ESA IP core extensions for LEON2FT

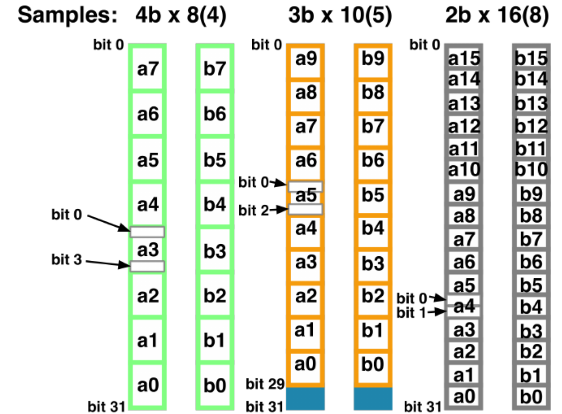
A new FPU with configurable (half/full/double) precision
SWAR (SIMD Within A Register) instruction extensions

TEC-ED & TEC-SW Final Presentation Days

2020-05-13

ESA UNCLASSIFIED - For Official Use

ESA IP core extensions for LEON2FT



- *Budget:* 156363 EUR (GSTP) + CCN 10000 EUR
- *Duration:* 2.5 years including CCN
- *Prime:* daiteq s.r.o., Prague, Czech Republic (CZ)
- Extensions will be available with the LEON2-FT ESA IP core
- *Main Objectives:*
  1. Develop a new versatile FPU for LEON2-FT;
  2. Specify and implement custom instructions for LEON2-FT;
     - SWAR: Multiple simultaneous operations of reduced bit-width
     - Apply new instructions on a GNSS SW receiver tracking loop
  3. CCN work: Port LEON2-FT and extensions to BRAVE-Medium
- Future work: evaluation of and extensions for the NOEL-V RISC-V IP core

ESA IP Core Extensions for LEON2: daiFPU and SWAR

ESA contract 4000122242/17/NL/LF

Presenter Martin Daněk
daiteq s.r.o.

TEC-ED & TEC-SW Final Presentation Days
May 13, 2020

Company overview

Based in Prague, the Czech Republic
Founded in 2013

Focus on

- Embedded systems, HW and low-level SW design
- FPGA acceleration for image processing and AI
- Processor architecture
  - Floating-point processing
  - Customized configurable datapaths
  - Dataflow processing, microthreading
- Independent technical reviews

Background: 15+ years of experience in digital design, FPGA technology, processor architecture, algorithm design, control.

Agenda

- Overview of the activity
- Application domain: GNSS
- Performance evaluation: tracking loop
- LEON2FT extensions: daiFPU, SWAR = SIMD-within-a-register
- Resource requirements: daiFPU, SWAR
- Performance evaluation: Whetstone
- Software support: binutils, llvm, SoftFloat
- Experience/Summary

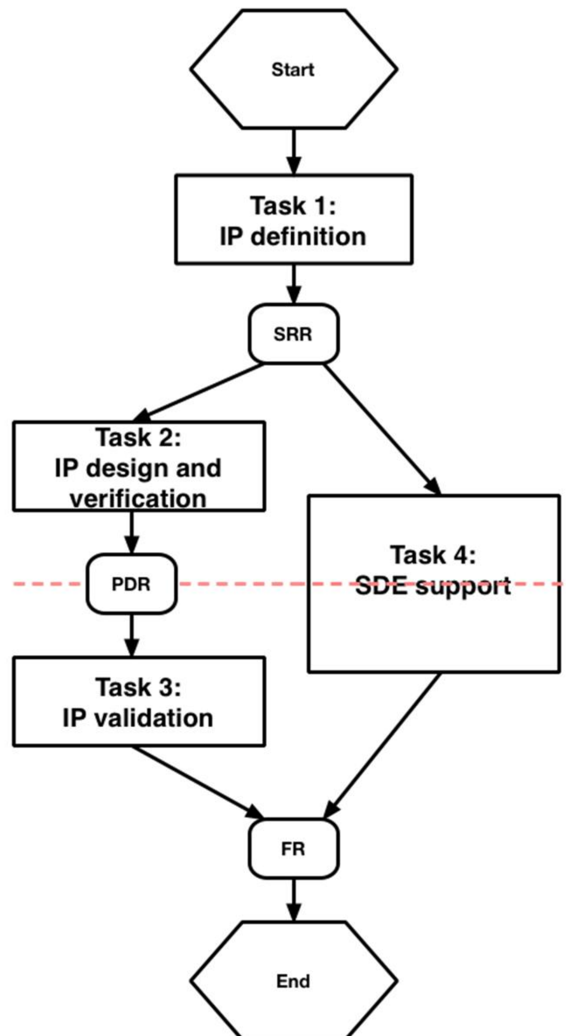
 Copyright (c) 2020 daiteq s.r.o.

Overview of the activity

Objectives:

- **Custom instruction set extensions**: main focus on **GNSS receivers**, but support also generic interface to enable users to add their own instructions. Consider **eFPGA** instead of hard-coded instructions.
- **FPU**: design a **highly configurable** unit, allow users to enable/disable individual instructions to trade-off accuracy, resolution, speed, area. Design/extend existing **llvm/binutils to support all FPU configurations**.
- **FPU survey**: compare performance and complexity of **different existing FPUs**, also including the influence of different compilers.
- **Validation**: VHDL simulation and FPGA prototyping.

Tasks



Reports:

- D01 – ISE specification
- D02 – FPU specification
- D03 – FPU survey report
- D04 – ISE design document
- D05 – FPU design document
- D06 – SDE design report and user manual
- Benchmarking framework document
- Tracking loop README and analysis
- CCN-1 – NG-Medium techmap
- LEON daiFPU/SWAR user manual

SW:

- leon2ft w/ daiFPU w/ SWAR
- leon2ft techmaps: XC6S, XC7V, MPF, NGM
- binutils w/ new FP and SWAR opcodes
- llvm & SoftFloat for daiFPU and SWAR
- Reference tracking loop w/ SWAR
- Benchmarking framework
- Test sets

Application domain

Domain: SDR for GNSS in LEON2
The main focus is on the tracking loop.

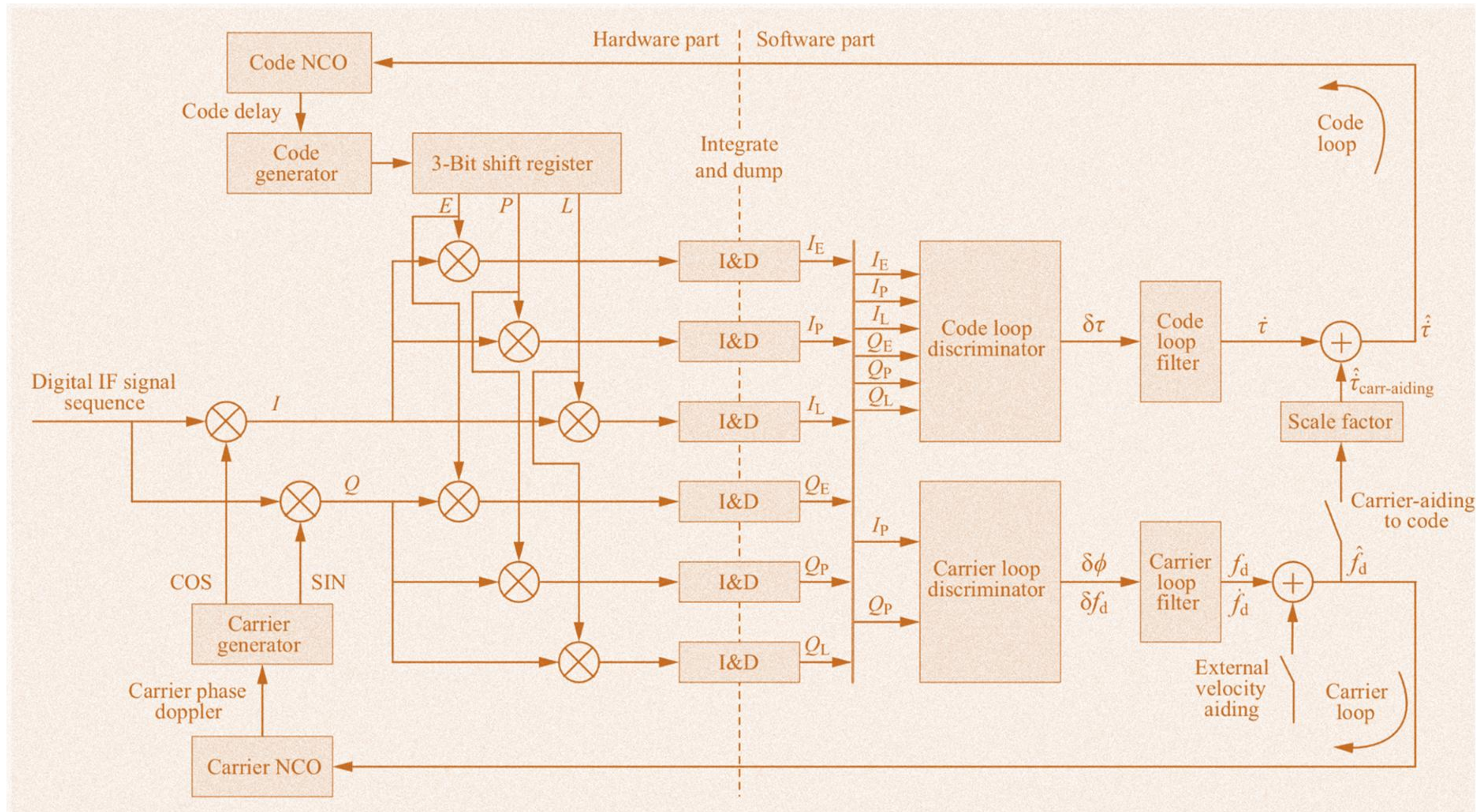
- Precise orbit determination with Galileo or GPS without customized ASIC solutions

