

Example processor models ASIP Designer

Version N-2018.03



Copyright (c) 2014-2018 Synopsys, Inc.

Copyright (c) 2014-2018 Synopsys, Inc. These Synopsys processor models capture an ASIP Designer Design Technique. The models and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys processor model or the associated documentation is strictly prohibited.

Introduction

Example Model Inventory

Location: The example processor models are located in the following directory

- On Linux: <SNPSROOT>/asip_designer/<version>/linux64/examples
- On Windows: <SNPSROOT>\ASIP Designer\<version>\win64\examples

Educational models	
Tinycore/Tinycore2	Example processor used in hands on tutorial
Ttcore	Historic model of DSP-like processor, still used in manuals
Tvec	Variants of SIMD processor (including OpenCL support)
Tvliw	Variants of VLIW processor
Microcontrollers	
Tmicro–Tnano	16-bit microcontrollers
DLX–TLX–FLX–ILX–PLX–VLX	Variants of 32-bit microcontroller (Hennessy and Patterson RISC)
Tmcu	32-bit microcontroller
Tzscale	Free and Open 32 bit RISC (Z-Scale implementation)
DSPs and generic parallel processors	
Tdsp	16/32-bit DSP

(1) Models that are not shipped with the ASIP Designer release but available upon request.

Domain-specific processors	
Tmotion	Accelerator of motion estimation kernel
Tcom8	SIMD processor optimized for some communication kernels
FFTcore	Scalar implementation of complex FFT
MXcore	Matrix processing ASIP for communication kernels
SHA256	Specialized instructions for secure hashing
Tgauss	Accelerator of Gaussian filtering on images
Primecore ⁽¹⁾	SIMD implementation of prime-factor algorithm for FFT and DFT
JEMA, JEMB ⁽¹⁾	Dual ASIP for JPEG encoding (respectively accelerating DCT, VLC)
Tcrypt ⁽¹⁾	Parallel AES Encryption and Decryption
Additional examples	
Tmicro_tcm	Example of tightly connected memory IO interface
Tmicro_cache	Example of cache IO interface
multicore_tutorial	Example of SystemC integration

What's New in the N-2018.03 example models

- DLX Family
 - MCLD model: bypasses added to multi cycle load
 - DLX model: a zero overhead looping mechanism was added
- Tgauss
 - New model which demonstrates how to implement an image processing kernel
 - Example kernel Gaussian filtering
 - Processor features: SIMD data path and line buffers
- Tmicro
 - DMA example based on Checkers pre/post functions
- Model fixes
 - By default the native implementation of inline functions is generated automatically
 - Tdsp: added chess view transitories
 - Tzscale: corrected range of int10ns16 and int10ps16 types

Feature overview

Example ASIP Models

Feature Matrix part 1

■ New in N-2017.09

	Tmicro	Tmcu	Tvliw Family			DLX Family									Tzscale	Tdsp	MXcore	Tcom8	JEMA	FFTcore	Primecore	Tvec Family					Tmotion
			Tvliw 1	Tvliw 2	Tvliw 3	DLX	TLX	FLX	ILX	PLX	VLX	PSTALL	MCLD	SHA256								Tvec 1	Tvec 2	Tvec 3	Tvec 4	Tvec 5	
ISA and Controller Features																											
• Instruction level parallelism		✓	✓	✓	✓											✓	✓	✓	✓	✓	✓						✓
• Variable length instructions		✓		✓												✓	✓										
• Zero overhead loops	✓	✓	✓	✓	✓	✓										✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
• Interrupts	✓	✓														✓		✓									
• Predication				✓																							
• Protected pipeline						✓		✓		✓	✓																
• Two-stage fetch pipe					✓																						
• OCD support	✓	✓				✓			✓	✓					✓	✓											
• Interleaved threading									✓	✓																	
• Precise stalling												✓															
• Multi cycle load													✓														
LLVM Support																											
• Basic LLVM support	✓	✓				✓					✓				✓	✓											
• C/C++: libcxx-lite + runtime library	✓	✓				✓					✓																
• C/C++: libcxx + newlib library		✓																									
• Vector features											✓																

Example ASIP Models

Feature Matrix part 2

■ New in N-_2017.09

	Tmicro	Tmcu	Tvlw Family			DLX Family									Tzscale	Tdsp	MXcore	Tcom8	JEMA	FFTCore	Primecore	Tvec Family					Tmotion
	Tvlw 1	Tvlw 2	Tvlw 3	DLX	TLX	FLX	ILX	PLX	VLX	PSTALL	MCLD	SHA256										Tvec 1	Tvec 2	Tvec 3	Tvec 4	Tvec 5	
Data-Types and Intrinsics																											
• Fractional arithmetic															✓												
• Floating-point emulation		✓			✓	✓	✓	✓	✓	✓	✓		✓														
• Floating-point hardware							✓									✓											
• Complex-arithmetic emulation																	✓										
• Complex-arithmetic hardware																			✓	✓							
• Application-specific intrinsics																		✓	✓	✓							✓
• Circular addressing															✓		✓			✓							
• FFT addressing modes																✓		✓	✓	✓							
• Extension instructions: security												✓															
SIMD Features																											
• Basic SIMD Support									✓								✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
• Vector predication																						✓		✓			
• Sequential vector predication																							✓		✓		
• Vector addressing (vector based)																								✓			
• Vector addressing (scalar based)																										✓	
• Matrix transpose																		✓		✓							
• On-chip debugging																					✓						

Manuals

- **Tinycore**

- [Guidelines for Hands-On Training](#)

- **Tmicro**

- [Tmicro Core - Processor Manual](#)
- [The Compiler Header File of the Tmicro Core](#)
- [Tmicro Core -- On Chip Debugging Manual](#)
- [Implementing Interrupts on the Tmicro Core](#)
- [Verification of the Tmicro Core](#)
- [Mapping of Tmicro Core to an FPGA board](#)
- [Mapping of Tmicro Core to a HAPS70 system](#)
- [Tmicro with Tightly Coupled Memory Interface](#)
- [Tmicro with Cache](#)

- **DLX–TLX–FLX–ILX–VLX**

- [DLX Core - Processor Manual](#)

- **Tzscale**

- [Tzscale - Processor Manual](#)

- **Tdsp**

- [Tdsp Core - Processor Manual](#)

- **Tmotion**

- [Tmotion core - Implementation of a Motion Estimation Algorithm](#)

- **Tvec**

- [Tvec core - Implementation of SIMD Instructions](#)

- **Tvliw**

- [Tvliw core - Processor Manual](#)

- **FFTcore**

- [Designing a Programmable FFT Accelerator with IP Designer](#)

- **Tgauss**

- [Tdsp Core - Processor Manual](#)

- **Multicore Tutorial**

- [Motion Estimation on a Multi-core System](#)

- **Generic regression tests**

- [HOWTO document](#)

Details

Tinycore

- Example processor that is used in hands on tutorial
- Processor features
 - 16 bit data path
 - Central register file with 8 fields
 - Supports basic ALU operations (addition, subtraction, bitwise logic operations, compare operations)
 - Indirect addressing with post-modification
 - Full set of control flow instructions, including zero overhead loop.
 - Instruction width: 13 bit
- Synthesis results for TSMC 28nm

Clock Frequency	Gate count	Core area
250 MHz	3600	2715 sqm
500 MHz	3800	2915 sqm
1 GHz	4700	3644 sqm

TCTcore

- Historic nML model of a DSP-like core, still used in many manuals
- Processor features
 - 16 bit data, 32 bit accumulator, 10 bit addresses
 - Distributed register files: AX/AY/AR, MX/MY/MR, Ia/Ma, Ib/Mb
 - ALU-Shift unit, MAC unit, two AGUs
 - Addressing modes: direct, indirect with post modify ($Ix = Ix + Mx$)
 - Instruction level parallelism: arithmetic | dual load/store
 - Instruction width : 18 bit
- Modelling features
 - Multiply-add data path pattern
 - Coupled register operands
 - Pointer modifier demotion

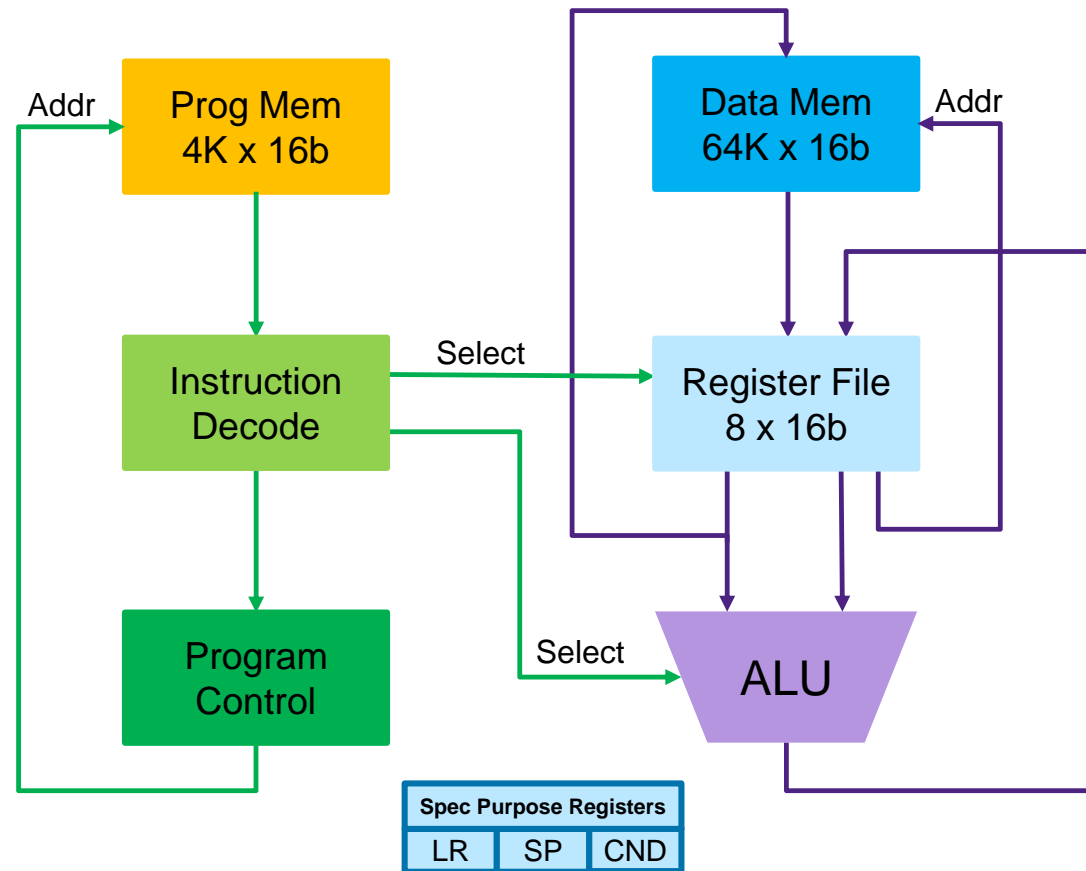
Tmicro

- 16 bit micro controller
- Processor features
 - 16 bit data, 16 bit addresses
 - Central register file with 8 register
 - ALU-Shift unit, MAC unit, AGU
 - Addressing modes: indirect with optional post modify
 - Instruction width : 16 bit, multi word multi cycle instructions
 - Gate count: 11kgates
- Modelling features
 - Dual precision implementation of 32 bit 'long' type and operations
 - Fast iterative division (divstep instruction)
 - Full implementation of on chip debugging (documented)
 - Vectored interrupts

Tnano Overview

- Minimal 16-bit RISC processor
 - 16-bit data and instruction
 - Small (8 field) register file
 - Simple addressing modes
- Full support for C based applications
 - 8/16/32-bit integer data types (32-bit is emulated)
 - Function call/return with full SW stack
 - Reduced runtime library (libc) included
- Very small implementation with simplified shallow high clock rate pipeline
 - 4.2K gates @ 1GHz using tcbn45g process library
- Intended usage:
 - Simple control applications, or as a minimal scalar base core in a larger design

Tnano Block Diagram



Tnano Base Instruction Set Architecture (1/3)

- Memory and registers

- Harvard architecture – separate data and program memory space
 - Data memory
 - 16-bit data
 - 16-bit address
 - 64K x 16b data memory space
 - Program memory
 - 16-bit instruction
 - 12-bit address
 - 4K x 16b program memory space
- Central register file (GPR) with 8 16-bit registers
- Separate special purpose registers (accessible via GPRs):
 - SP: function call stack pointer
 - LR: function call return pointer
 - CND: result of last compare instruction

Tnano Base Instruction Set Architecture (2/3)

•Data instructions

- Datapath and ALU (16-bit)
 - Standard (add, sub, and, or, xor, shift), and extended (add/sub with carry/borrow)
 - Mask instruction: return lower n bits (n= 0-15)
 - Register moves
 - GPR↔GPR
 - GPR↔(special purpose regs)
 - Sel instruction
 - return source0 or source1 based on CND
 - 4-bit immediate version available for mask, and, shift, sel
 - 7-bit immediate version available for add
- Bitwise compare and set condition register (CND)
 - Equal, not-equal, signed/unsigned greater-than/less-than
 - Compare chaining: set condition and'd/or'd with last value of CND
 - 5-bit immediate version available for equal/not-equal (with optional chaining)

