f_y]
                             ×W[k][c][f_x][f_y]
```

Index	I	W	O	Mult
B	n	1	n	n
K	1	n	n	n
C	n	n	1	n
Y	n	1	n	n
X	n	1	n	n
Fy	n	n	n	1
Fx	n	n	n	1

Each index affect everyplace if it is used there. example:
“b” is used in “O” and “I”. Thus having n times unrolling “b” requires:

1) n times more output result signals
2) n times more input signals
3) as it is used for “I” in multiplication. It requires n times more Multipliers.

4) There is no effect on “W”

Streaming does not affect anything.
Streaming + Windowing reduces the IOs by saving them inside
Who can be windowed? The elements of input’s or weight’s indexes which includes more than one elements (each element has the chance to be choosen).
X, Fx, Y, Fy all are the candidates. Selecting one or more is acceptable. Since they are in “I”, they just affect the “I” requirements. The affects of selecting a variable to be windowed is the added shift registers and in adition to unrolled regisres which can be relaised by the algorithm and loop orders.

For every parameter in a given algorithm

Precision

Maybe 9x9 is better than 8x8

1) It is Xilinx style
2) Fit BRAM well and URAM in a good shape (URAM's width = 72 = 8*9. So both 8 and 9 are great.
3) Then supporting 27x18 is available (keep in mind that Acc width is pain full. 45 bits at least).
    1) A size: 3
    2) B size: 2 x reuse
    3) Acc size: reuse x (at least 45)

Reusement

Reusement factor for different layers (for input act)

1) Standard: KKC
2) DW: C
3) PW: C
4) FC: C
5) Mat-Mat Mult: C

Which PE arch to pick?

- Highest utilisation rate.
- Power will be managed by scheduling (Stanford paper): Power analysis on PE structure is generally non significant
- Scalability
- Flexible precision
- Reusement
- Less partial outputs
- Don’t trust on batch parallelism since it is not the case for embedded designs
- Limited number of multipliers

Ideas

1) Lop-based model → for different algorith → ASIC PE

2) Bench marking by Ideal PE archs(from 1) of current architectures (How well archs can implement the Ideal PEs)

3) How Synthetic arch can be generated?

Intel Architecture

Same as Xilinx
DSP is bigger, BRAM is bigger as well

DSP 28nm:

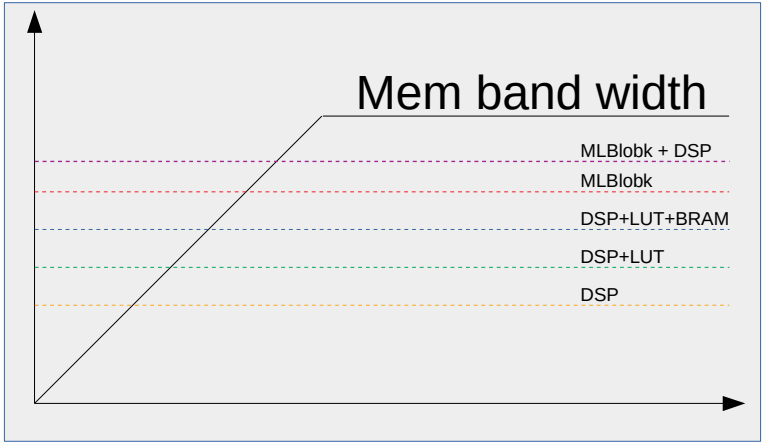
TABLE II: Area of enhanced DSP blocks and overhead of supporting different modes compared to the baseline.