Requirements: **low cost, high availability**

- Implementation targets: FPGA, ASIC
- FPU+SWAR: Moderate increase in resource requirements
- SWAR: Generic design, help users implement custom ISE for LEON (both VHDL and llvm/binutils)
- SWAR: Allow for implementation in eFPGAs (ASIC target) integrated in the LEON2FT integer pipeline

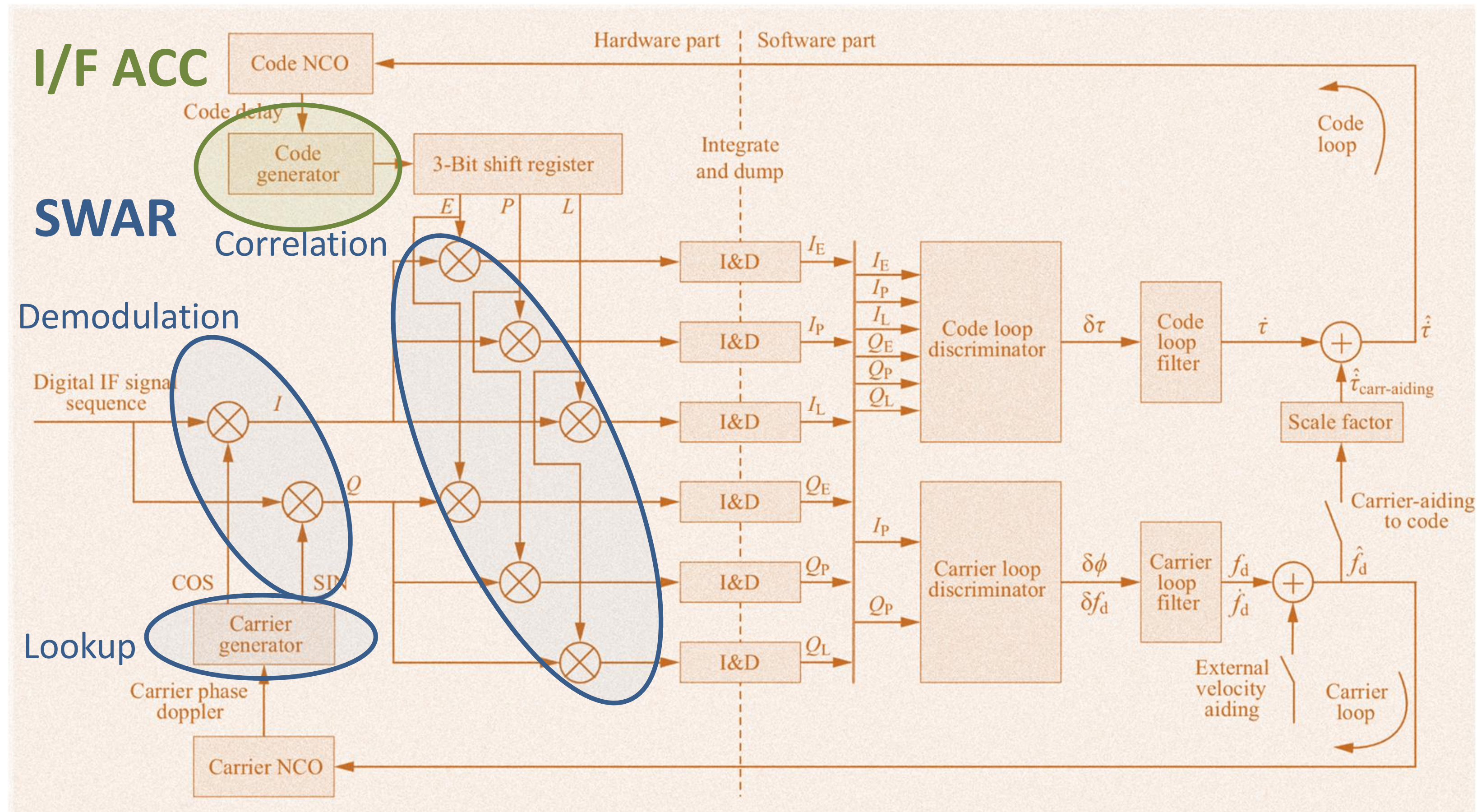
daiteq

Case study: GNSS tracking loop



Source: Springer Handbook on GNSS

Case study: GNSS tracking loop



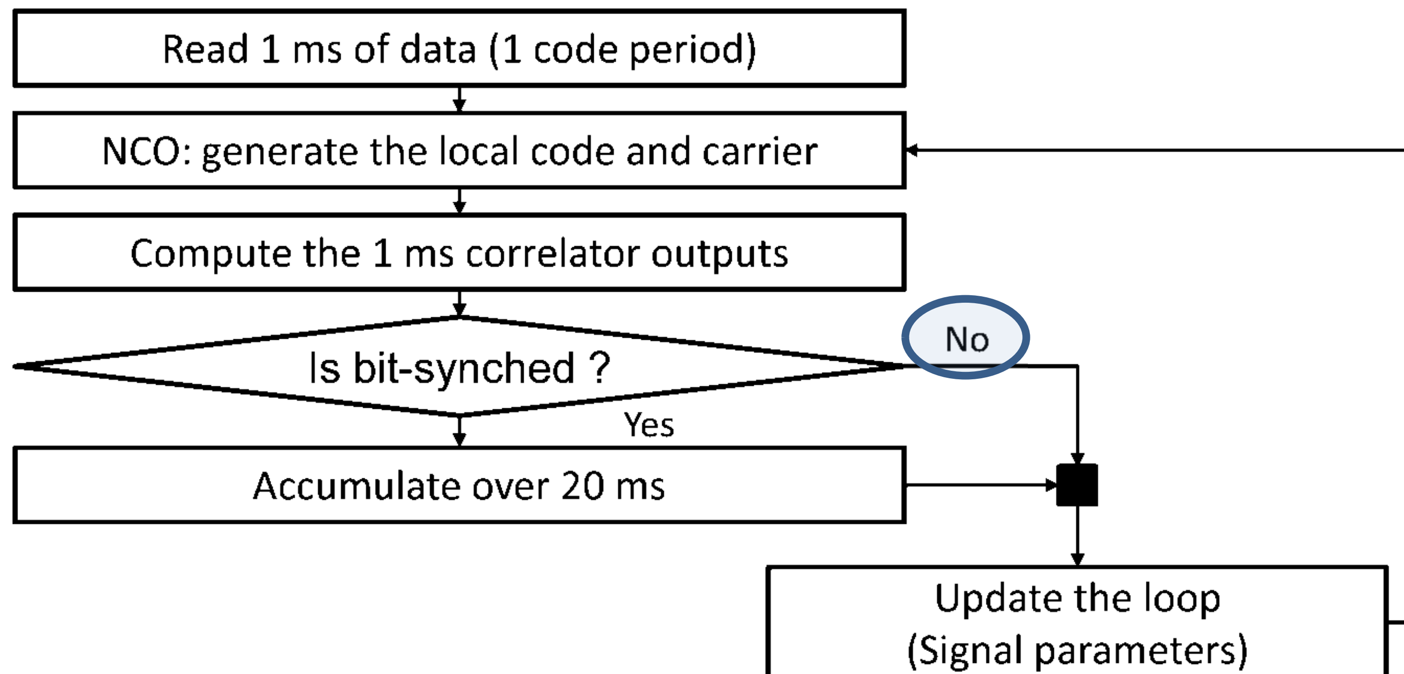
Source: Springer Handbook on GNSS

Structure of the code

Tracking loop code adopted from Borre et al.: *A Software-Defined GPS and Galileo Receiver. A Single-Frequency Approach*

Initialization

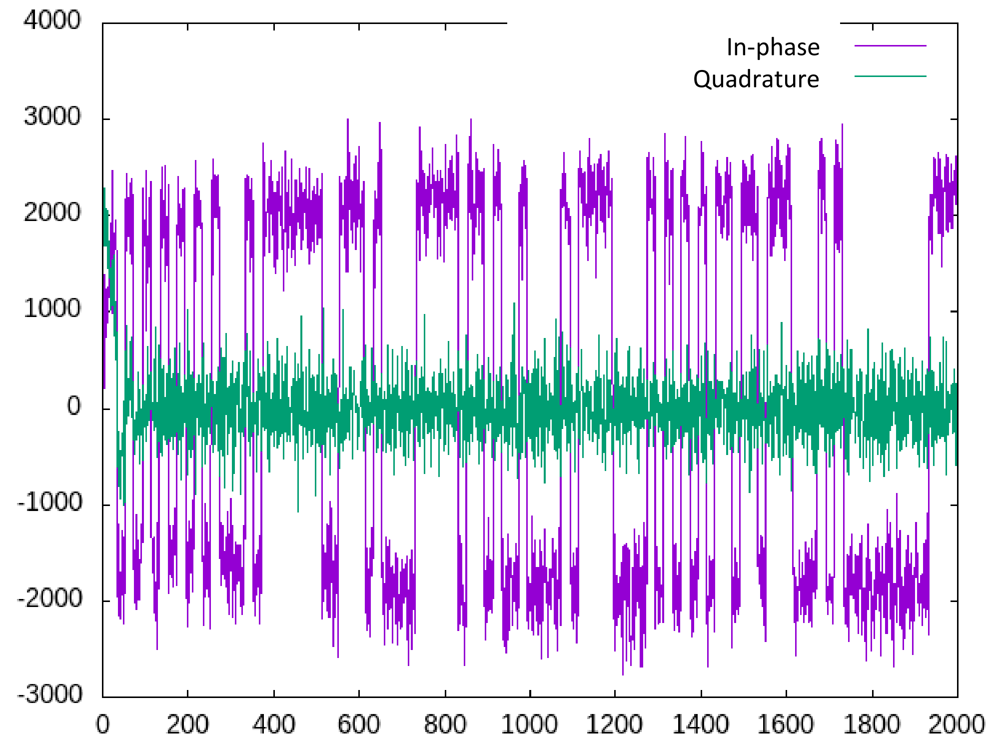
Main processing loop:
at each epoch 1 ms of data is read and processed = 1 I/Q pair