Tnano Base Instruction Set Architecture (3/3)

- Memory Access
 - 16-bit load/store
 - Address modes:
 - Register indirect w/ optional post increment(1-3) or decrement(1-4)
 - SP + 8-bit(unsigned) offset
- Control and misc.
 - Direct branch instructions (12-bit absolute program address)
 - Unconditional branch
 - Conditional (if CND=1) branch
 - Unconditional branch with link (LR←return address)
 - 12-bit address: 4K instruction word limit
 - Indirect (LR) branch instructions
 - Call (jump to LR, LR←return address)
 - Return (jump to LR)
 - NOP

Tnano Micro-architecture

- 4-stage pipeline
 - IF: instruction fetch
 - ID: instruction decode
 - E1: execute 1 – ALU and compare instructions
 - E2: execute 2 – Data memory load result
- No bypass, register forwarding or HW stalls
 - Register read and write (except for load) happens in E1 stage
 - All delay slots and dependencies scheduled by compiler
 - NOPs inserted where necessary
 - 1 cycle delay for memory load
 - Register write in E2 stage
 - 2 delay slots for branch, call, return

Tnano Optional Features

- Hardware multiplier
 - Adds 16x16 mul instruction → 16-bit result
- Zero overhead loop
 - Adds loop instruction with start (LS), end (LE) and count (LC) registers
- Interrupts
 - Adds support for 8 external maskable HW interrupts
 - Adds SW interrupt instruction
 - Optional shadow register file for fast context switching
- Byte addressable data memory
 - Adds byte load and store instructions and byte addressable memory
 - Memory size reduced to 64K x 8-bit
- On-chip Debug (OCD)
 - Adds logic to support HW on-chip debug via JTAG debug port
- **Options can be enabled via pre-processor defines when building model**

Tnano

- A number of options can be enabled/disabled
- Synthesis results
 - Technology: TSMC 45 nm G
 - Clock frequency: 1 GHz
 - Gate count: from 4278 to 9300

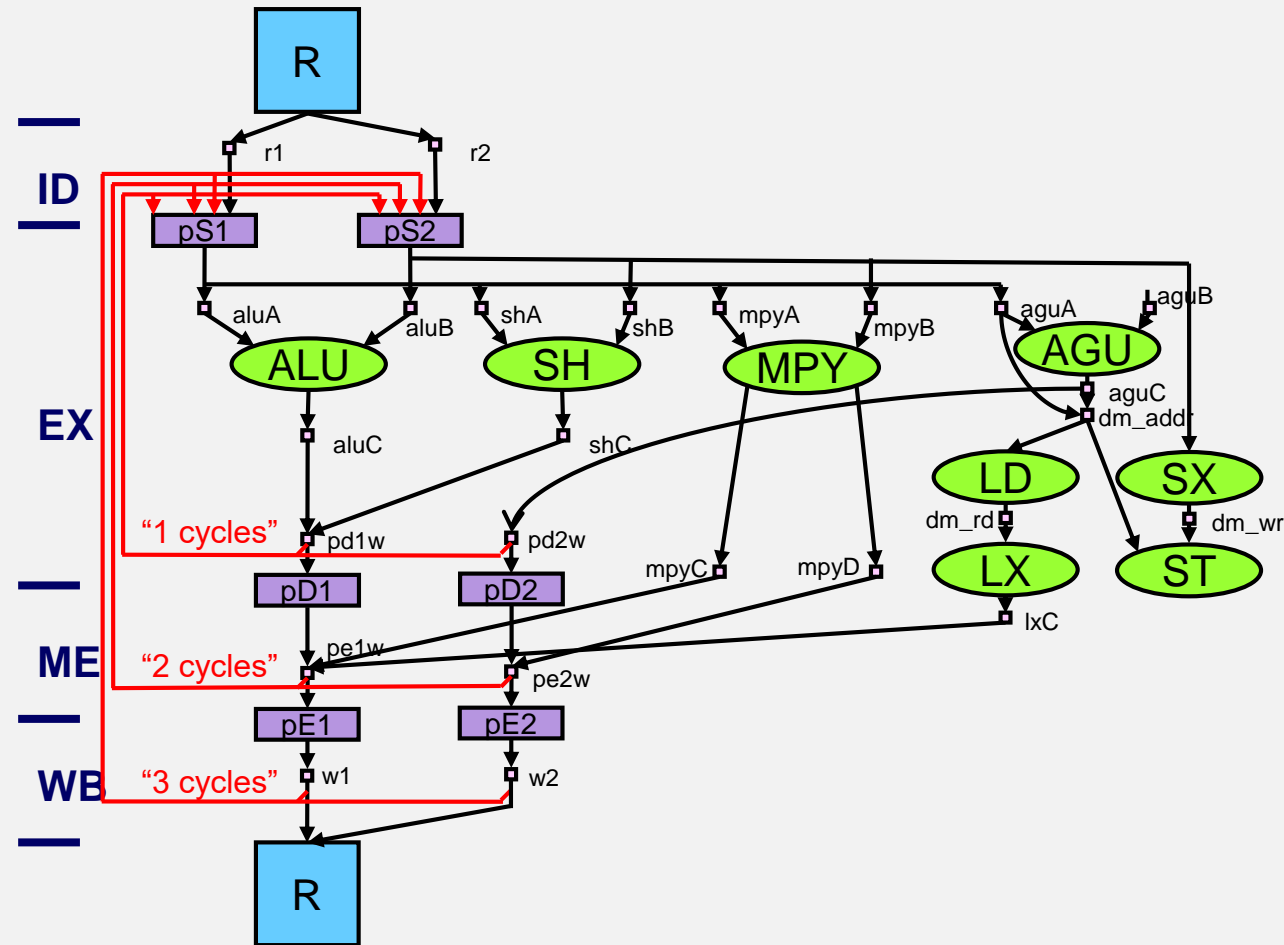
	Gate count	Delta	
Tnano Basic processor	4278		
Tnano + Interrupts	5192	914	21%
Tnano + OCD	6812	2534	51%
Tnano + Zero overhead loops	4894	571	13%
Tnano + Byte addressable DM	4395	117	3%
Tnano + Multiplier	5172	894	21%
Tnano + Interrupts +OCD +Zero overhead loops + Byte addressable DM + Multiplier	9300		

DLX

- 32 bit RISC (Hennessy and Patterson).
- Processor features
 - 32 bit data, 32 bit addresses, 32 bit instructions.
 - Central register file with 32 fields.
 - Byte addressed data and program memory.
 - Five stage protected pipeline: IF – ID – EX – ME – WB.
- Modelling features
 - Register bypasses and hardware stall rules.
 - Multi cycle functional unit for division.
 - IO interface maps 8/16/32 bit transfers onto 4 byte-wide memory banks.
 - Implementation of on chip debugging.
 - Software emulation of IEEE floating point operations.
 - Evaluation of Boolean expressions on ALU.

DLX

Data path and bypass network

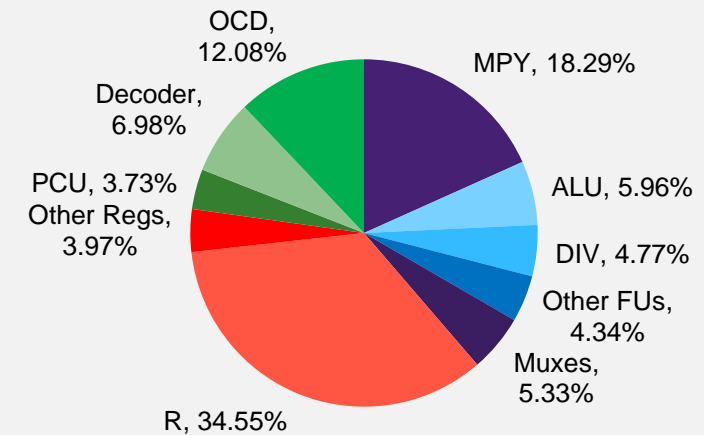


DLX Synthesis results for TSMC 28nm

- Technology: TSMC 28nm
- Area versus Frequency

	DLX		TLX	
Clock Frequency	Gate count	Core area	Gate count	Core area
250 MHz	31 k	23105 sqm	24 k	20276 sqm
500 MHz	34 k	25080 sqm	31 k	22411 sqm
700 MHz	38 k	26581 sqm		
800 MHz	40 k	26647 sqm		

- Area distribution per unit



DLX Variants

- TLX
 - Reduced register file with 16 fields
 - Reduced pipeline: IF – ID – EX, no register hazards
- FLX
 - PDG implementations of single precision IEEE float operations
- ILX
 - Pipeline with read in ID, write in WB : IF – ID – EX – ME – WB.
 - Interleaved threading with 4 threads, hence hazard free execution
 - Division on multi cycle functional unit, thread is skipped until done
 - Mutex test and set instruction
 - Implementation of on chip debugging

DLX Variants

- IO Interfaces

- MCLD: IO interface example, load vector of 4 words in 4 cycles

Cycle	:	0	1	2	3	4

A-phase	:	A0	A1	A2	A3	
D-phase	:		D0	D1	D2	D3
rv_addr	:	a	a+1	a+2	a+3	
rv_data	:		a000	ab00	abc0	abcd

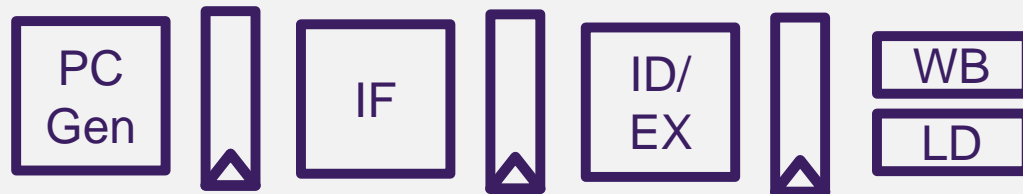
- PSTALL: IO interface example of store-load dependency. The load is stalled in the EX stage in case there is a dependency

sw:	IF	ID	EX	ME(*)	WE	(*)	write new value
lw:		IF	ID	EX(#)	ME	WB	(#) read old value

Tzscale

- Implementation of the RISC-V ISA: Free and Open RISC Instruction Set Architecture
- Processor features
 - 32 bit data, 32 bit addresses, 32 bit instructions.
 - Central register file with 16 or 32 fields.
 - Byte addressed data and program memory, unaligned data memory access.
 - Three stage protected pipeline: IF – ID/EX – WB, cf Z-scale model

<http://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf>



- Implemented RISC-V ISA
 - RV32I – Base integer instruction set
 - RV32M – Multiplication and Division
 - RV32C – Compressed (16 bit) instructions

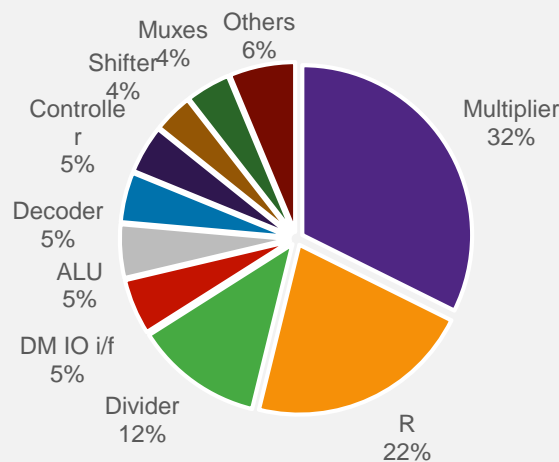
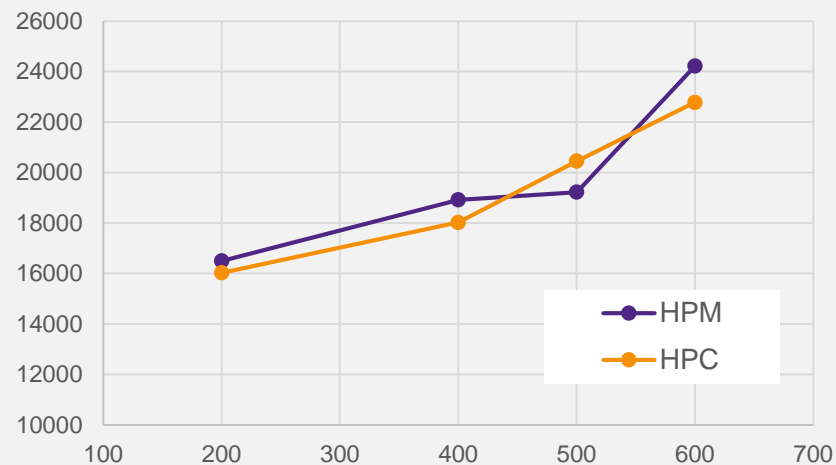
Tzscale

- Modelling features
 - Register bypasses and hardware stall rules.
 - Multi cycle functional unit for division.
 - On chip debugging.
 - IO interface maps unaligned 8/16/32 bit transfers onto 2 memory banks with byte enable.
 - Software emulation of IEEE floating point operations.