DSP Block	Post-Synth. Area (μm^2)	Post-P&R Area (μm^2)	Area Ratio
Baseline DSP Block	8404	9875	1.00
Add 9×9 Mult.	**8368**	**10320**	**1.04**
Add 9×9 MAC	**8810**	**10384**	**1.05**
Add 4×4 Mult.(1)-max reuse	9571	–	–
Add 4×4 Mult.(2)-min reuse	**9104**	**10752**	**1.09**
Add 4×4 Mult.(3)-mid reuse	8909	11651	1.18
Add 4×4 MAC using $C2$	9543	11887	1.20
Add 4×4 MAC using $C5$	**9389**	**11108**	**1.12**

	intel® Agilex™ F-SerieS							intel® Agilex™ i-SerieS	
M20K	1900	2844	3792	5568	7110	11616	13272	11616	13272
DSP18x19	2300	3280	4592	8000	9020	12500	17056	12500	17056
DSP27x27	1150	1640	2296	4000	4510	6250	8528	6250	8528
ratio DSP/M20K	1.21	1.15	1.21	1.44	1.27	1.08	1.29	1.08	1.29
	0.61	0.58	0.61	0.72	0.63	0.54	0.64	0.54	0.64

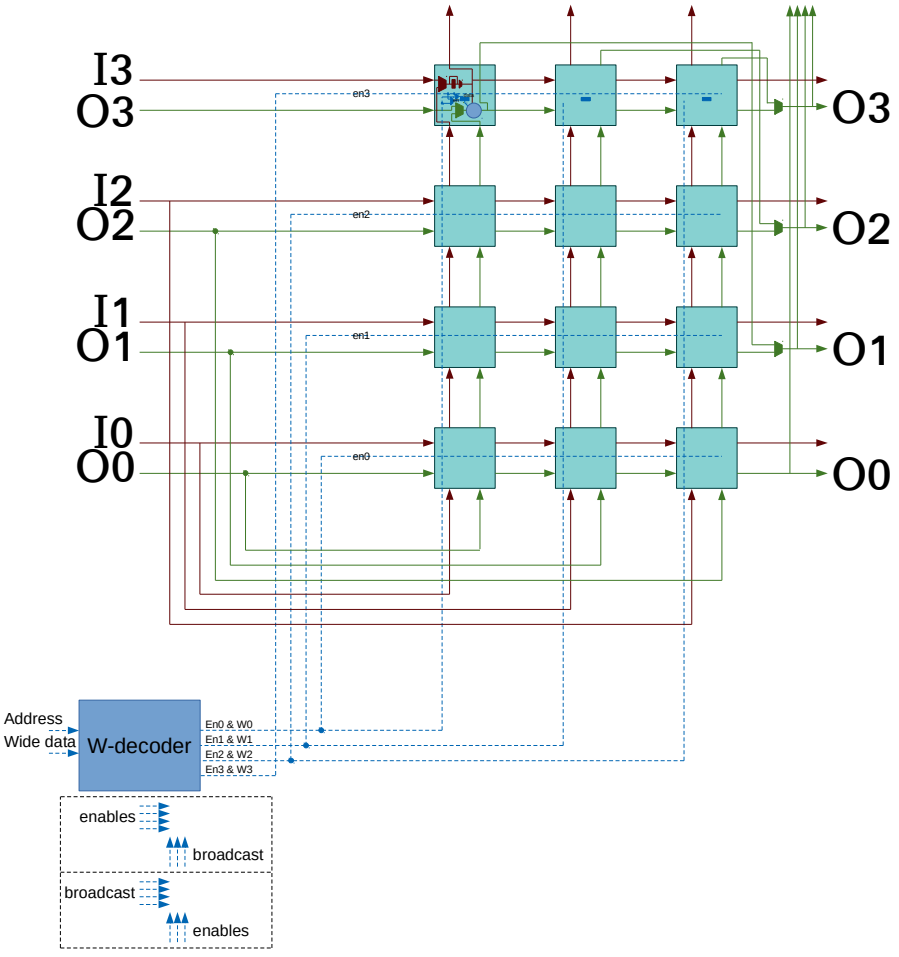
BRAM18	4033
DSP27x18	9024
ratio DSP/M20K	2.24

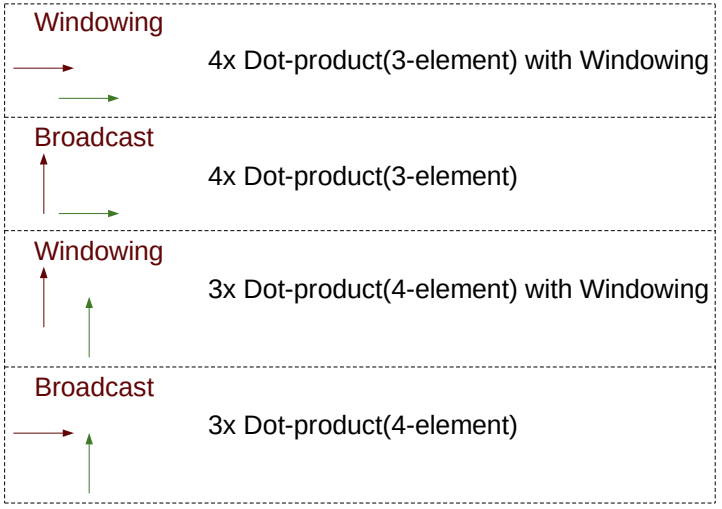
Roofline model

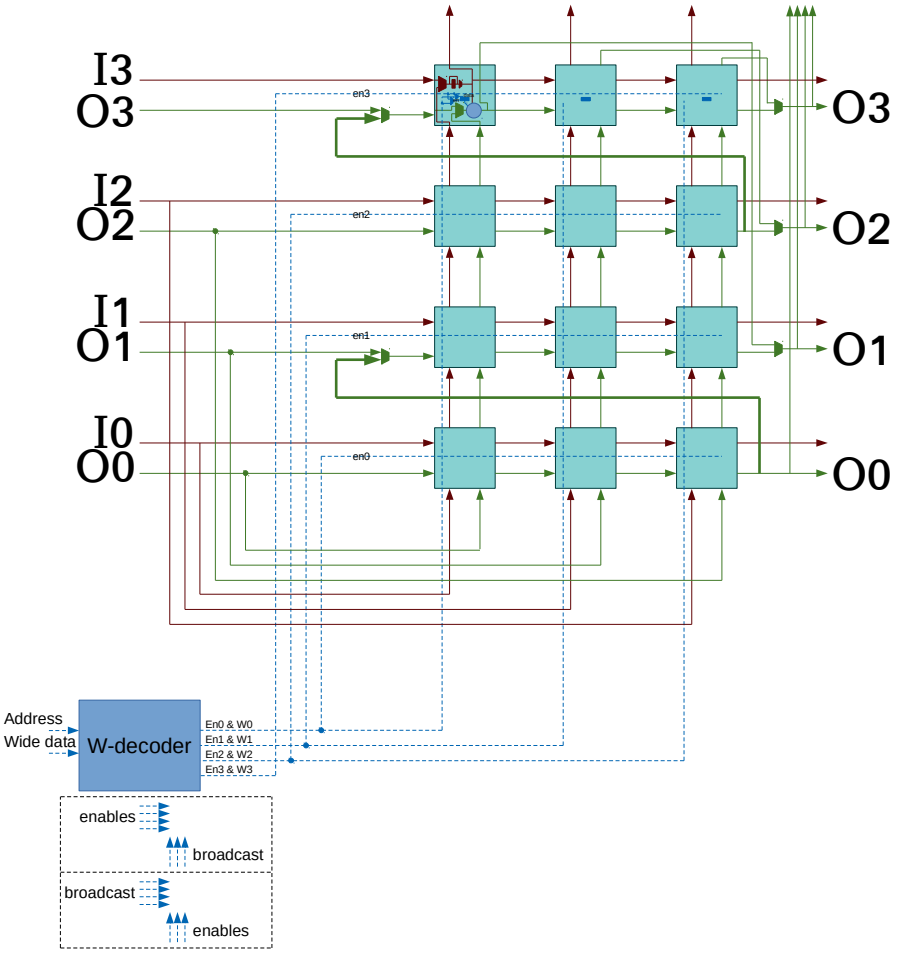


Generalised Architecture

- Architecture input
  - Array size: 3x4
  - Selected configurations (after filtering)
  -


I3
O3
I2
O2
I1
O1
I0
O0
en3
en2
en1
en0
Address
Wide data
W-decoder
En0 & W0
En1 & W1
En2 & W2
En3 & W3
enables
broadcast
broadcast
enables


Windowing
4x Dot-product(3-element) with Windowing
Broadcast