Sample I/Q decoding

GNSS dataset provided by ESA

Fs=25MHz, IF=4MHz
2x 16b complex signal
= 2 samples in one 32b word



The figure shows 2 seconds of navigation data.
1 point is computed from approx. 25000 complex samples (50000 values)
1 sample = 2x 16b words (w/o quantization) – but 2x 32b processing
1 sample = 2x 2b words (w/ quantization)

Execution platforms

Start of the activity: leon2ft_2015.3_nomeiko

- No FPU
- SPARCv8 integer arithmetic only
- **25MHz** processor clock

End of the activity: leon2ft_2020.5_daifpu_swar

- Blocking (serial) FPU
- SIMD-within-a-register integer arithmetic
- 25MHz processor clock

 Copyright (c) 2020 daiteq s.r.o.

Tracking loop profiling - start - execution time

Computation of 1 ms of data	Blksize – time to compute [s]	One value – time to compute [s]	One value – Exec cycles [1]
Signal input (FIFO)	0.001653400	6.6136e-8	1.65
Signal unpacking	0.06228932	2.49157e-6	62.29
One spreading code	1.40639172	5.62557e-5	1406.40
Sampling times	3.26883952	1.30754e-4	3268.85
All sine waves	6.488456840	2.59538e-4	6488.45
One sine wave point		1.29769e-4	3244.23
All demodulation	0.05603316	2.24133e-6	56.03
One signal value		1.12066e-6	28.02
All correlations	0.10813604	4.32544e-6	108.14
One correlation pt		7.20907e-7	18.03
TOTAL	**14.86242812**		**219’106’131**

Code (and processor) evolution

1. Original code in C (octave-equivalent), w/o FPU
   Stages communicate via buffers
2. +Samples quantized to 2b values (input, carrier)
3. +Integer argument for PRN code expansion and carrier generation, no SWAR, buffers
4. +SWAR for demodulation
5. +SWAR for correlation
6. +SWAR for sine/cosine lookup
7. +fused carrier generation and demodulation
8. +fused correlation
9. +just one PRN code expanded for all E,P,L
10. +HW support for PRN code expansion
11. +downsampling 1:3
12. and 1:10

----

13. Like 10, but with 4-bit samples

Tracking loop profiling – end - Execution time after each optimization step

Time to process 1ms of samples

2 – 2b quantization

5 – SWAR for demod and correl

6 – SWAR for sin/cos lookup, demod, correl

7 – fused carrier gen and demod

8 – fused carrier gen, demod, correl, no buffer for input samples

9 – just one PRN code expanded

10 – HW code expansion

11 – separate PRN codes, downsampling 1:3

12 – downsampling 1:10

13 – like 10 but 4-bit samples

Config /Iter	FIFO [us]	Codes [us]	Carrier [us]	Demodulation [us]	Correlation [us]	Total [us]
A - Original reference code, 2-bit values, SoftFloat						
2 / 0	63‘942	4‘874‘978	9‘624‘254	56‘033	108‘136	14‘530‘587
2 / 1	63‘943	4‘874‘998	9‘756‘074	56‘033	108‘136	14‘862‘450
B - like A plus SWAR instructions and daiFPU						
5 / 0	2‘343	82‘868	105‘190	9‘186	5‘771	208‘005
5 / 1	2‘125	82‘849	105‘185	9‘190	6‘077	205‘427
6 / 0	2‘430	82‘689	20‘601	9‘182	6‘204	121‘286
6 / 1	2‘125	82‘935	20‘595	9‘187	6‘074	120‘915
C - like B plus fused steps						
7 / 0	2‘413	83‘874		20‘753	5‘812	113‘396
7 / 1	2‘207	83‘844		20‘753	5‘809	112‘914
8 / 0		83‘873		20‘129	6‘116	110‘118
8 / 1		84‘847		20‘131	5‘943	109‘920
D - like C plus just one expanded code						
9 / 0		27‘772		18‘245	4‘366	50‘802
9 / 1		27‘751		18‘234	4‘366	50‘625
10 / 0		15‘286		18‘300	4‘501	38‘502
10 / 1		15‘269		18‘286	4‘502	38‘451
E - Like C Config 7 plus downsampling						
11 / 0	2‘326	80‘369		52‘854	1‘644	110‘039
11 / 1	2‘208	80‘252		52‘334	1‘540	109‘630
12 / 0	2‘414	72‘793		23‘645	591	99‘620
12 / 1	2‘207	72‘860		23‘648	455	99‘468
F - like D Config 10, but with 4-bit values						
13 / 0		15‘554		24‘458	4‘823	45‘250
13 / 1		15‘536		24‘454	4‘820	45‘089

Speedup through code optimization, FPU and SWAR

Start of the activity:

- 1 ms real-time = 14.87 s @ 25MHz
- LEON2 ~15000x slower than real signal

End of the activity:

- 1ms real-time = 38.45 ms @ 25MHz
- LEON2 ~39x slower than real signal
- But:
  - 9.6 ms @ 100MHz
  - 0.96 ms @ 100MHz w/ downsampling 10:1

LEON2 - where can we gain performance?

- Improve FP performance → HW FPU instead of SoftFloat
- Improve IPC → packed floating-point formats, SWAR integer formats, other HW acceleration
  - Customized instruction extensions – SWAR:
    - Arithmetic instructions, e.g. correlation and demodulation
    - Table lookup, e.g. sine and cosine
  - Customized interfaces, e.g. support for PRN code expansion

daiFPU – family of daiteq floating-point units

daiFPU configurations

Table 1: Pre-configured FPU precisions.

Abbreviation	Precision [b]	Partitioning
DP	53	(1,11,52)
SP	24	(1,8,23)
HP	11	(1,5,10)
PHP	11	((1,5,10),(1,5,10))
PSP	24	((1,8,23),(1,8,23))

Table 9: LEON2 - extended configuration register *conf.fpu.core* values (*leon/device.vhd*).

fpu.core	Flavour	Format 1	Format 2
daifpu_hp	single	HP	N/A
daifpu_sp	single SP	SP	N/A
daifpu_dp	single DP	DP	N/A
daifpu_php	single packed	HP	PHP
daifpu_psp	single packed	SP	PSP
daifpu	Meiko	DP	SP
daifpu_dual_dpsp	dual	DP	SP
daifpu_dual_sphp	dual	SP	HP

Table 10: LEON2 - extended configuration register *conf.fpu.divsqrt* values (*leon/device.vhd*).

fpu.divsqrt	FPDIV	FPSQRT
none	N	N
divonly	Y	N
divsqrt	Y	Y

Table 17: daiFPU - generics introduced in Version 2. (T/F = true/false)

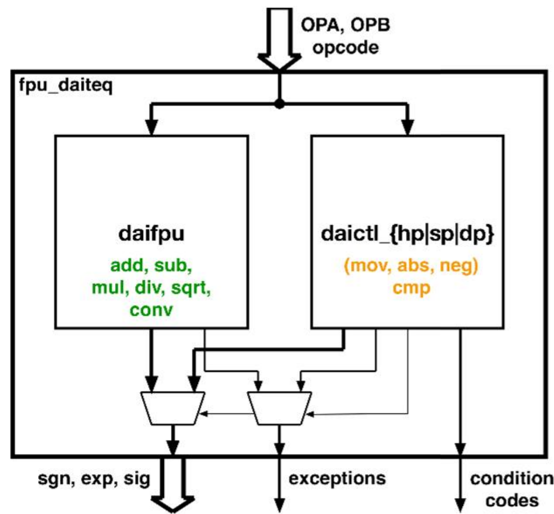
Generic	Range	Default	Comment
MULLAT	1-5	2	Select multiplier latency
MULRAW	T/F	T	Select multiplier implementation
LATENCY	1-2	2	Add pipeline stage in pre_norm
DELAY	T/F	T	Add pipeline stage in postnorm_mux
XTRASTAGES	0-1	1	Add additional stage in post_norm
SPECVALS	T/F	T	Instantiate *specvals* module
PIPELINED	T/F	T	Generate flags in post_norm

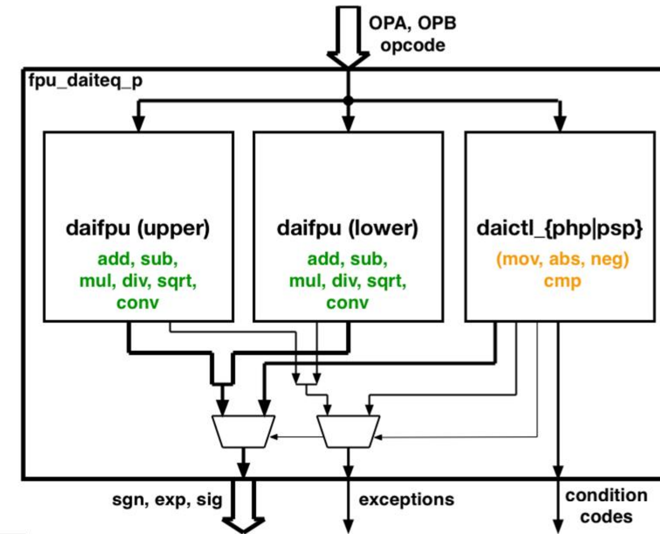
Table 3: Latencies of the floating-point operations, daiFPU w/ *specvals*, *C_PRENORM_LAT=2*, *XTRASTAGES=1*, as observed on the LEON2FT *fpui / fpuo* interface. When the daiFPU is configured with the *specvals* module, all arithmetic operations (*FADD, FSUB, FMUL, FDIV, FSQRT*) are finished in 6 clock cycles for those values at inputs that result in invalid operations.