Tzscale

• Synthesis results

- Architecture configuration
 - Register file with 16 fields
 - IO interface for unaligned access into two 32 bit banks
- Technology
 - TSMC 28nm HPC and HPM
 - Fmax: 600 MHz
 - Gate count: 16k .. 24k



• Simulation speeds benchmarks

<i>ISS Type and Configuration</i>	<i>Simulation speed [MIPS]</i>
Cycle Accurate Full set of debug features	0.44
Cycle Accurate Fast configuration	2.60
Instruction Accurate Full set of debug features	4.80
Instruction Accurate Fast configuration with JIT	60.58

Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz

Tzscale

- Software benchmarks (16 field register file)

- Dhrystone1 (all code in a single file)

<i>Front End</i>	<i>Dhryst/s</i>	<i>DMPS/MHz</i>
Chess	1404	0.80
LLVM	2849	1.62

- Dhrystone2 (original version with two C files)

<i>Front End</i>	<i>Dhryst/s</i>	<i>DMPS/MHz</i>
Chess	1404	0.80
LLVM	2143	1.22

- Coremark

<i>Number of registers</i>	32	32	32	32	16	16	16	16
<i>Compressed instructions present in ISA</i>	YES	YES	NO	NO	YES	YES	NO	NO
<i>Compiler frong end</i>	Chess	LLVM	Chess	LLVM	Chess	LLVM	Chess	LLVM
<i>Coremark/MHz</i>	1.94	2.28	1.94	2.31	1.85	2.23	1.85	2.28
<i>Code size*</i>	7062	7914	7608	8672	7360	8244	7876	8804

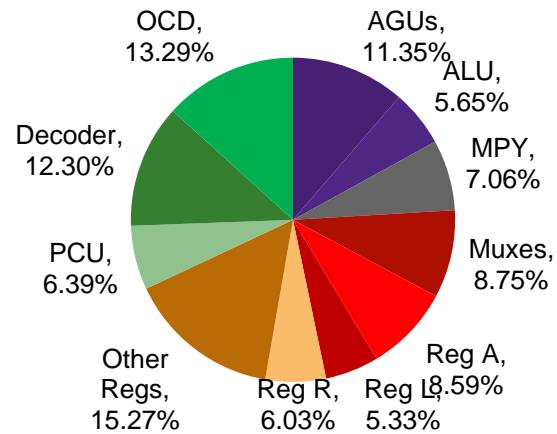
(*) Total size of: core_list_join, core_main, core_matix, core_portme, core_state and core_util

Tdsp

- Example DSP core.
- Processor features
 - Single MAC, 16/32 bit fractional arithmetic
 - Three-way ILP: arithmetic + dual load
 - Efficient support of ETSI EFR basic operations
 - Addressing modes: direct, indirect with post modify, circular
 - Variable length instructions:
 - Instructions can be 16 or 32bit
 - Instruction fetch is always 32 bit
 - Optional alignment of jump targets (instruction elongation)
 - On chip debugging
 - Interrupts
 - IO interface for odd/even bank selection

Tdsp

- Synthesis results for TSMC 28nm
 - Area breakdown per unit



- Area-speed values

Clock Frequency	Gate count	Core area
250 MHz	24 k	21588 sqm
500 MHz	29 k	23332 sqm
650 MHz	42 k	29087 sqm

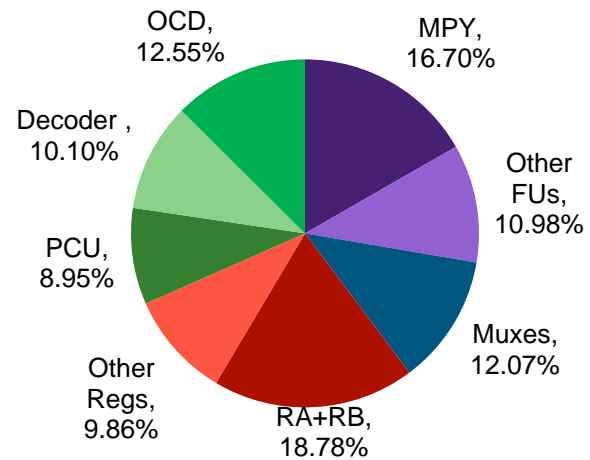
Tmcu

Architecture features

- 32 bit MCU
- Harvard architecture
- Variable length instructions
 - 16-bit arithmetic, move and control flow instructions
 - 32-bit arithmetic, move and control flow instructions with large immediate operands
 - 32-bit instructions where 16-bit arithmetic operations are executed with in parallel with a move
- 16 central registers, 32-bit wide
 - R0-R7: operand registers for arithmetic instructions
 - R8-R15: indirect address registers for move instr
- Data path with ALU, shifter and single cycle multiplier
- Byte addressed DM, little endian
- Three-stage pipeline
 - 32-bit instruction fetch and alignment stage:
 - Instruction storage can be unaligned
 - 16 or 32-bit instruction decode stage, including:
 - Address sent to memory (for loads)
 - Address sent to program memory (for unconditional jumps)
 - Execution stage:
 - Arithmetic execution & store to destination register
 - Write to destination register (for loads)
 - Execution of stores
 - Address sent to program memory (for conditional jumps)
- Benchmarks
 - Dhrystone/MHz : 1691
 - CoreMark/MHz : 2.39

Tmcu

- Synthesis results for TSMC 28nm
 - Area breakdown per unit



- Area-speed values

Clock Frequency	Gate count	Core area
250 MHz	32 k	24738 sqm
500 MHz	36 k	27516 sqm
700 MHz	46 k	31177 sqm

Mxcore: ASIP for Block Matrix Inversion

- **Requirements**


- Inversion of 8x8 matrix using block-based algorithm
- Matrix elements are complex floating-point values
- 64k inversions per second

- **Computational load (“real” operations) [Di Wu et al, ISVLSI 07]**

Multiplication	2032
Addition	1788
Division	4
Total	3824

- **Cycle budget**

- Assume 500 MHz clock frequency
- $500\text{MHz} / 64\text{k-inv./s} = \mathbf{7812}$ cycles per 8x8 inversion



~ 2 cycles/op
Fits on
scalar
architecture

Mxcore Architecture Overview

- Floating-point data path
 - Supports “real” operations, hardware support for complex operations is not required
 - IEEE754 floating-point representation
 - Separate FMUL and FADDSUB unit
 - Shallow pipeline: single-stage FP units
 - Float division support: fdiv_pre, div_step3, fdiv_post
- Integer ALU
 - 32-bit, matched to width of single-precision float
 - Control code and address generation
- Register set
 - 8 x 32-bit data registers
 - 8 x 18-bit pointer registers
- Instruction set
 - Limited instruction level parallelism: arithmetic | load/store
 - pointer + idx[0..63] addressing can access half of complex 8 x 8 matrix
 - Mix of 16 and 32-bit word instructions
 - Short instructions reduce program size
 - Long instructions for parallelism and immediate operands
- Two level of zero overhead loops
- C support for (unsigned) int, (unsigned) long long, float

Mxcore Interfaces

- **Control signals**

- In: clock, reset

- **Memory interfaces**

Bus width				
Name	Use	Load	Store	Addr
PM	program	32	-	16
DM	data	32	32	16

- Synchronous:
 - addr cycle n, store cycle n, load cycle n+1
 - can be adapted to real memory in I/O interface (supported by tools)
- 16-bit address buses can be adapted as needed

Mxcore Results – (1)

- **8 x 8 cmplx_t with 32-bit float re and im**
- **Library functions for matrix operations**
 - mxneg, mxadd, mxsub, mxmul, mxinv2, mxinv4
 - mxdecomp4/8, mxupdate4/8

- **Results**

Func + desc	Cycle count
mxinv8	6871
mxmul4	3612
mxmul2	1202
mxinv2	304

Memory sizes	
PM	1491 16-bit words = 2.9 KB
DM	530 16-bit words = 1.1 KB

Mxcore Instructions generated by compiler

```
.text global 0 void_mxadd2__P_A2cmplx_t__P_A2cmplx_t__P_A2cmplx_t
.function_start
/*      0      */      ld r0,(a1,#0)
/*      1      */      ld r1,(a2,#0)
/*      2      */      fadd r0,r0,r1 | ld r1,(a1,#1)
/*      4      */      ld r2,(a2,#1)
/*      5      */      fadd r1,r1,r2 | ld r2,(a1,#2)
/*      7      */      ld r3,(a2,#2)
/*      8      */      fadd r0,r2,r3 | st r0,(a0,#0)
/*     10      */      st r1,(a0,#1)
/*     11      */      st r0,(a0,#2)
.swstall conflict__dm_addr
/*     12      */      nop
/*     13      */      ld r0,(a1,#3)
/*     14      */      ld r1,(a2,#3)
/*     15      */      fadd r1,r0,r1 | ld r0,(a1,#5)
/*     17      */      ld r2,(a2,#5)
/*     18      */      fadd r0,r0,r2 | st r1,(a0,#3)
/*     20      */      st r0,(a0,#5)
.swstall conflict__dm_addr
/*     21      */      nop
/*     22      */      ld r0,(a1,#4)
/*     23      */      ld r1,(a2,#4)
/*     24      */      fadd r2,r0,r1 | ld r0,(a1,#7)
/*     26      */      ld r1,(a2,#7)
/*     27      */      fadd r1,r0,r1 | ld r0,(a1,#6)
/*     29      */      ld r3,(a2,#6)
/*     30      */      fadd r0,r0,r3 | st r2,(a0,#4)
/*     32      */      st r0,(a0,#6)
/*     33      */      st r1,(a0,#7)
/*     34      */      rts2
```

Mxcore Results – (2)

Synthesis Inputs

Library

16 nm FF 0.8 V 85° GL

Constraints

clock	500 MHz
input delay	30 % of clock period
output delay	30 % of clock period
port driving cell	strength-2 D flipflop
port load	5 x NAND3 input pin
clock uncertainty	164 ps
clock latency	5 – 10 % of clock period
timing derate	6 %
floorplan	default (square, 60 % utilization)

Synthesis Result

Core area (μm^2)	Leaf cell count	Equiv NAND2 count	Slack
8.6 K	16 K	55 K	met

Mxcore Results – (3)

- Power estimations
 - simulate one inversion on RTL in VCS, producing SAIF file
 - synthesizing with SAIF input produces power estimation

Result for mxinv8	
	power (mW)
random matrix 1	4
random matrix 2	4

- Accuracy of pre-layout RTL power estimation
 - TLU+ phys. characteristics, ideal clock network
 - only 1 program simulated
 - RTL simulation sometimes conservative
- margin ~ 10 .. 20 %

Tvec

Example processor with SIMD instructions

- Processor features

- lib1:

- Basic SIMD model with vector of 8x 16 bit

- lib2

- Vector predication: 8 condition bits to enable or disable the vector lanes

- lib3

- Nested vector predication based on sequential execution and condition stack

- lib4

- Lane based vector addressing: each vector memory lane is addressed separately

- lib5

- Scalar based vector addressing: each vector lane has global access to vector memory

- lib6

- Multiple vector types: 8x 16 bit and 4x 32 bit

Tvliw

- Example model of 4-slot VLIW core
- Processor features
 - lib1
 - Two scalar arithmetic units, 16 bit data width
 - Two load/store units
 - Four slot ILP: D0, D1, M0, M1
 - lib2
 - Predicated execution: each slot is predicated by a condition
 - lib3
 - Variable length instructions: instructions can be 16, 32, 48 or 64 bit
 - Instruction fetch is always 64 bit
 - Optional alignment of jump targets (elongation)
 - lib4
 - Two stage PM fetch

Tmotion

- Motion estimation (SAD + search)**

<i>Design</i>	<i>Cycle count</i>	<i>Gate count</i>
1 Mapping on 16 bit microcontroller	69.088	11.658
2 Scalar absdiff unit and parallel loads	13.864	12.625
3 SIMD unit	5.000	27.387
4 SIMD unit + 1 parallel load	3.344	28.120
5 SIMD unit + 2 parallel loads	2.000	28.923
6 Transparent unaligned vector load		
a) Dual port	1.793	26.145
b) Wait state	2.321	28.868
c) Banks	1.793	27.093

Tcom8

- SIMD optimised for some communication kernels
- Processor features
 - Based on Tmicro (16 bit microcontroller)
 - SIMD data path
 - Vector data registers, partitioned (VA[2], VB[2]), 8x16 bit (v8int_t,v4cmplx_t)
 - Vector accumulator, partitioned (WA[2], WB[2]), 8x40 bit (v8accumt_t,v4caccu_t)
 - Eight 16x16->32 multipliers
 - 8x40 bit adders, shifters and bitwise unit
 - Conjugate, swap, round_sat, transpose
 - Vector memory
 - Two banks enable dual load
 - Cyclic, bit-reverse and next-butterfly addressing modes.