4x Dot-product(3-element)
Windowing
3x Dot-product(4-element) with Windowing
Broadcast
3x Dot-product(4-element)



Windowing

4x Dot-product(3-element) with Windowing

Broadcast

4x Dot-product(3-element)

Windowing

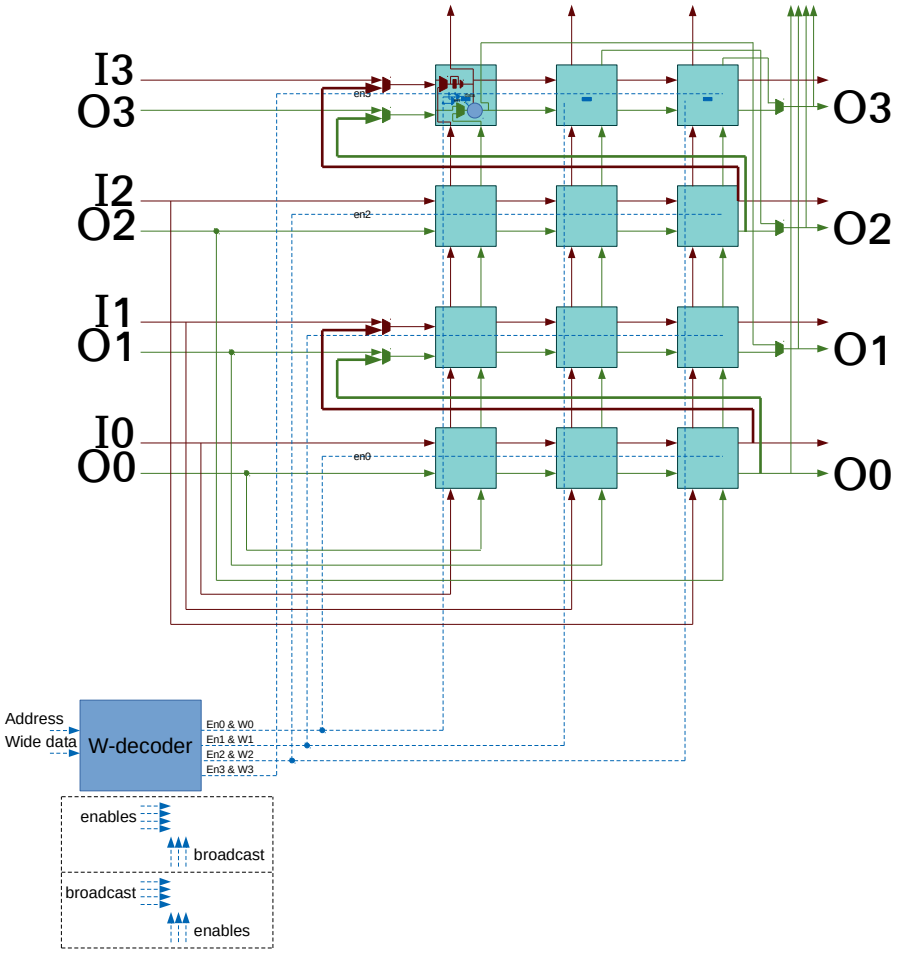
3x Dot-product(4-element) with Windowing

Broadcast

3x Dot-product(4-element)

Windowing

2x Dot-product(two 3-element-windowed)



Windowing

4x Dot-product(3-element) with Windowing

Broadcast

4x Dot-product(3-element)

Windowing

3x Dot-product(4-element) with Windowing

Broadcast

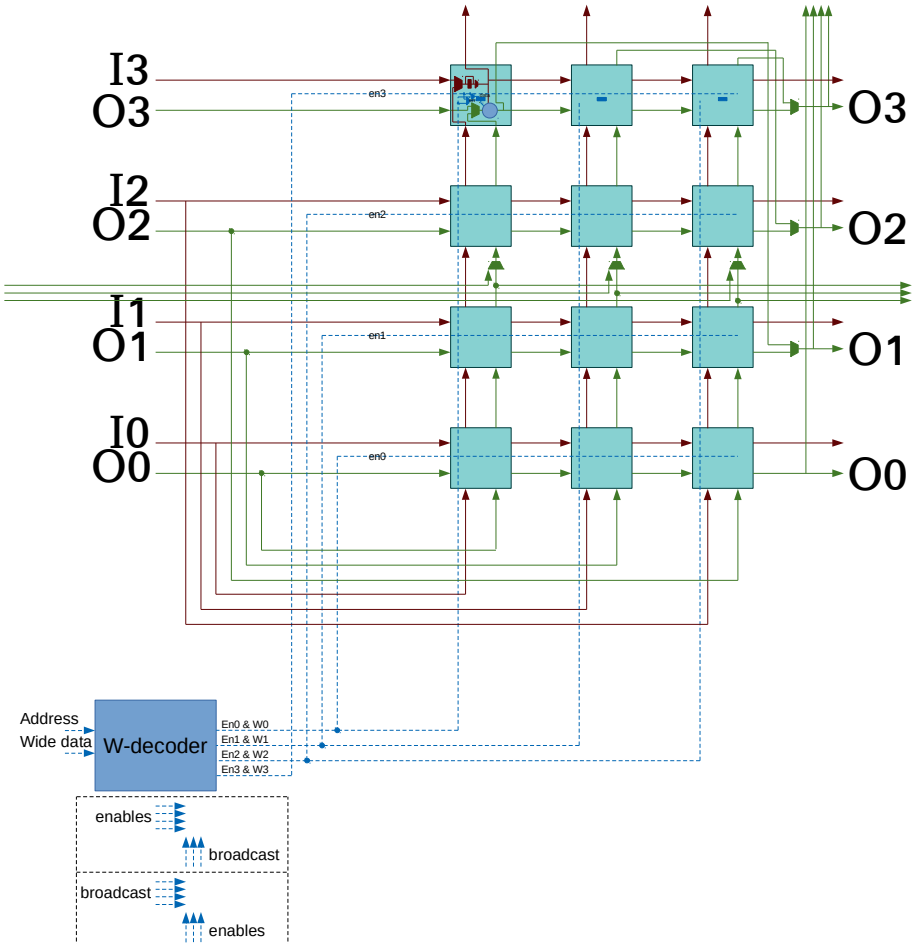
3x Dot-product(4-element)

Windowing