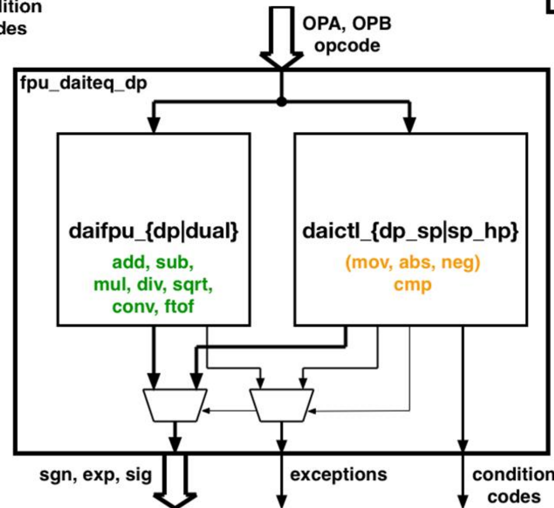
Operation	Module	Latency, MULLAT=2(3)
		Arithmetic operations
FADD	daifpu	6 or 9
FSUB	daifpu	6 or 9
FMUL	daifpu	6 or 10(11)
FDIVH	daifpu	6 or 22
FDIVS	daifpu	6 or 35
FDIVD	daifpu	6 or 64
FSQRTH	daifpu	6 or 22
FSQRTS	daifpu	6 or 35
FSQRTD	daifpu	6 or 64
		Conversions
ITOF	daifpu	9
FTOI	daifpu	5
FTOF	daifpu	9
FMOV	iu	1
FNEG	iu	1
FABS	iu	1
		Comparisons & unimp
FCMP	daictl	3
FCMPE	daictl	3
invalid	iu	1

 Copyright (c) 2020 daiteq s.r.o.

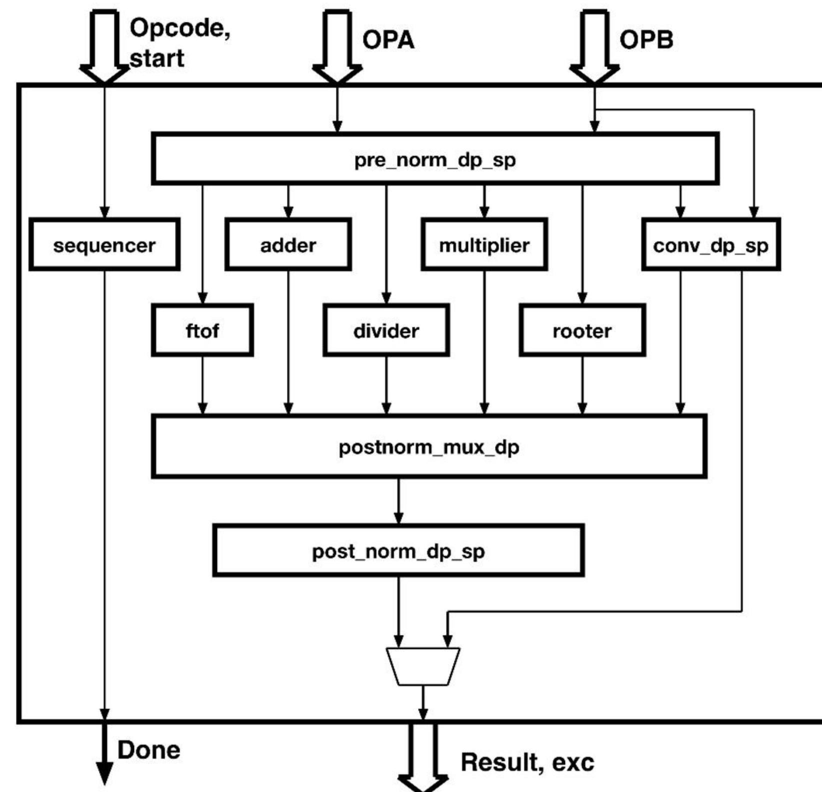
daiFPU flavours







daiFPU datapath



daiFPU: resource requirements

Table 18: daiFPU w/ AT697F-488284 - XC7V - FPU-only, resources used.

Flavour	Slices	Slice regs	LUTs	LUTRAM	DSP48E1
daifpu					
divsqrt	3665	3397	9489	385	15
divonly	3259	2918	8163	362	15
none	2946	2543	7039	279	15
daifpu-dual-dpsp					
divsqrt	3116	3402	9442	385	15
divonly	3006	2918	8092	362	15
none	2474	2528	6670	279	15
daifpu-dual-sphp					
divsqrt	1965	2195	5143	157	2
divonly	1853	1927	4499	157	2
none	1651	1713	3827	130	2
daifpu-dp					
divsqrt	2476	2581	6164	321	15
divonly	2162	2218	5227	296	15
none	1602	1911	4225	226	15
daifpu-sp					
divsqrt	1251	1503	3209	147	2
divonly	1016	1530	2778	109	2
none	946	1232	2283	105	2
daifpu-hp					
divsqrt	774	985	1791	74	1
divonly	655	906	1500	68	1
none	541	765	1346	57	1
daifpu-psp					
divsqrt	2812	3148	6598	283	4
divonly	2354	2713	5658	216	4
none	2047	2122	4645	220	4
daifpu-php					
divsqrt	1418	1851	3656	173	2
divonly	1270	1768	3050	127	2
none	1062	1362	2589	137	2

Table 21: daiFPU w/ AT697F-488284 - MPF300 - FPU only, resources used.

Flavour	Fabric 4LUT	Fabric DFF	uSRAM 1K	uSRAM 18K	Math (18x18)
daifpu					
divsqrt	14468	3935	0	0	12
divonly	13030	3412	0	0	12
none	11134	2868	0	0	12
daifpu-dual-dpsp					
divsqrt	14528	3941	0	0	12
divonly	12827	3409	0	0	12
none	10769	2816	0	0	12
daifpu-dual-sphp					
divsqrt	7345	2169	0	0	3
divonly	6539	1892	0	0	3
none	5651	1593	0	0	3
daifpu-dp					
divsqrt	9020	2917	0	0	12
divonly	7955	2592	0	0	12
none	6361	2095	0	0	12
daifpu-sp					
divsqrt	4466	1580	0	0	3
divonly	3959	1410	0	0	3
none	3264	1157	0	0	3
daifpu-hp					
divsqrt	2507	959	0	0	3
divonly	2241	864	0	0	3
none	1923	728	0	0	3
daifpu-psp					
divsqrt	9063	3147	0	0	6
divonly	8121	2807	0	0	6
none	6563	2301	0	0	6
daifpu-php					
divsqrt	5053	1859	0	0	6
divonly	4550	1669	0	0	6
none	3847	1397	0	0	6

daiFPU: XC7V (non-TMR): 541 – 3665 slices. MPF (non-TMR): 1924 – 14528 4LUTs.

LEON2FT: XC7V: non-TMR @ 100MHz, TMR @ 75MHz. MPF: non-TMR @ 100MHz.