Tcom8

- Processor features (continued)
 - Pipeline
 - IF: Instruction fetch
 - ID: Instruction decode, address generation, start load
 - E1: scalar ALU, vector ALU, vector multiplier
 - E2: vector add
 - ILP
 - AM instructions (arithmetic || moves)
 - PAR instructions (4 slots, mix of arithmetic and moves)
 - Instruction set (32 bit)
 - AM instructions : arithmetic || moves
 - PAR instructions : 4 slots, mix of arithmetic and moves
 - noILP : immediate formats and miscellaneous instructions

Tcom8

- Kernels
 - FFT :
 - 2 complex butterfly operations and 2 complex mult per cycle
 - 256 point FFT in 586 cycles
 - 512 point FFT in 1188 cycles
 - 1024 point FFT in 2498 cycles
 - 2048 point FFT in 5388 cycles
 - FIR
 - 6 cycles / sample for 32 tap FIR

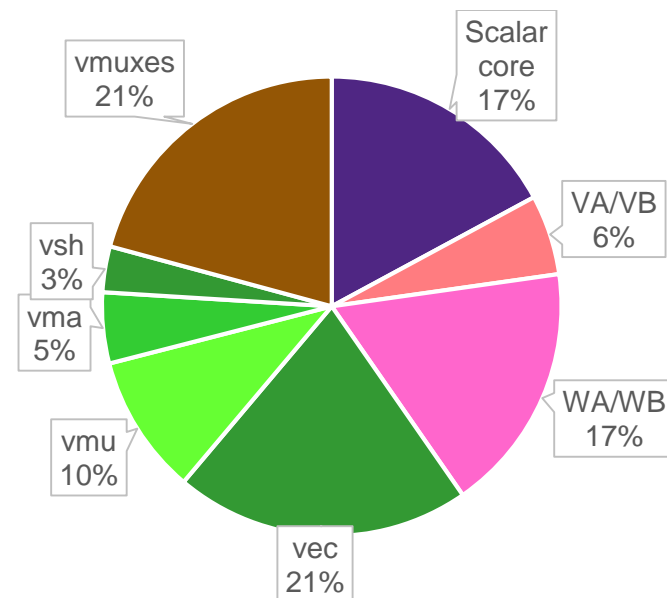
Tcom8

- Synthesis results for TSMC 28nm

- Area breakdown per unit

- VA[2] / VB[2]: 8x 16 bit register file
 - WA[2] / WB[2]: 8x 40 bit register file
 - Vec: general purpose vector unit
 - Vmu: 8x 16x16 bit multiplier
 - Vma: 8x 40 bit adder

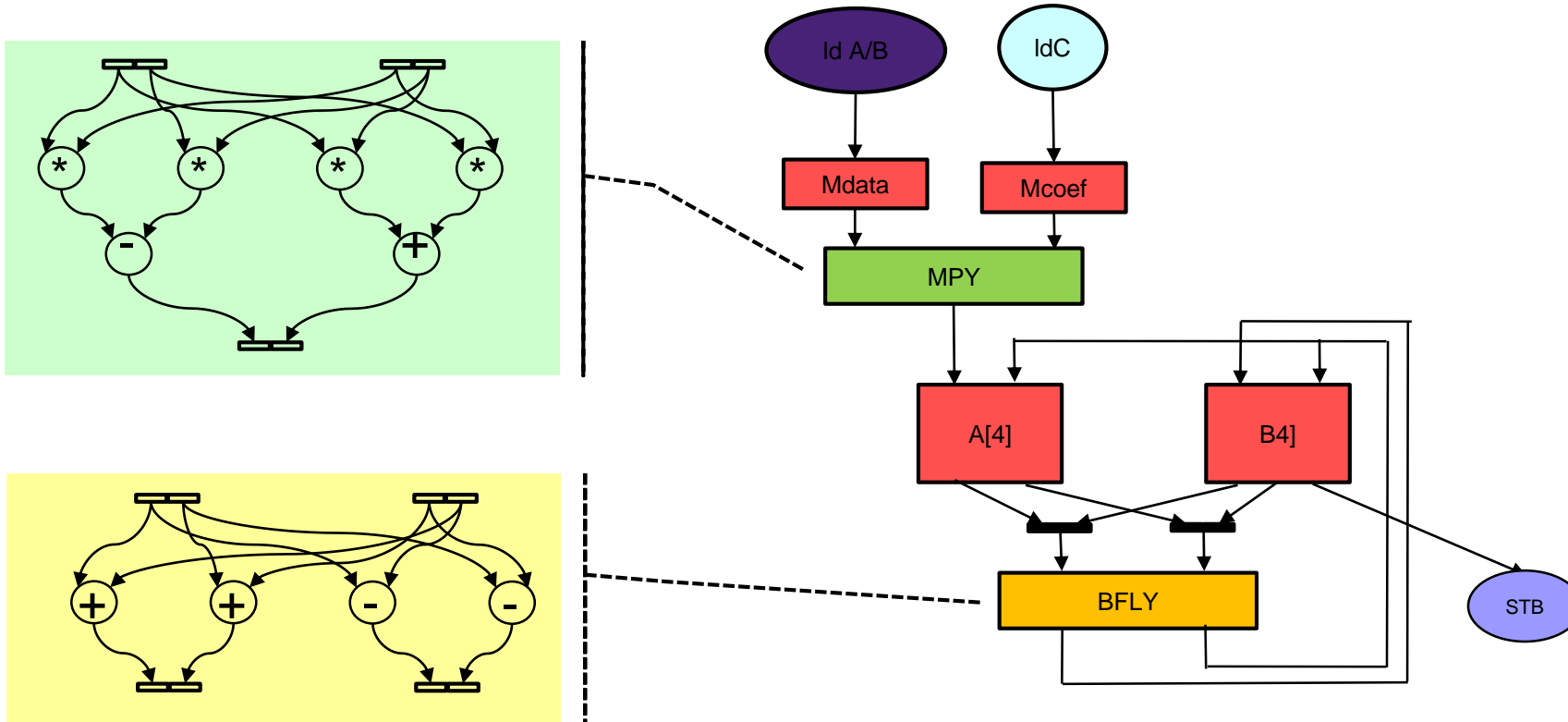
- Area-speed values



Clock Frequency	Gate count	Core area
400 MHz	113k	81105 sqm
500 MHz	126k	87790 sqm
530 MHz	131k	89523 sqm

FFTcore

- Data path



FFTcore

- Coalesced inner loops
 - Classical 3 nested loop structure: pre and postamble of bfly loop consume many cycles

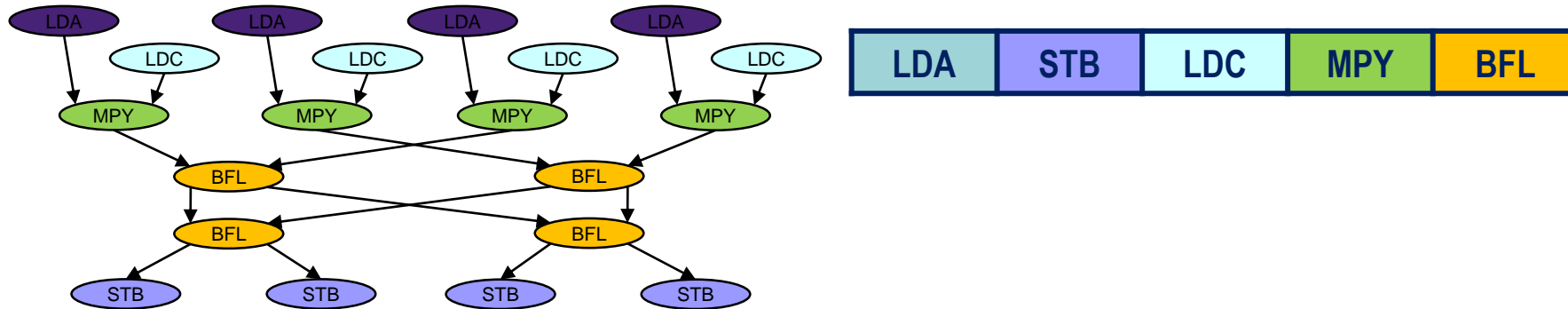
```
for (int stage = 0; stage < 3; stage++)  
    for (int block = 0; ...) {  
        // pre  
        for (int bfly = 0; ...  
            // post  
        }  
    }
```

- Optimised 2 nested loop structure: requires dedicated address generation

```
for (int stage = 0; stage < 3; stage++)  
    for (int i = 0; i < Nbfly; i++) {  
        ... = *dap; dap = next_bfly(dap, d_mask);  
    }  
}
```

FFTcore

- Radix 4 inner loop: matched instruction



```
DO cnt,LE
(delay slot)
md=*pa(next_bfly) *pb(+s)=b1      mc=*pr(next_bfly_rdx4)  a2=md*mc  b3,b2=bfly(a2,a3)
md=*pa(+s)         *pb(+s)=b3      mc=*pr(+s)          a3=md*mc  b1,a2=bfly(a1,a2)
md=*pa(+s)         *pb(+s)=b0      mc=*pr(+s)          a1=md*mc  b0,a3=bfly(a0,a3)
md=*pa(+s)         *pb(next_bfly)=b2 mc=*pr(+s)          a0=md*mc  b1,b0=bfly(b1,b0)
```

FFTcore

• Results

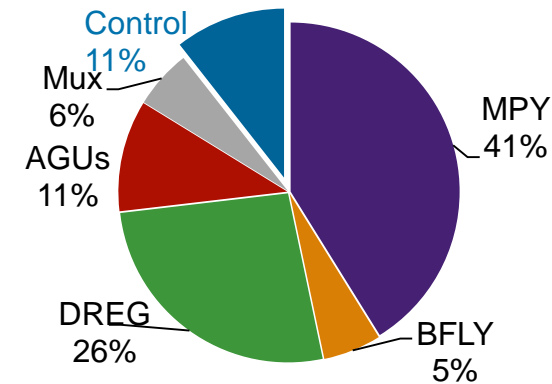
- Application specific processor (600 lines nML)
 - Matched instruction level parallelism
 - Specific data paths: complex multiplier, complex butterfly unit
 - Specific addressing modes: enable loop coalescing
- Efficient assembly code generated
 - Radix-4: 4 cycles/ butterfly, radix-2: 2 cycles/butterfly
 - 4096-point FFT (radix-4)
 - 24,668 cycles (50 μ s / transform for 500 MHz clock)
 - 99.6% of time spent in inner loops
 - 2048-point FFT (2x 1024-pt radix-4 + 1x 2048-pt radix-2)
 - 12,376 cycles (25 μ s / transform for a 500 MHz clock)
- Efficient RTL code generated (Verilog/VHDL)
 - 20k gates, 500 MHz clock, 28nm TSMC

	Radix 2		Radix 4 + 2	
N	Cycles	Code size	Cycles	Code size
64	458	42	237	73
128	978	52	570	88
256	2144	42	1088	60
512	4712	52	2637	75
1024	10358	42	5195	75
2048	22654	52	12376	90
4096	49292	42	24668	60

FFTcore

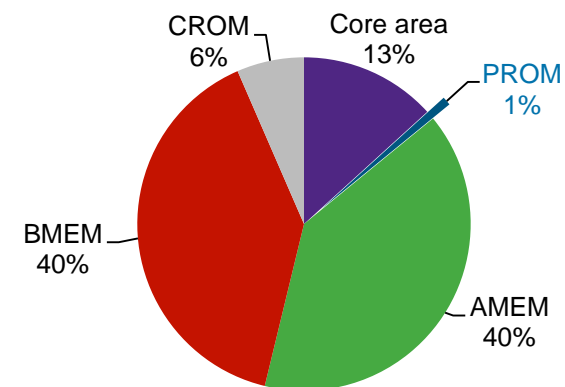
- Detailed area metrics
 - Core cell area: 20k gates, 500 MHz clock, 28nm TSMC

Module	Cell Area (sqm)	Gates
MPY	3161.88	8218
BFLY	426.56	1109
DREGs	2031.35	5280
AGUs	817.34	2124
Muxes	432.84	1125
Control	813.62	2115
<i>Total</i>	<i>7683.59</i>	<i>19970</i>



- Core total area and memory area (sqm)

Core area	15625
PROM (128*20 bit)	992
AMEM (4k * 48 bit)	46800
BMEM (4k * 48 bit)	46800
CROM (4k * 32 bit)	7690



- Control overhead (**Controller + PROM**) is limited

Primecore - Features

- Features
 - Supports DFT prime-factor algorithm and FFT for LTE
 - FFT: 8, 16, 32, 64, 128, 256, 512, 1024 and 2048 point
 - DFT: 6, 12, 18, ... 1296 points (total of 46 sizes)
 - SIMD
 - 8 way
 - 2x16 bit complex value per lane,
 - ILP
 - Three data path units (complex fixed point)
 - VU1: butterfly
 - VU2: vcmul, vmul, vsummul
 - VU3: irdx
 - Two vector memories: DM and CM
 - ILP of 5: VU0 | VU1 | VU2 | LD/ST-DM | LD-CM