2x Dot-product(two-channel 3-element-windowed)

Windowing

2x Dot-product(6-element-windowed)



Windowing

4x Dot-product(3-element) with Windowing

Broadcast

4x Dot-product(3-element)

Windowing

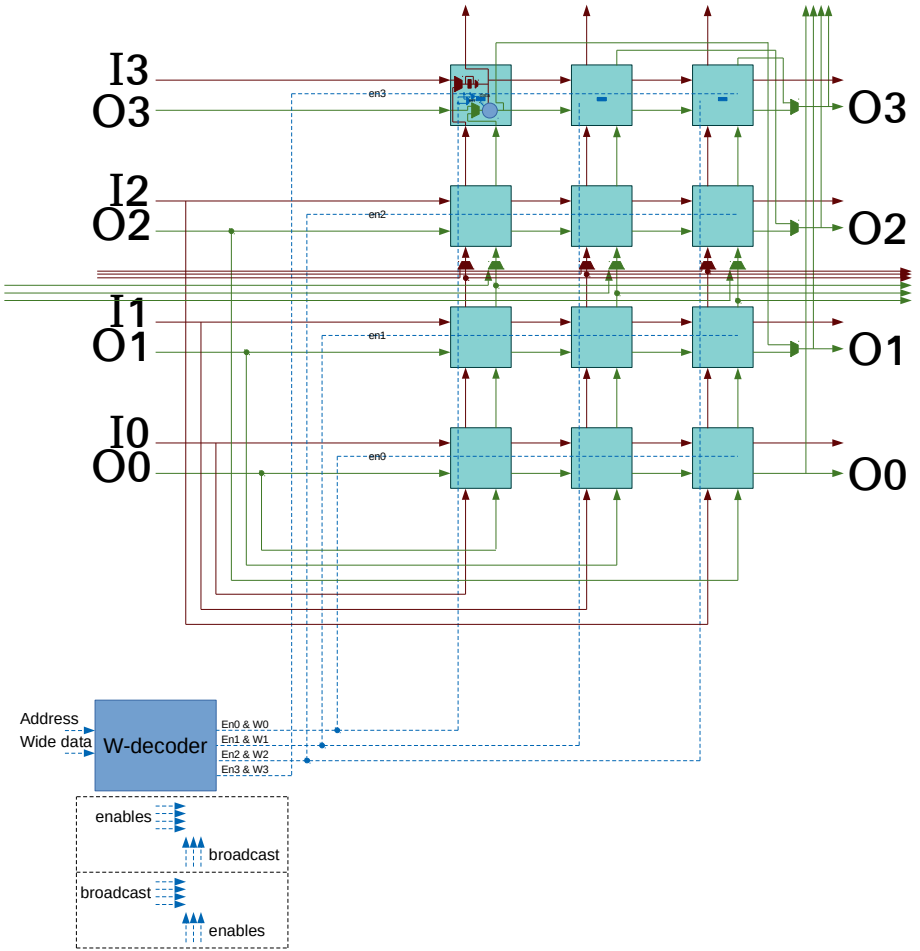
3x Dot-product(4-element) with Windowing

Broadcast

3x Dot-product(4-element)

Broadcast

6x Dot-product(2-element)



Windowing

4x Dot-product(3-element) with Windowing

Broadcast

4x Dot-product(3-element)

Windowing

3x Dot-product(4-element) with Windowing

Broadcast

3x Dot-product(4-element)

Windowing

6x Dot-product(2-element) with Windowing

Broadcast

6x Dot-product(2-element)