Performance: fpu_daiteq_dp vs. Meiko

SP605 ddr

50MHz,	SP: 13.99 MWIPS,	DP: 12.22 MWIPS

VC707: ddr

50MHz,	SP: 14.74 MWIPS,	DP: 12.43 MWIPS
100MHz,	SP: 27.40 MWIPS,	DP: 23.30 MWIPS

AT697F sdram

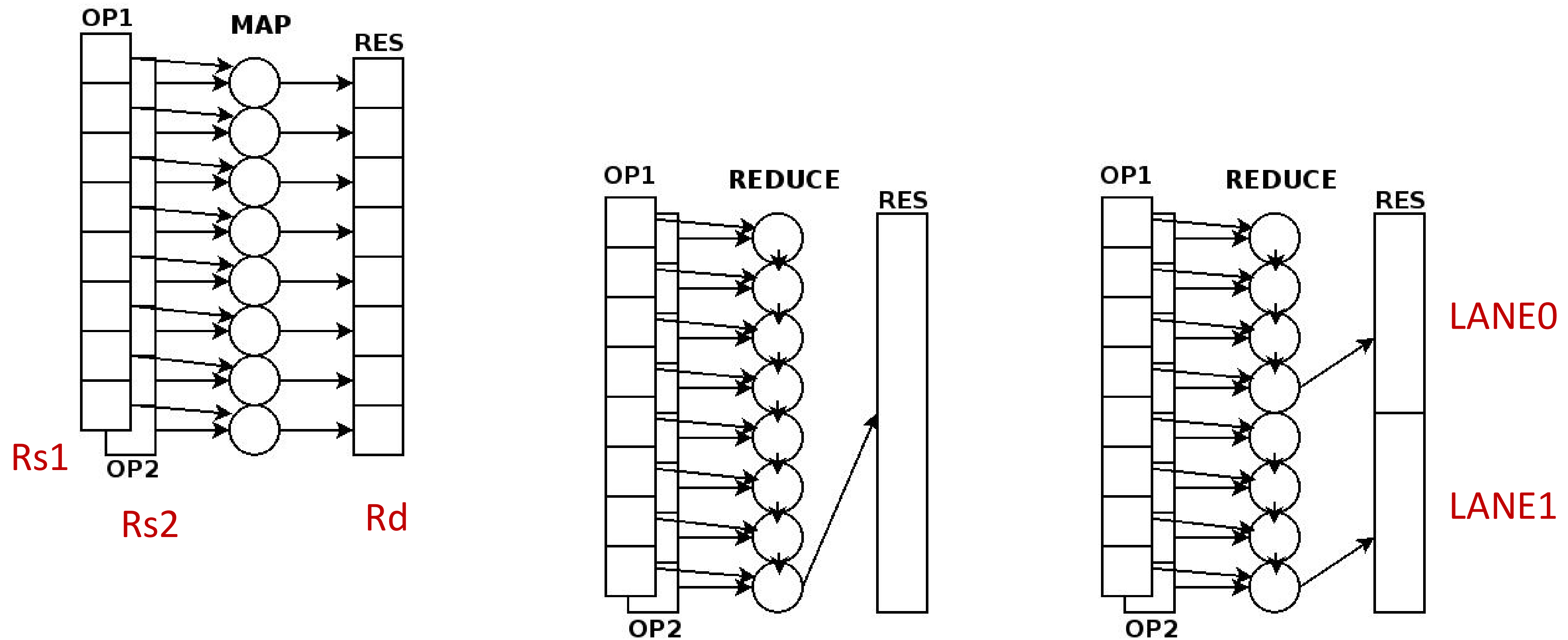
40MHz WS0:	SP: 11.73 MWIPS,	DP: 9.95 MWIPS
60MHz WS0:	SP: 17.65 MWIPS,	DP: 14.95 MWIPS

AT697F sram

40MHz WS1:	SP: 11.65 MWIPS,	DP: 9.65 MWIPS
60MHz WS1:	SP: 17.54 MWIPS,	DP: 14.51 MWIPS
100MHz WS2:	SP: 27.43 MWIPS,	DP: 22.32 MWIPS
100MHz WS3:	SP: 25.89 MWIPS,	DP: 21.62 MWIPS

SWAR – SIMD-within-a-register

SWAR: application view



SWAR: register partitioning

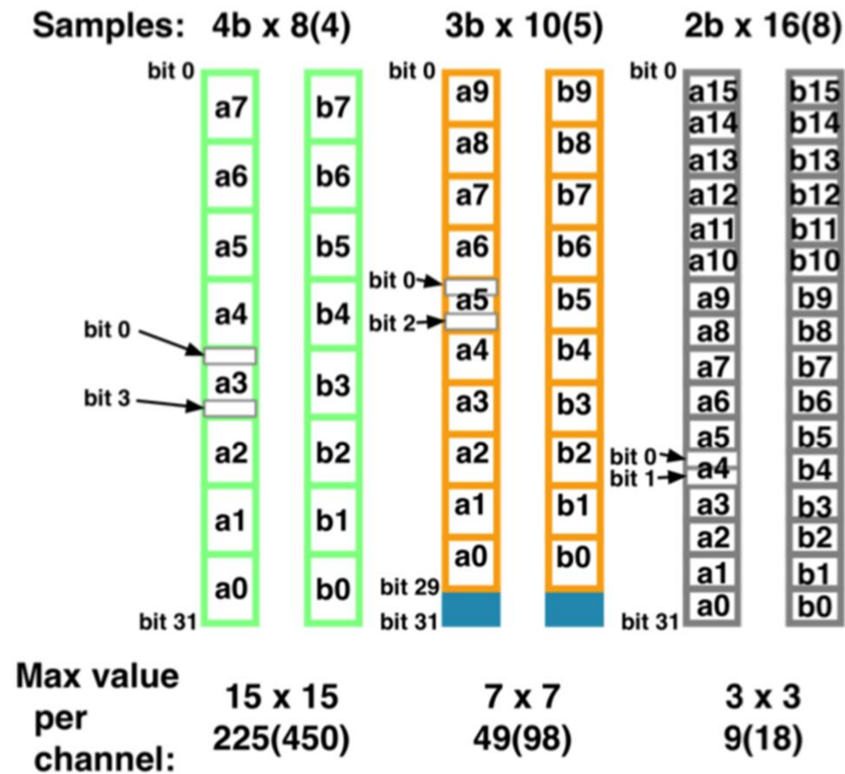
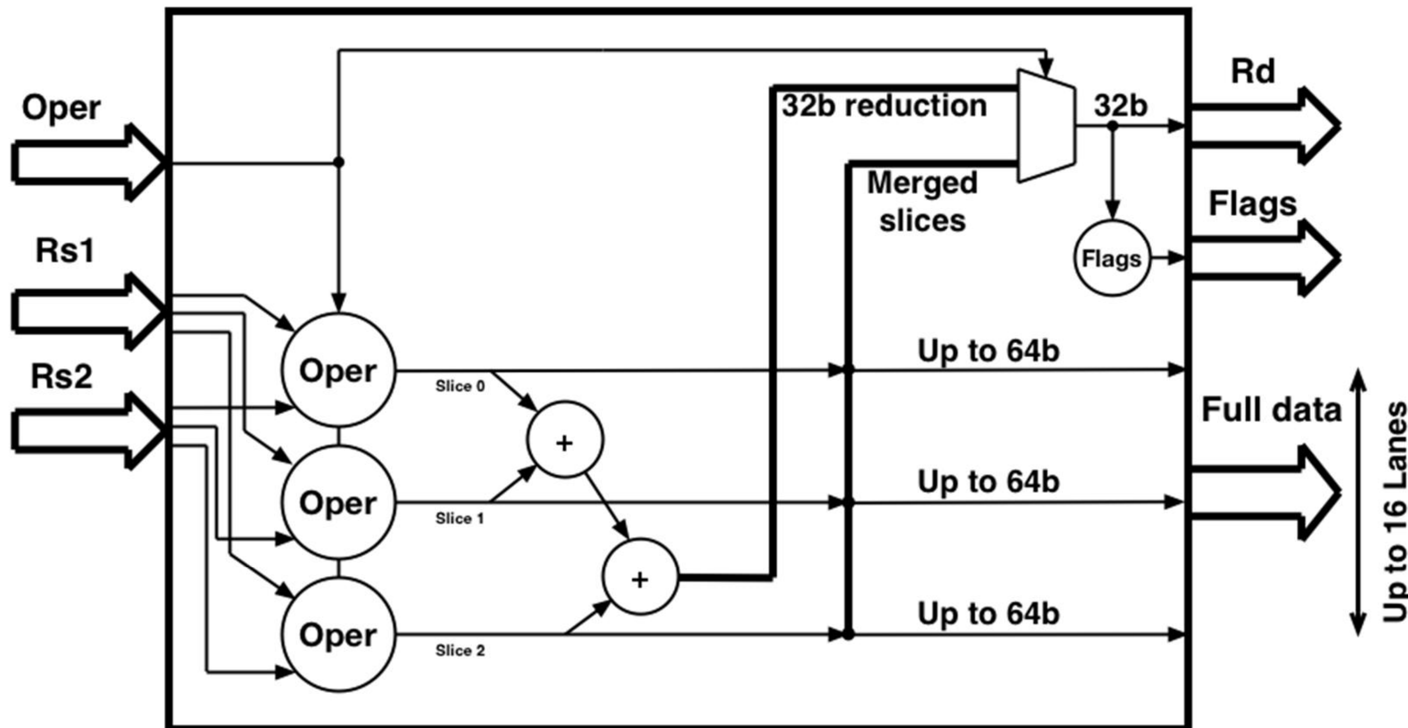


Figure 2: Vector multiplication - mapping of subwords in source registers.