Primecore - Features

- Features
 - Scaling after each stage
 - Distributed and partitioned register files
 - Variable length instructions: 16, 32 and 64 bit
 - Pipeline: IF – ID – E1 – E2

Primecore - Features

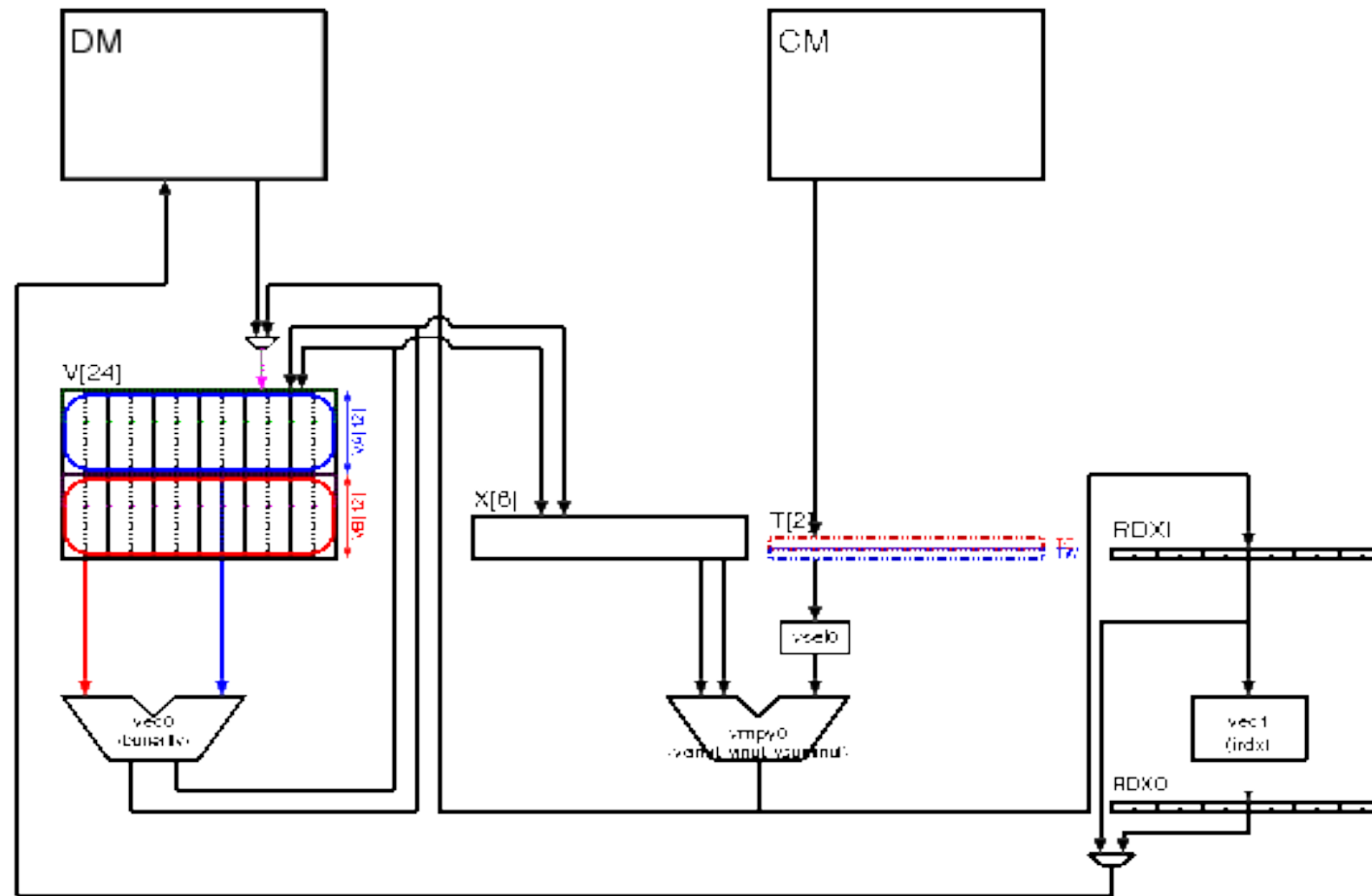
- Results
 - Efficient RTL implementation: 325k gates @ 500 MHz in 65 nm
 - Cycle counts for FFT and DFT

FFT		
Points	Cycle count	Cycles /point
8	4	0.50
16	12	0.75
32	28	0.88
64	43	0.67
128	69	0.54
256	172	0.67
512	307	0.60
1024	601	0.59
2048	1189	0.58

DFT		
Size	Cycle count	Cycles /point
6	4	0.67
12	12	1.00
18	17	0.94
24	17	0.71
...	...	
1080	686	0.64
1152	713	0.62
1200	755	0.63
1296	798	0.62

- 2014 point FFT
 - 2.4 μ s/transform
 - 420k transforms/second

Primecore – data path



Primecore – C code

```
inline void vmodule10io(vcmplx_t i0, .. i9, vcmplx_t& o0, .. o9,
                        base_t c5_1, cmplx_t cm1, cmplx_t cm2)
{
    vcmplx_t t0,t1,t2,t3,t4,t5,t6,t7,t8,t9;

    vmodule5io(i0,i6,i2,i8,i4, t0,t6,t2,t8,t4, c5_1, cm1, cm2);
    vmodule5io(i5,i1,i7,i3,i9, t5,t1,t7,t3,t9, c5_1, cm1, cm2);

    vcbf0(t0,t5,o0,o5,0,0,SCALE);
    vcbf0(t6,t1,o6,o1,0,0,SCALE);
    vcbf0(t2,t7,o2,o7,0,0,SCALE);
    vcbf0(t8,t3,o8,o3,0,0,SCALE);
    vcbf0(t4,t9,o4,o9,0,0,SCALE);
}
inline void vmodule5io(vcmplx_t i0, .. i4, vcmplx_t& o0, .. o4,
                        base_t c1, cmplx_t cm1, cmplx_t cm2)
{
    vcmplx_t t1,t2,t3,t4,t5,t6,t7,y1,y2,y3,y4,y6;
    vcbf0(i1,i4,t1,t3,0,0,SCALE);
    vcbf0(i2,i3,t2,t4,0,0,SCALE);
    vcbf0(t1,t2,t5,t6,0,0,SCALE);
    y6 = t6 * c1;
    vcbf0(i0,t5,o0,t7,SCALE*2,2,SCALE);
    vcbf0(t7,y6,y1,y2,0,0,0);
    y3 = vsummul(t3,t4,cm1);
    y4 = vsummul(t3,t4,cm2);
    vcbf1(y1,y4,o4,o1,0,0);
    vcbf1(y2,y3,o3,o2,0,0);
}
```

Primecore – Generated instructions

Loop scheduled in 20 cycles

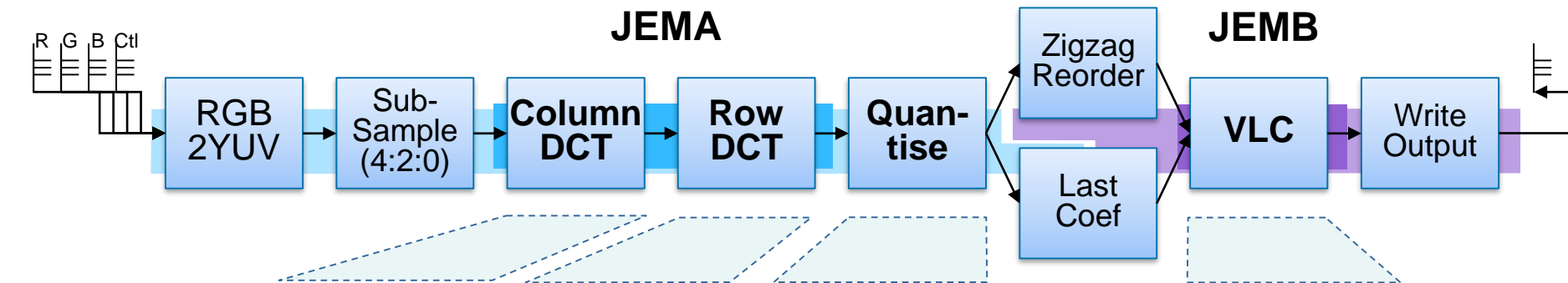
```
vmod10_s1 void_vmod10_s1__Pvcmplx_t__Pvcmplx_t__sint
397 nop ; nop ; v03=dm[p0+6*m0] ; nop ; p2=#0
398 v04=v03 ; nop ; v13=dm[p0+4*m0] ; nop ; r0=m0
399 v06=(v04 +v13 )/2; x01=(v04 -v13 )/2; nop ; v15=dm[p0+7*m0] ; tc=cm[p2] ; p2=#640
400 v05=v15 ; nop ; v02=dm[p0+2*m0] ; nop ; m1=#16
; v14=dm[p0+8*m0] ; nop ; nop
; v01=dm[p0+1*m0] ; nop ; nop
; v12=dm[p0+3*m0] ; nop ; nop
; v15=dm[p0+9*m0] ; nop ; nop
405 v07=(v03 -v15 )/2; nop ; v00=dm[p0+0*m0] ; nop ; nop
406 v09=(v00/4-v18/4)/2; v09=(v00/4-v18/4)/2; v13 =vsmul(x03,x00,tc[0]); v03=dm[p0+5*m0] ; nop ; nop
407 v07=(v07 -v16 )/2; nop ; nop ; nop ; do r0,429
408 v11=(v03/4-v19/4)/2; v11=(v03/4-v19/4)/2; v15 =vmul(x07,tc[[1]]) ; nop ; [p0+m1] ; nop
409 x06=(v10 +v20 )/2; x07=(v10 -v20 )/2; v12 =vsmul(x01,x05,tc[1]); nop ; tw=cm[p2+5*m0]; nop
410 v08=(v11 +v15 ) ; v11=(v11 -v15 ) ; v15 =vsmul(x01,x05,tc[0]); v03=dm[p0+6*m0] ; nop ; nop
411 v23=(v11 +v13r ) ; v21=(v11 -v13r ) ; rdx=vcmul(x07,tw) ; v02=dm[p0+2*m0] ; tw=cm[p2+0*m0]; nop
412 v09=(v09 +v14 ) ; v10=(v09 -v14 ) ; rdx=vcmul(x06,tw) ; v13=dm[p0+4*m0] ; tw=cm[p2+8*m0]; nop
413 v11=(v10 +v15r ) ; v10=(v10 -v15r ) ; v15 =vsmul(x03,x00,tc[1]); v14=dm[p0+8*m0] ; nop ; rdxo=irdx(rdx)
414 x07=(v11 +v23 )/2; x05=(v11 -v23 )/2; nop ; dm[p1+5*m0]=rdxo; nop ; rdxo=irdx(rdx)
415 x02=(v10 +v21 )/2; x07=(v10 -v21 )/2; rdx=vcmul(x07,tw) ; dm[p1+0*m0]=rdxo; tw=cm[p2+7*m0]; nop
416 v22=(v08 +v15r ) ; v23=(v08 -v15r ) ; rdx=vcmul(x07,tw) ; v15=dm[p0+7*m0] ; tw=cm[p2+6*m0]; nop
417 v10=(v09 +v12r ) ; v11=(v09 -v12r ) ; nop ; v12=dm[p0+3*m0] ; nop ; rdxo=irdx(rdx)
418 x07=(v11 +v23 )/2; x04=(v11 -v23 )/2; nop ; dm[p1+8*m0]=rdxo; nop ; rdxo=irdx(rdx)
419 x07=(v10 +v22 )/2; x06=(v10 -v22 )/2; rdx=vcmul(x07,tw) ; dm[p1+7*m0]=rdxo; tw=cm[p2+4*m0]; nop
420 v04=v03 ; v05=v15 ; rdx=vcmul(x07,tw) ; v01=dm[p0+1*m0] ; tw=cm[p2+3*m0]; nop
421 v06=(v04 +v13 )/2; x01=(v04 -v13 )/2; rdx=vcmul(x05,tw) ; v15=dm[p0+9*m0] ; tw=cm[p2+2*m0]; rdxo=irdx(rdx)
422 v17=(v02 +v14 )/2; x05=(v02 -v14 )/2; rdx=vcmul(x02,tw) ; dm[p1+6*m0]=rdxo; tw=cm[p2+1*m0]; rdxo=irdx(rdx)
423 v16=(v05 +v12 )/2; x00=(v05 -v12 )/2; rdx=vcmul(x04,tw) ; dm[p1+4*m0]=rdxo; tw=cm[p2+9*m0]; rdxo=irdx(rdx)
424 v18=(v06 +v17 )/2; x07=(v06 -v17 )/2; rdx=vcmul(x06,tw) ; dm[p1+3*m0]=rdxo; [p2+m1] ; rdxo=irdx(rdx)
425 v07=(v01 +v15 )/2; x03=(v01 -v15 )/2; v14 =vmul(x07,tc[[1]]) ; dm[p1+2*m0]=rdxo; tw=cm[p2+5*m0]; rdxo=irdx(rdx)
426 v19=(v07 +v16 )/2; x07=(v07 -v16 )/2; v12 =vsmul(x01,x05,tc[1]); v00=dm[p0+0*m0] ; nop ; nop
427 v10=(v00/4+v18/4)/2; v09=(v00/4-v18/4)/2; v15 =vmul(x07,tc[[1]]) ; v03=dm[p0+5*m0] ; [p0+m1] ; nop
428 v20=(v03/4+v19/4)/2; v11=(v03/4-v19/4)/2; v13 =vsmul(x03,x00,tc[0]); dm[p1+1*m0]=rdxo; nop ; rdxo=irdx(rdx)
429 x06=(v10 +v20 )/2; x07=(v10 -v20 )/2; nop ; dm[p1+9*m0]=rdxo; [p1+m1] ; nop
430 rt
```

Critical slot: 20 ld/st

Cycle counts

size	stage0	stage1	stage2	vload/store		vmultiplications		vbutterflies		primecore	
				#	avg/cycle	#	avg/cycle	#	avg/cycle	cycles	rel cycles
DFT (PFA)											
6			irdx6	2	0.33	0	0.00	0	0.00	4	0.67
12		2	irdx6	4	0.33	2	0.17	1	0.08	12	1
18		3	irdx6	6	0.33	4	0.22	3	0.17	17	0.94
24		3	irdx8	6	0.25	4	0.17	3	0.13	17	0.71
30		5	irdx6	10	0.33	8	0.27	7	0.23	20	0.67
900	10	15	irdx6	600	0.67	530	0.59	645	0.72	755	0.84
960	10	12	irdx8	480	0.50	382	0.40	468	0.49	559	0.58
972	9	18	irdx6	648	0.67	576	0.59	729	0.75	798	0.82
1080	9	15	irdx8	540	0.50	501	0.46	594	0.55	686	0.64
1152	8	18	irdx8	576	0.50	436	0.38	594	0.52	713	0.62
1200	10	15	irdx8	600	0.50	530	0.44	645	0.54	755	0.63
1296	9	18	irdx8	648	0.50	576	0.44	729	0.56	798	0.62
FFT											
8			irdx8	2	0.25	0	0.00	0	0.00	4	0.5
16		2	irdx8	4	0.25	2	0.13	1	0.06	12	0.75
32		4	irdx8	8	0.25	4	0.13	4	0.13	28	0.88
512	8	8	irdx8	256	0.50	160	0.31	208	0.41	307	0.6
1024	8	16	irdx8	512	0.50	352	0.34	480	0.47	601	0.59
2048	16	16	irdx8	1024	0.50	768	0.38	1088	0.53	1189	0.58

JPEG Dual-ASIP Design



Design target:
 - 100k gates
 - 1 pixel/cycle

→ Average workload per macroblock (384 cycles)

>	0
*	480
+	2592
= *x	768
*x =	768
= *c	480

>	384
*	384
+	384
= *x	384
*x =	384
= *c	384

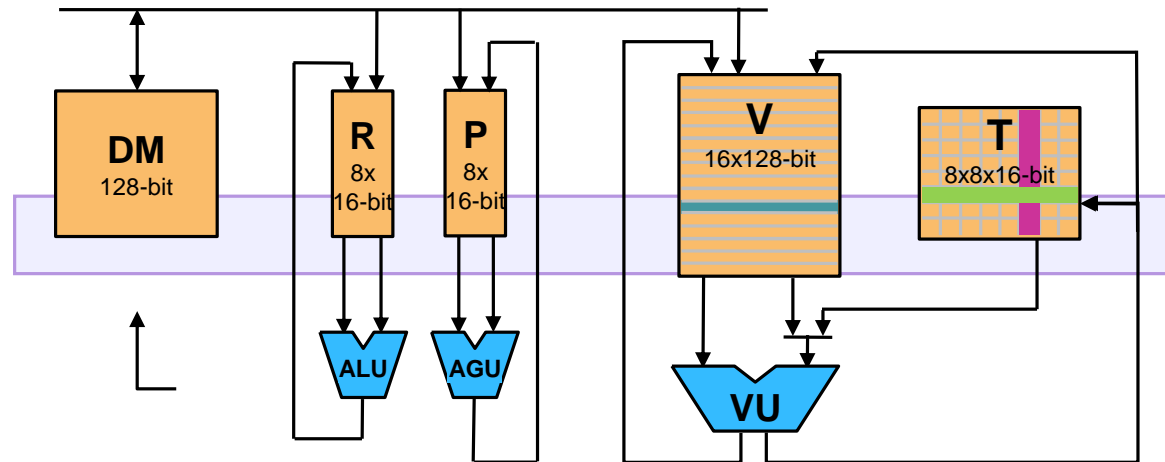
>	188
*	0
⊗	188
= *x	47
*x =	47
= *c	94

⊗ = VLC primitives

- Col/Row DCT → Design specialised 8-element vector ASIP – “**JEMA**”
- VLC → Design specialised scalar ASIP – “**JEMB**”
- Other blocks → Explore mapping and partitioning onto **JEMA/JEMB**

Design, mapping, partitioning enabled by ASIP Designer

“JEMA” DCT ASIP – Initial



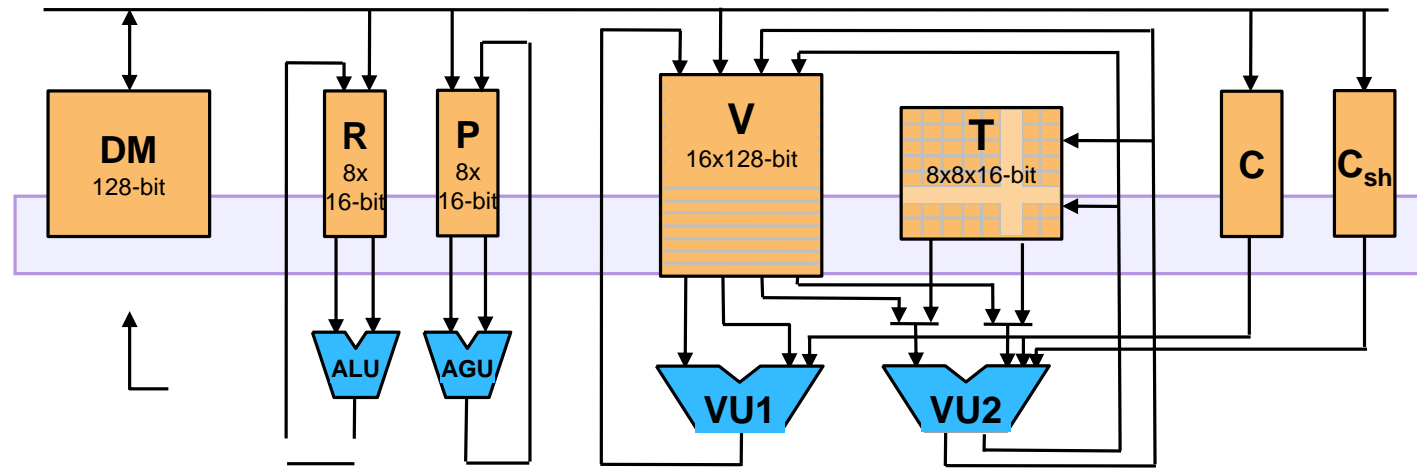
- Architecture

- VU: basic vector arithmetic – vadd, vsub
quantisation and scaling primitives – vscale, vmul
- T: transposable register file – write columns, read rows

- Performance

- Cycle count of compiled code: 150 cycles / DCT loop,
i.e. **150% over cycle budget**

“JEMA” DCT ASIP – Speed Opt.



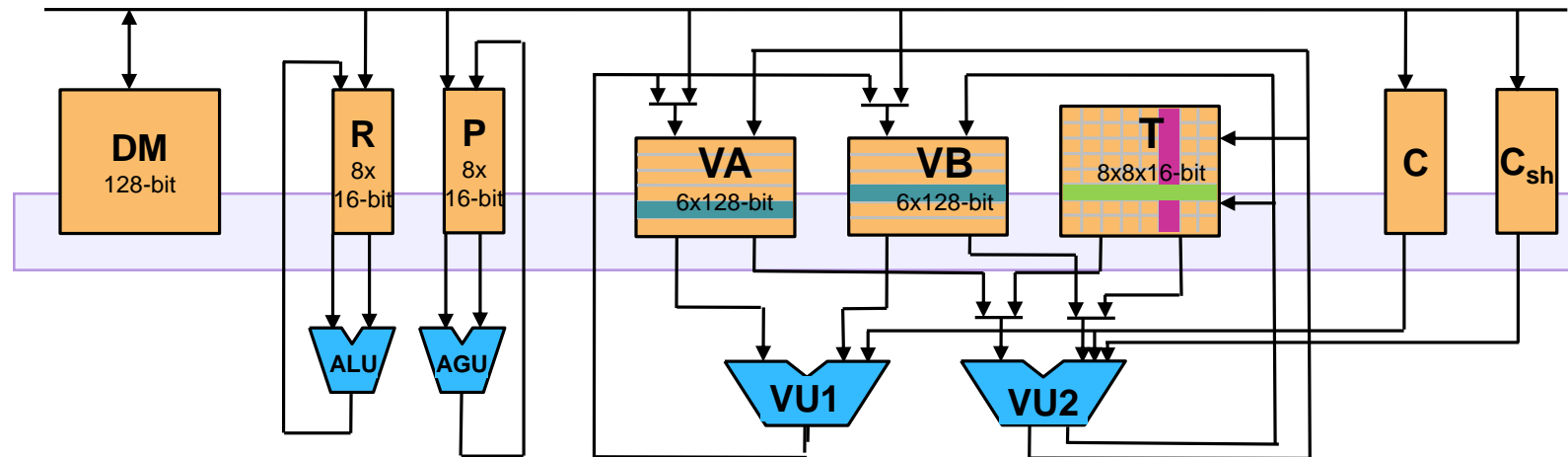
- Architecture

- Added VU2 // VU1, load-store // VU2
- Added vaddsub, optimised distribution of VU1 & VU2 instructions
- Added dual T operands, constant register files

- Performance

- Compiled code: 59 cycles / DCT loop, i.e. **within cycle budget**
- Gate count: 90K, i.e. **almost entire JPEG encoder area budget**

“JEMA” DCT ASIP – Area Opt.



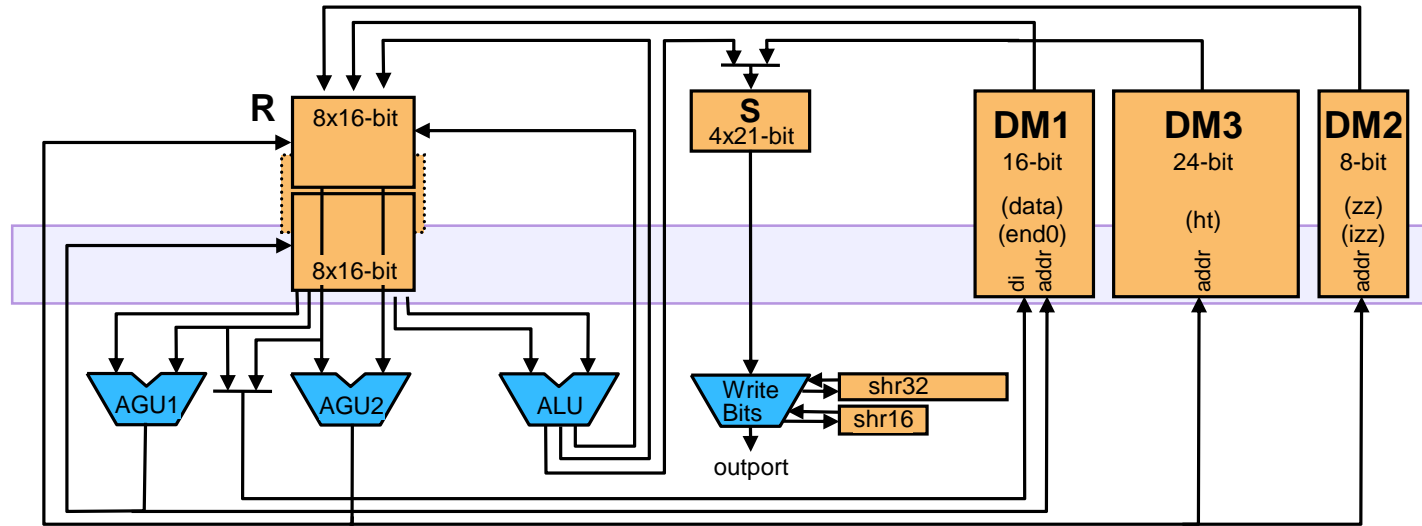
- Architecture

- Split vector register file: V (16 elements, 4R + 4W ports)
 - VA + VB (6 elements, 2R + 2W ports each)
- Encoded opcode fields: 48-bit → 28-bit instruction word