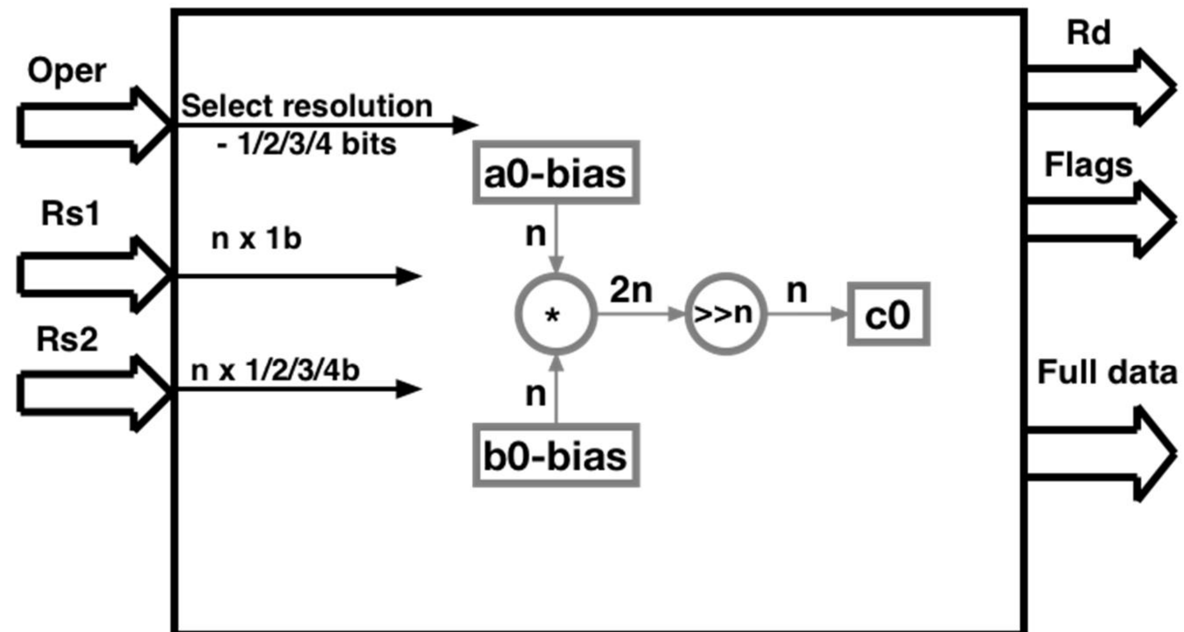
SWAR architecture: module interfaces

SWAR architecture: generic ALU

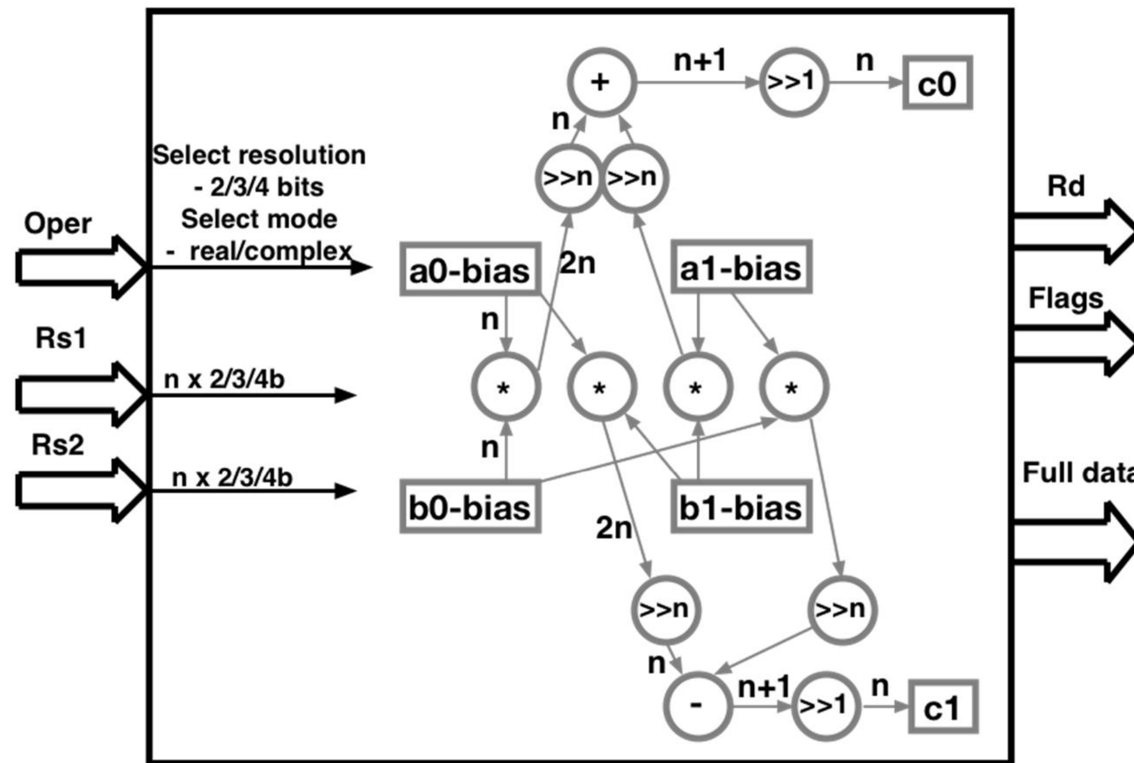


Oper: ADD, SUB, MUL

SWAR architecture: correlation

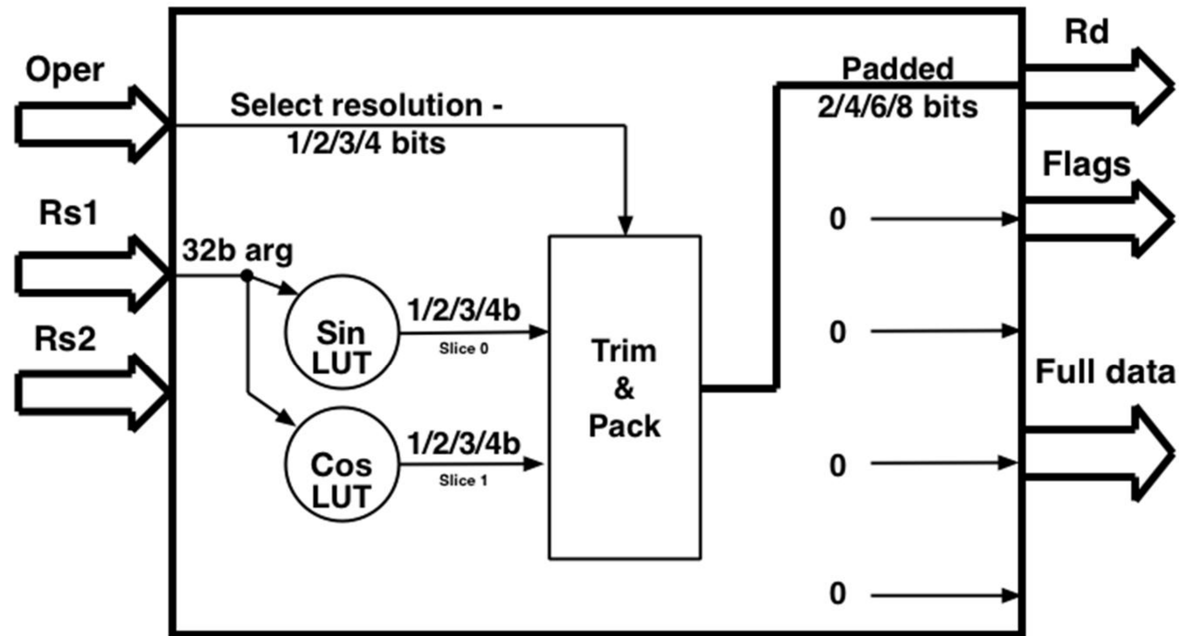


SWAR architecture: demodulation

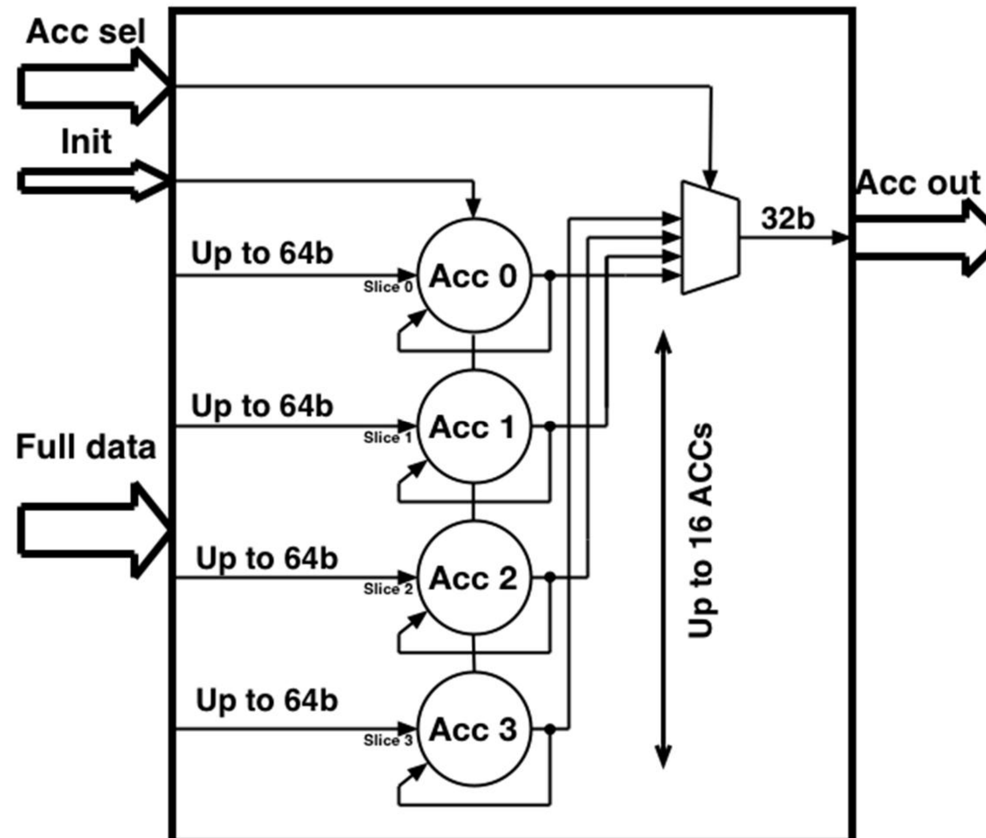


Only complex mode shown. Real mode not shown.