- Performance

- Compiled code: 59 cycles / DCT loop, i.e. **within cycle budget**
- Gate count: 65K, i.e. **within area budget**

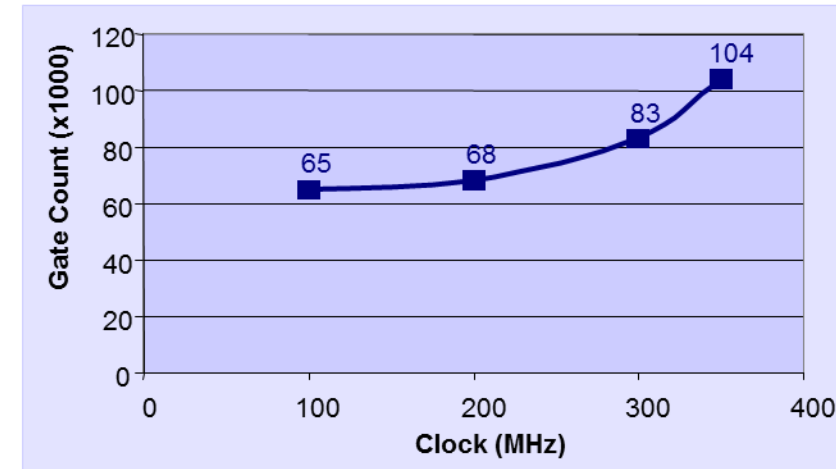
“JEMB” VLC ASIP



- Architecture (after exploration with ASIP Designer)
 - Multiple data memories, for different data-types
 - Instruction-level parallelism: DM1 // DM2 // ALU // WriteBits
 - WriteBits: shift primitives for bit concatenation
 - ALU: standard arithmetic + Huffman table address & code calculation primitives
 - AGUs: mix of pre- and postincrement modes

Dual-ASIP - Results

- Gate count: 65K (JEMA) + 11K (JEMB) = 76K
- RTL results (CMOS90) →
- Throughput: 1 pixel / cycle
- Optimizing C-compiler
- Effort: 1 person-month



- Preparing C source code
- Initial modelling of JEMA ASIP
- Architectural exploration* JEMA ASIP
- Initial modelling of JEMB ASIP
- Architectural exploration* JEMB ASIP
- Tuning partitioning
- Documentation

3 days

3 days

4 days

3 days

3 days

3 days

4 days

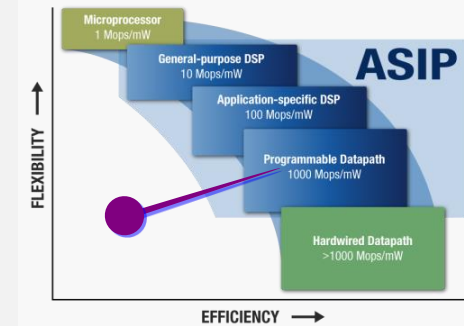
23 days

* Tuning processor model, compilation, simulation, RTL generation and synthesis

AES Encryption & Decryption

Specification

- http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- Encryption, decryption and key expansion
- Throughput: ~20Gbit/s range (with min SIMD2)



AES Encryption & Decryption

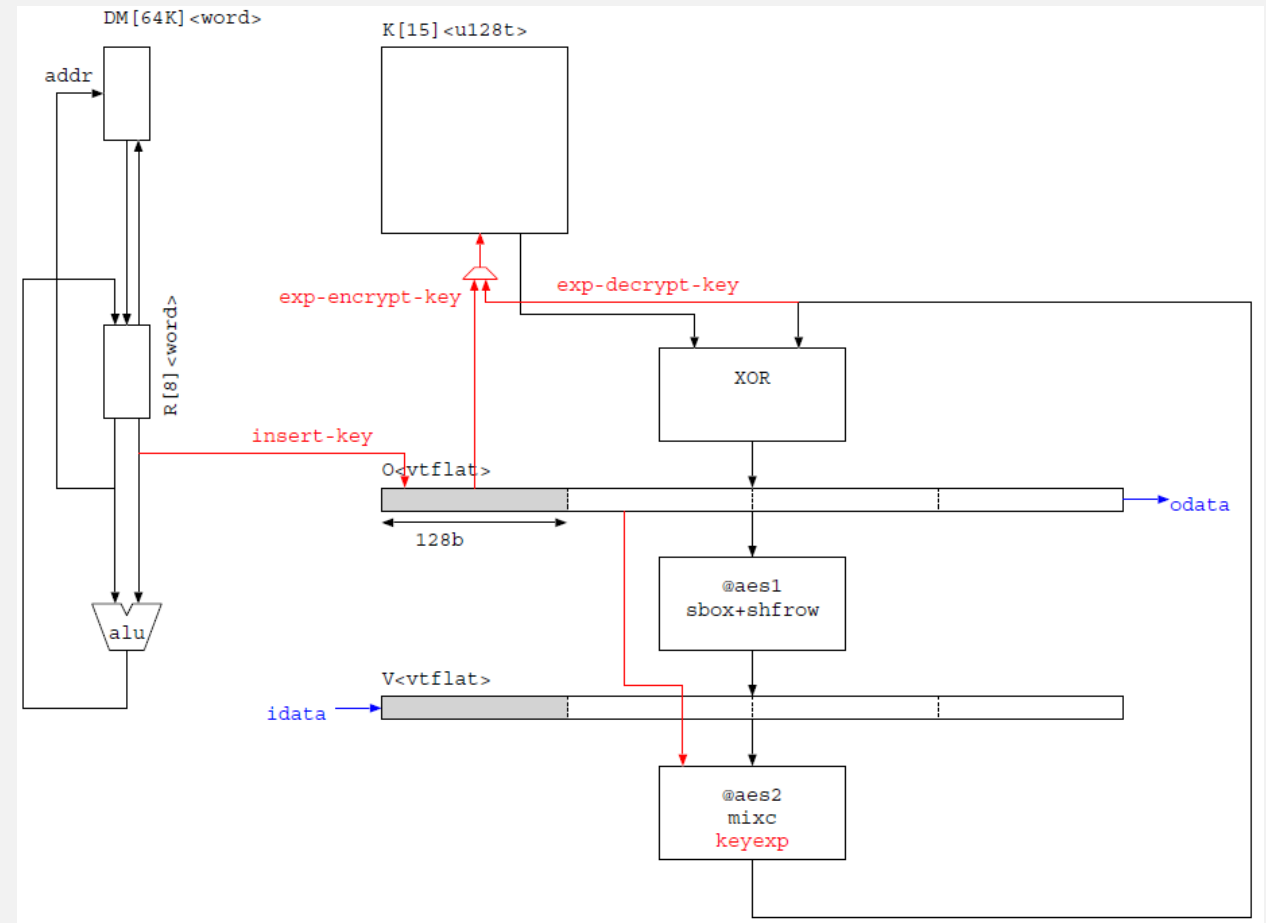
Algorithm

- AES algorithm executes multiple rounds of 4 basic steps:
 - Initial Round: AddRoundKey
 - Rounds (10, 12 or 14, depending on key length: 128, 192 and 256 bit):
 - SubBytes: a non-linear substitution step where each byte is replaced with another according to a lookup table
 - ShiftRows: a transposition step where the last three rows of the state are shifted cyclic a certain number of steps
 - MixColumns: a mixing operation which operates on the columns of the state, combining the four bytes in each column
 - AddRoundKey: each byte of the state is combined with a block of the round key using bitwise xor
 - Final Round (no MixColumns): SubBytes + ShiftRows + AddRoundKey
- KeyExpansion: round keys are derived from the cipher key using Rijndael's key schedule

AES Encryption & Decryption

Architecture of Tcrypt

- Simple 16 bit scalar core:
 - Complete C support
 - 16-bit char, short & int
 - 32-bit long (emulated)
 - ALU also used for address generation
- 128-bit data path on the side:
 - Specialized instructions
 - SIMD4 (depicted), but scalable
 - Each round split into 2 primitive operations:
 - AES1: SubBytes & ShiftRows
 - AES2: MixColumns & AddRoundKey
 - 2 vector REGs: VA & VB
 - Key RF: K[15]
 - 2 streaming ports: idata & odata
 - Connections for **KeyExpansion**
 - Sbox HW part of AES1 is reused
 - Keyexp HW part of AES2



AES Encryption & Decryption

Application is programmed in C code

- C code is mapped onto ASIP architecture using an optimizing compiler
- C code can also be compiled native (g++ or VisualC++) and executed and tested using the native application header of the Trypt core

```
void aes128_encrypt(int16_t key[8], /* 16b per word */
                   int data_size /* nr data-blocks */)
{
    // state for key-expansion
    vdata exp_key = undef_vdata();

    // insert first 128b of key
    for (int i=0; i<8; i++) {
        exp_key = upd_elem(exp_key, key[i], i);
    }

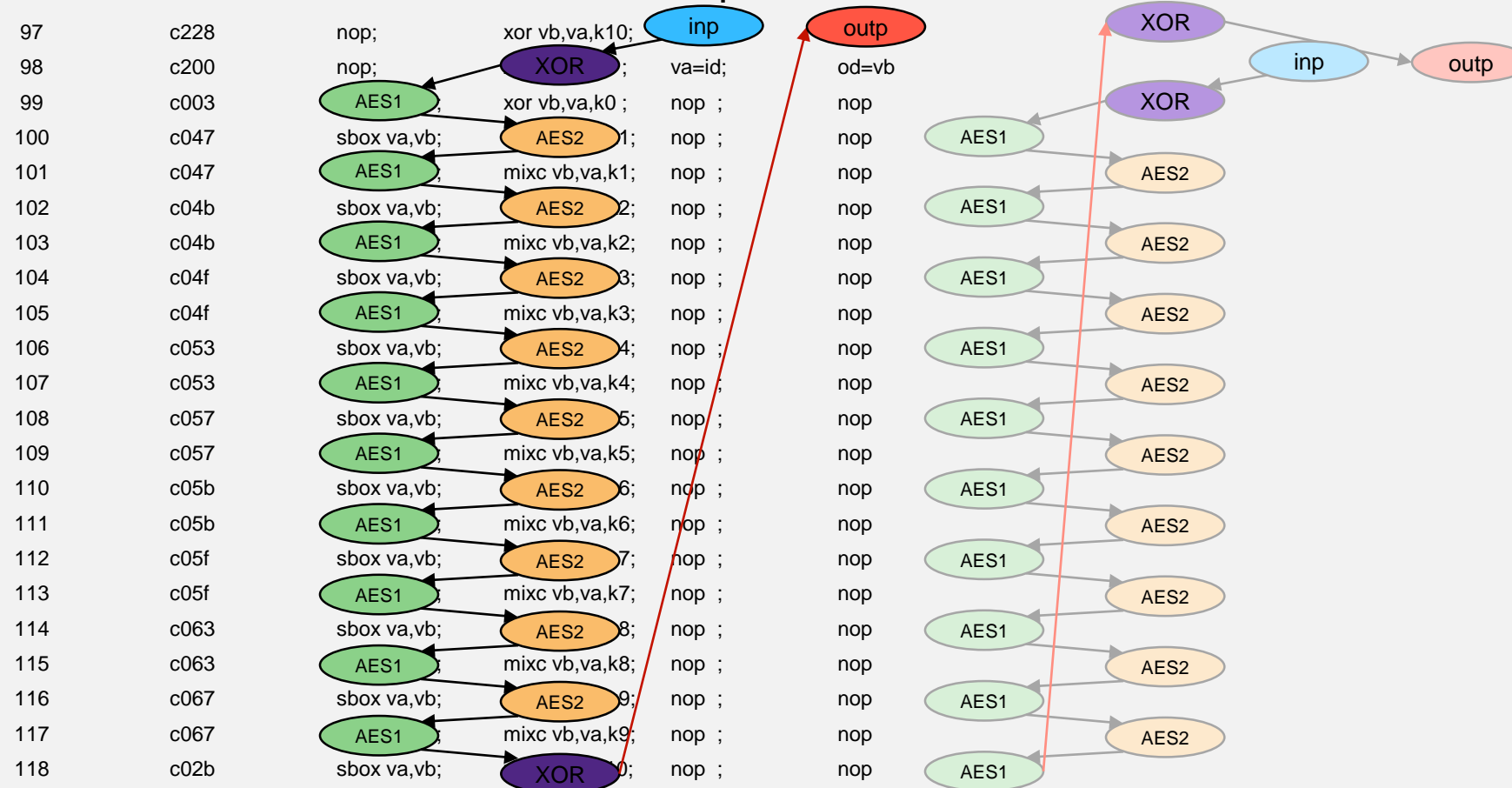
    /* expand keys */
    uint128 key0 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(0));
    uint128 key1 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(1));
    uint128 key2 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(2));
    uint128 key3 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(3));
    uint128 key4 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(4));
    uint128 key5 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(5));
    uint128 key6 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(6));
    uint128 key7 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(7));
    uint128 key8 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(8));
    uint128 key9 = ext_key(exp_key); exp_key = aesenc_keyexp(exp_key, KEYEXP128(9));
    uint128 key10 = ext_key(exp_key);

    // encrypt
    data_size >= 1; // unrolled with a factor 2 (interleaved) to keep ILP busy
    for (int i=0; i<data_size; i++) chess_loop_range(2,) chess_prepare_for_pipelining {
        vdata input0 = get_stream();
        vdata input1 = get_stream();
        vdata vs0 = input0 ^ key0;
        vdata vs1 = input1 ^ key0;
        vs0 = aesenc_round(vs0) ^ key1;
        vs1 = aesenc_round(vs1) ^ key1;
        vs0 = aesenc_round(vs0) ^ key2;
        vs1 = aesenc_round(vs1) ^ key2;
        vs0 = aesenc_round(vs0) ^ key3;
        vs1 = aesenc_round(vs1) ^ key3;
        vs0 = aesenc_round(vs0) ^ key4;
        vs1 = aesenc_round(vs1) ^ key4;
        vs0 = aesenc_round(vs0) ^ key5;
        vs1 = aesenc_round(vs1) ^ key5;
        vs0 = aesenc_round(vs0) ^ key6;
        vs1 = aesenc_round(vs1) ^ key6;
        vs0 = aesenc_round(vs0) ^ key7;
        vs1 = aesenc_round(vs1) ^ key7;
        vs0 = aesenc_round(vs0) ^ key8;
        vs1 = aesenc_round(vs1) ^ key8;
        vs0 = aesenc_round(vs0) ^ key9;
        vs1 = aesenc_round(vs1) ^ key9;
        vs0 = aesenc_last_round(vs0) ^ key10;
        vs1 = aesenc_last_round(vs1) ^ key10;
        put_stream(vs0);
        put_stream(vs1);
    }
}
```