SWAR architecture: sine/cosine lookup



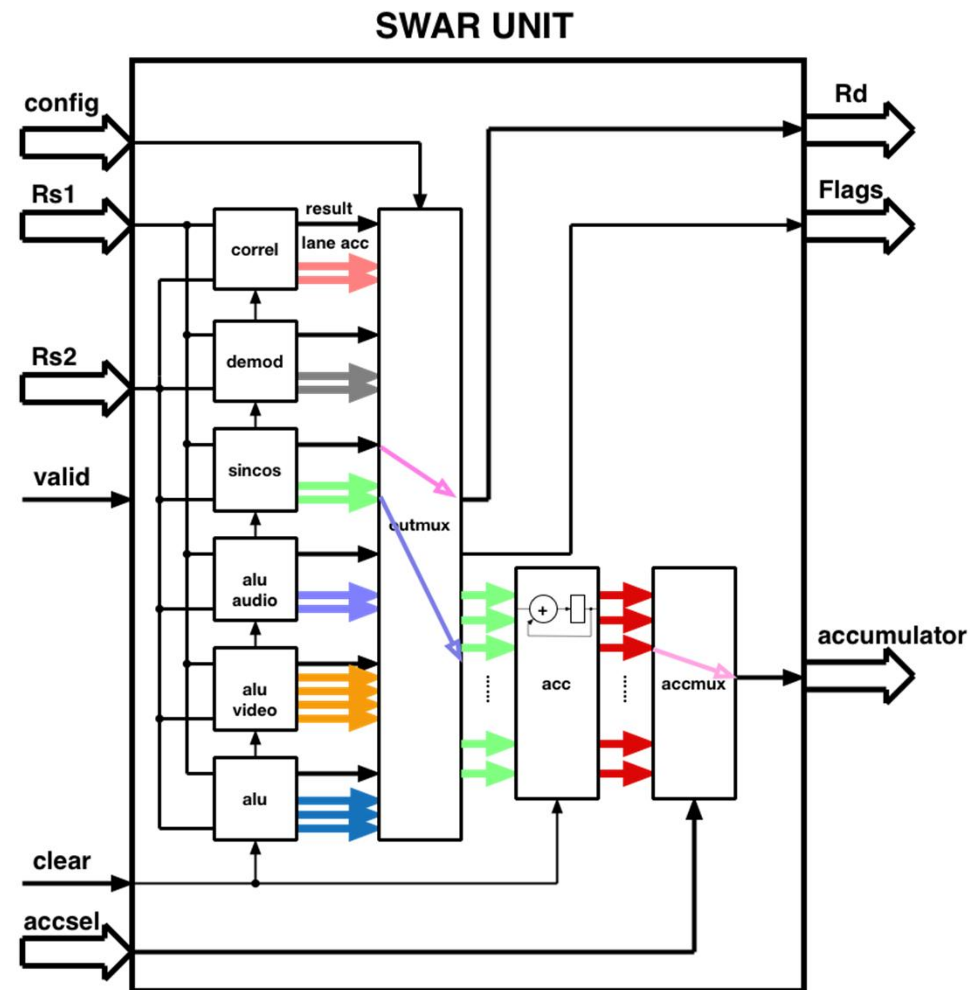
SWAR architecture: accumulator



SWAR architecture: SWAR unit

Targeted applications:

- **GNSS**
  16x 2b words
  10x 3b words
  8x 4b words
- **Audio**
  2x 16b words
- **Video/image**
  4x 8b words



SWAR operations and configuration

Table 1: SWAR operations (*%asr26*).

SWAR Selector	Code	Values	Comment
00000000	SWADD	2 or 4	Addition, 2x16b (audio) or 4x8b (video)
00001000	SWSUB	2 or 4	Subtraction, 2x16b (audio) or 4x8b (video)
00001100	SWMUL	2 or 4	Multiplication, 2x16b (audio) or 4x8b (video)
00000100	COR1b	32	1bx1b sum of products - correlation (signed=1) or bit masking
00000101	COR2b	16	1bx2b sum of products - correlation (signed=1) or bit masking
00000110	COR3b	10	1bx3b sum of products - correlation (signed=1) or bit masking
00000111	COR4b	8	1bx4b sum of products - correlation (signed=1) or bit masking
00001001	DEMR2b	16	2b real demodulation
00001010	DEMR3b	10	3b real demodulation
00001011	DEMR4b	8	4b real demodulation
00001101	DEMC2b	8	2b complex demodulation, results in (Re, Im) order
00001110	DEMC3b	5	3b complex demodulation, results in (Re, Im) order
00001111	DEMC4b	4	4b complex demodulation, results in (Re, Im) order
00000001	DEMC2bG	8	2b complex demodulation, results all Re, then all Im parts
00000010	DEMC3bG	5	3b complex demodulation, results all Re, then all In
00000011	DEMC4bG	4	4b complex demodulation, results all Re, then all In
00010000	SC1b	32	(1+1)x1b sine / cosine value
00100000	SC2b	16	(1+1)x2b sine / cosine value
00110000	SC3b	10	(1+1)x3b sine / cosine value
01000000	SC4b	8	(1+1)x4b sine / cosine value

Table 1: LEON2 - extended configuration register (*leon/device.vhd*).

Register	Module	Description
conf.iu.fifoen	code, fifo	Enable the sequential memories
conf.iu.swaren	swar_unit	Enable the SWAR unit
conf.swar.correl	swar_correl	Enable the SWAR correlation extensions
conf.swar.demod	swar_demod	Enable the SWAR demodulation extensions
conf.swar.sincos	swar_sincos	Enable the SWAR sine/cosine lookup
conf.swar.audio	swar_audio	Enable the SWAR audio extensions
conf.swar.video	swar_video	Enable the SWAR video extensions
conf.swar.alu	swar_alu	Enable the generic SWAR ALU
conf.swar.acc	swar_acc	Enable the SWAR lane accumulators
conf.swar.lanes		Set the number of SWAR lanes
conf.swar.swidth		Set the SWAR slice width
conf.swar.awidth		Set the SWAR lane accumulator width

Signed / unsigned
Saturated / non-saturated

SWAR: resource requirements

Table 3: SWAR configurations evaluated - supported sub-32b word operations.

Identifier	correl	demod	sincos	audio	video	ALU	ACC
ALL FUNCTIONS							
swarall	Y	Y	Y	Y	Y	3x10b	N
8b, 10b, 12b ALU w/ ACC							
swaralu	N	N	N	N	N	3x10b	3x14b
swaraudio	N	N	N	Y	N	N	2x20b
swarvideo	N	N	N	N	Y	N	4x12b
GNSS							
swargnss	ALL	ALL	ALL	N	N	N	N
swargnss-2b	2b	2b	2b	N	N	N	N
swargnss-3b	3b	3b	3b	N	N	N	N
swargnss-4b	4b	4b	4b	N	N	N	N

Audio = 2x 16b
Video = 4x 8b

Table 13: SWAR w/ AT697F-488284 w/ daiFPU - XC7V - SWAR unit, resources used.

Flavour	Slices	Slice regs	LUTs	LUTRAM	DSP48E1
swarall	929	215	2558	0	9
swaralu	137	94	333	0	3
swaraudio	83	92	287	0	2
swargnss	729	85	1799	0	0
swargnss-2b	223	83	695	0	0
swargnss-3b	297	78	755	0	0
swargnss-4b	359	82	895	0	0
swarvideo	165	74	367	0	4

Table 16: SWAR w/ AT697F-488284 w/ daiFPU - MPF300 - SWAR unit, resources used.

Flavour	Fabric 4LUT	Fabric DFF	uSRAM 1K	uSRAM 18K	Math (18x18)
swarall	2679	189	0	0	45
swaralu	716	90	0	0	3
swaraudio	471	88	0	0	2
swargnss	1500	82	0	0	36
swargnss-2b	673	78	0	0	0
swargnss-3b	696	78	0	0	20
swargnss-4b	751	82	0	0	16
swarvideo	559	46	0	0	4

Software Development Environment

SDE: new data types

Floating point:

- Half precision, including compile-time constants
- Packed floating-point types for single and half precision
- Data conversions

SWAR:

- User-defined packed integer types
- Data conversions

Sample code with the new half type

```
#include <stdint.h>
volatile half a,b,r;
volatile int i,j;
void test_half(void)
{
        r = a + b;
        r = a - b;
        r = a * b;
        r = a / b;
        a = (half)i;
        j = (int)r;
}
```

```
int main(void)
{
        volatile half cf = 1.234567H;
        volatile half df = 3.1415926H;
        volatile half ef = cf + df;
        test_half();
        return 0;
}
```

Sample code with the new packed half type

```
typedef half half2 __attribute__((ext_vector_type(2)));

half2 p = {3.1415h, 2.718h};
volatile half2 q,r;

void test(void)
{
        r = p + q;
        r = p - q;
        r = p * q;
        r = p ^ q;
        r = p / q;
}

int main(void)
{
        p.x = 1.234h;
        test();
        return p.y;
}
```

Definition of SWAR types – compatibility

```
/* handle swar types for systems without swar extension */
#if USE_SWAR
        typedef unsigned int subU4b  __attribute__((__subword(4)));
        typedef unsigned int sub8U4b  __attribute__((__subword__(4, 8)));
#else
        typedef unsigned int subU4b[1];
        typedef unsigned int sub8U4b[8];
#endif

#if USE_SWAR
        typedef unsigned int subU4b  __attribute__((__subword(4)));
        typedef unsigned int sub8U4b  __attribute__((__subword__(4, 8)));
#else
        typedef unsigned char subU4b[1];
        typedef unsigned char sub8U4b[8];
#endif
```

Definition of SWAR types – more examples

```
/* two ways how to define vector with 1 item (size = 2bit)  */
typedef unsigned int s1x2b __attribute__((subword(2)));
typedef unsigned int s1x2b_alter __attribute__((subword(2, 1)));

/* define type of swar vector of two-bit items in one U32 word  */
typedef unsigned int s16x2b __attribute__((subword(2, 16)));

/* define type of swar vector of one-bit items */
typedef unsigned int s16x1b __attribute__((subword(1, 16)));

/* define type of long swar vector (vector with 50 two-bits items organized in
 * unsigned int (U32) words - it occupies 4 U32 words  */
typedef unsigned int s50x2b __attribute__((subword(2, 50)));

/* define another swar type with one-bit items */
typedef unsigned int s50x1b __attribute__((subword(1, 50)));
```

Declaration of SWAR variables

```
/* vectors, each fits in one 32-bit word */
s1x2b sitem;
s16x2b svecA, svecB;
s16x1b svecC;

/* long vectors that are stored in several 32-bit words */
s50x2b lvecL, lvecM, lvecN;

/* reduction variable */
unsigned int rvec;
```

Sample code with SWAR operations

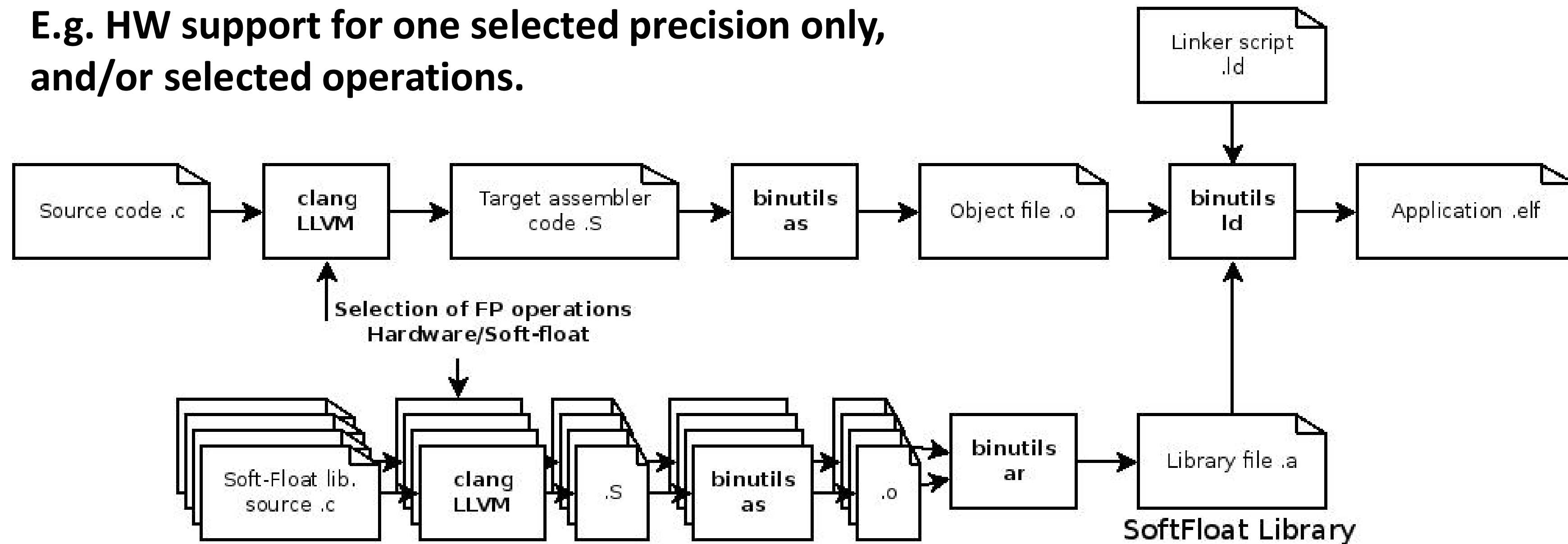
```
int main(void){
        /* set item in a vector */
        svecA[2] = 3;

        // svecA[3] = 4; /* compiler returns compilation error – value too big */

        /* direct operations - SWADD, SWSUB, SWMUL - variables have to contain
         * the same number of items and fit in one 32-bit word */
        //svecA = svecB + sitem; /* compiler returns compilation error - operands do
                                  * not have the save number of elements */

        svecA = svecB + svecC;
        svecA = svecB - svecC;
        svecA = svecB * svecC;  /* element-wise multiplication */
… (listing continues on the following slide)
```

Sample code with SWAR operations (cont'd)

```
    /* for-loop is used for long vectors and vectors with different lengths */
    unsigned int accum = 0;
    for (int i=0;i<50;i++)
        accum += lvecL[i];
    /* element-wise product - operands have the same width */
    for (int i=0;i<16;i++)
        svecA[i] = svecA[i]*svecB[i];
    /* map-reduce */
    rvec = 0;
    for (int j=0;j<50;j++)
        rvec += lvecM[j]*lvecN[j];
    /* get item from a vector */
    int o = svecA[5];
    return 0;
}
```

Compilation and linking: daiFPU and SWAR

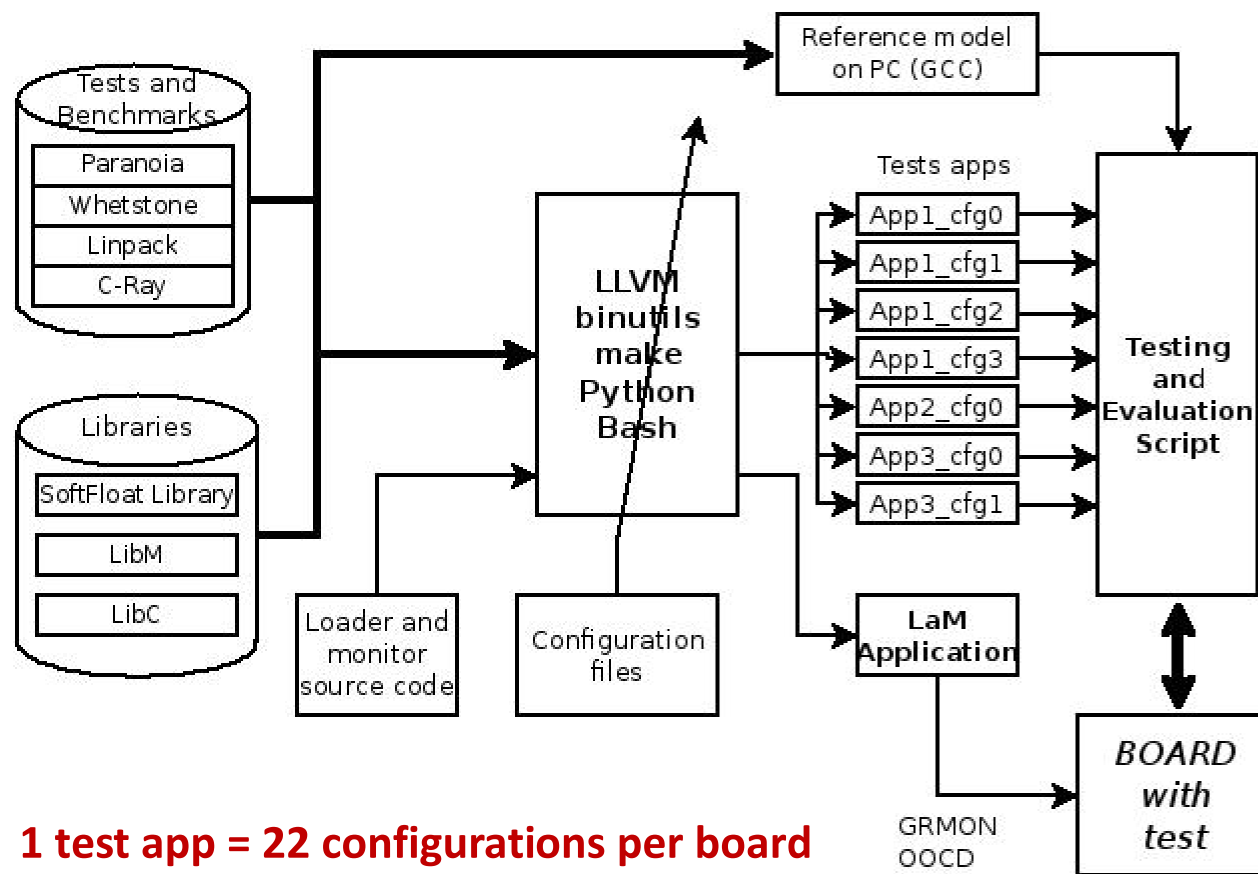
E.g. HW support for one selected precision only, and/or selected operations.



LLVM IR: as new 'subword' subtypes of vector integer types for
<1-32 x i1 in i32>,<1-16 x i2 in i32>,<1-10 x i3 in i32>,
<1-8 x i4 in i32>,<1-4 x i8 in i32>,<1-2 x i16 in i32>
e.g. %a = alloca <subword 1 x i2 in i32>, align 4
%z = alloca <subword 4 x i8 in i32>, align 4

SoftFloat library built using John Hauser’s SoftFloat version 3e

Benchmarking framework



1 test app = 22 configurations per board

Experience / Summary

- Precise computation always in the floating-point domain.
- Iterations within one data batch in the integer domain.
- Impact of instruction set extensions:
  *speedup per instruction * instruction frequency*.
- Just fusing very simple operations often achieves significant speedup (e.g. code expansion).
- LEON2FT w/ daiFPU w/ SWAR has an interesting potential for the space domain – GNSS, image processing.

Thank you