AES Encryption & Decryption

Generated assembly code

The schedule of the AES-128 inner-loop:



AES Encryption & Decryption

Results

- Number of SIMD can be scaled (minimum 2) to increase the throughput
- Gatecount (TSMC 65GP , excluding memories):
 - Basic: ~25k gates (zero lanes), including 5k gates for scalar core
 - Increase with 33k gates/lanes
- PM and DM can be very small, depending on extra functionality on scalar core:
 - DM: minimum 32 words (16 +16 for stack)
 - PM: AES-128 encryption is 80 words (16-bit)
- Design completed in 3 weeks

#SIMD	gatecount	freq	Throughput		
			AES-128	AES-192	AES-256
2	91K	1GHz	23.3Gbps	19.7Gbps	17.1Gbps
4	158K	1GHz	46.5Gbps	39.4Gbps	34.1Gbps
6	224K	1GHz	69.8Gbps	59.1Gbps	51.2Gbps
8	TBD	TBD	93.1Gbps	78.8Gbps	68.3Gbps

But, Does it Accelerate?

Let's look at AES-128 performance in ARMv8

ARMv8 also adds some new cryptographic instructions for hardware acceleration of AES and SHA1/SHA256 algorithms. These hardware AES/SHA instructions have the potential for huge increases in performance, just like we saw with the introduction of AES-NI on Intel CPUs a few years back. Both the new advanced SIMD instructions and AES/SHA instructions are really designed to enable a new wave of iOS apps.

64-bit Performance Gains

Geekbench 3 was among the first apps to be updated with ARMv8 support. There are some minor changes between the new version of Geekbench 3 and its predecessor (3.1/3.0), however the tests themselves (except for the memory benchmarks) haven't changed. What this allows us to do is look at the impact of the new ARMv8 A64 instructions and larger register space. We'll start with a look at integer performance:

Apple A7 - AArch64 vs. AArch32 Performance Comparison			
	32-bit A32	64-bit A64	% Advantage
AES	91.5 MB/s	846.2 MB/s	825%

Source: <http://www.anandtech.com/show/7335/the-iphone-5s-review/4>

- ARMv8 A32
 - No AES acceleration instructions
 - 91.5 MB/s (0.732 Gbps)
- ARMv8 A64
 - AES acceleration instructions
 - 846.2 MB/s (6.77 Gbps)

9x
- Tcrypt ASIP Accelerator
 - SIMD2 = 23.3 Gbps
 - SIMD4 = 46.5 Gbps
 - SIMD6 = 69.8 Gbps
 - SIMD8 = 93.1 Gbps

30-100x

AES Encryption & Decryption

Further improvements

- Critical path is in the scalar ALU. Only minor frequency gain be achieved. This is dependent on the technology used.
- ASIP is doing encryption or decryption, so half of the resources are not used, but both data paths are there (limited sharing). Separate ASIP for encryption and decryption doubles performance for a limited gate count increase.
- Sbox operation is the most expensive, but is not used for 2 out of 22 cycles of inner loop. These slot can be filled if the input & output stream get an additional vector RF and execute initial and final xor in parallel.
- Further reduce scalar core
- Merge PM and DM to 1 single memory. Limited performance penalty for scalar code.
- Burn PM into ROM. AES functions unlikely to change.
- Execute key expansion on scalar core at the expense of cycles (if keys do not change frequently)
- Adapt input/output streaming: bus-signals, DMA, Synchronization

AES Encryption & Decryption

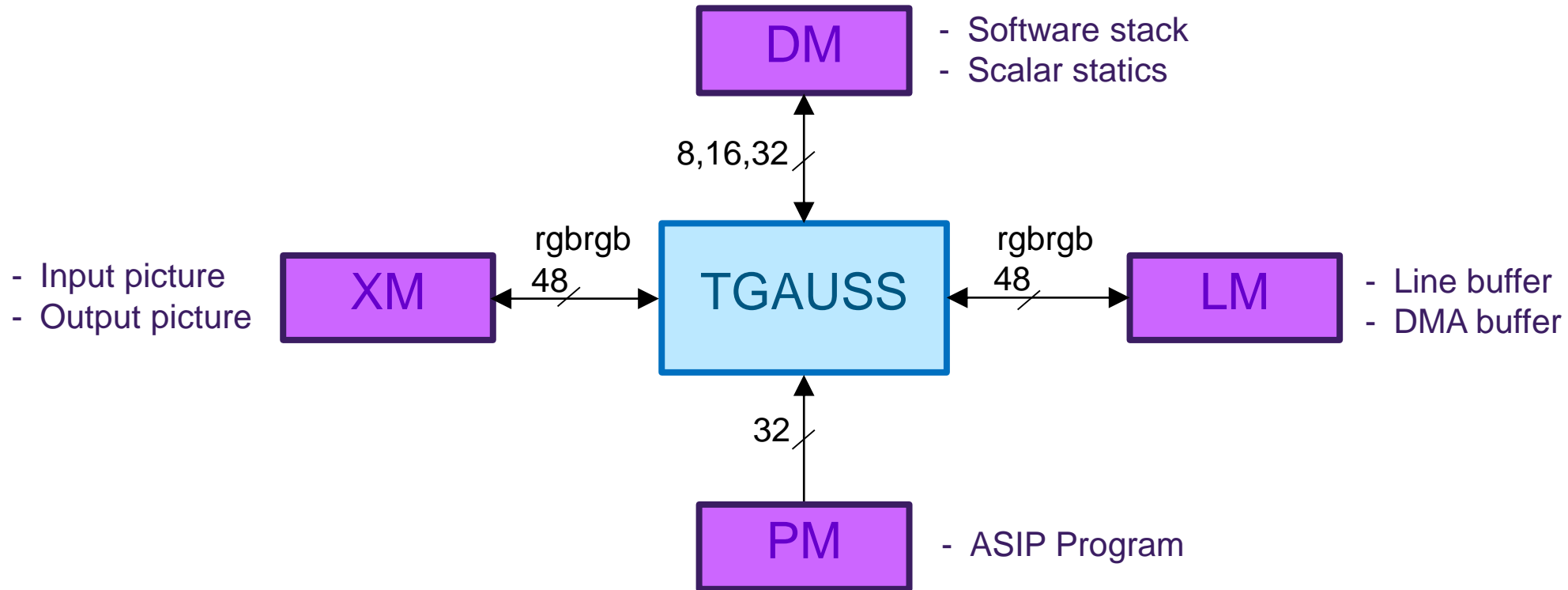
Alternative pipeline approaches

- Organize the two primitive operations in different ways in instruction set
 - Pipelined instruction:
 - Issue 2 primitives in a single instruction in consecutive stages
 - Vector RF needs 2 fields to keep the 2 stages busy.
 - Instruction level parallelism (as shown in this design):
 - Two primitives are executed in a separate instruction slot.
 - The scheduler can schedule the 2 parts in parallel; 2 separate registers can be used.

Tgauss Processor

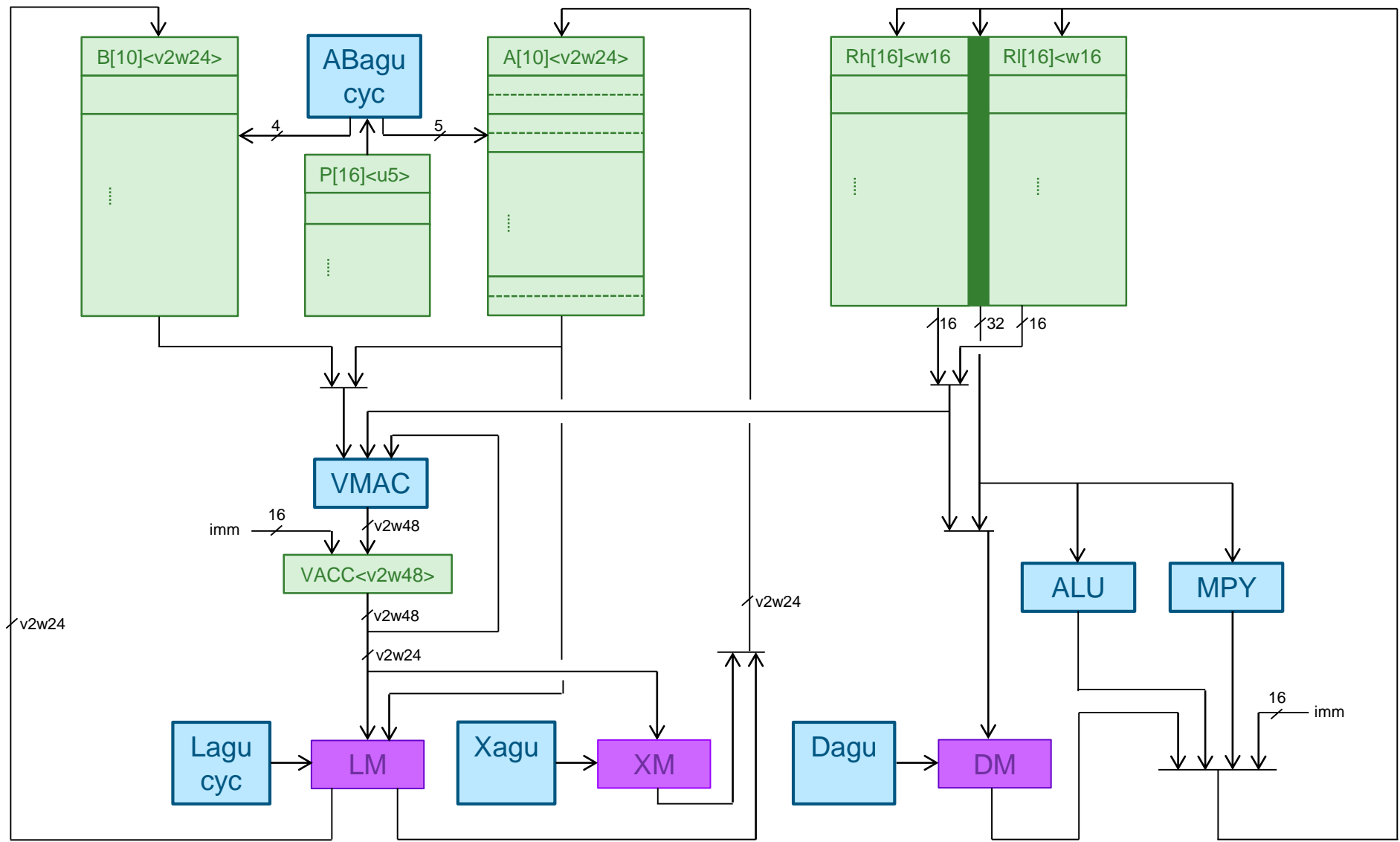
- Started from tlx = dlx with IF-ID-EX pipeline and 16 32-bit registers
- Extensions:
 - Vector data path to process six bytes (= two RGB pixels) in parallel:
 - vmul, vmac
 - vector register files A, B with cyclic buffer access
 - Separate vector memory for the input/output image.
 - Separate vector memory for buffering lines between the horizontal and vertical filter phases.
 - Split 32-bit register file into 16-bit Rh and Rl components for filter coefficient storage.
 - 3-Level hardware loop.
 - Memory interfaces to support delays in image memory response.
 - Two pipeline ex stages (E1, E2) for higher frequency synthesis

Top-level block diagram

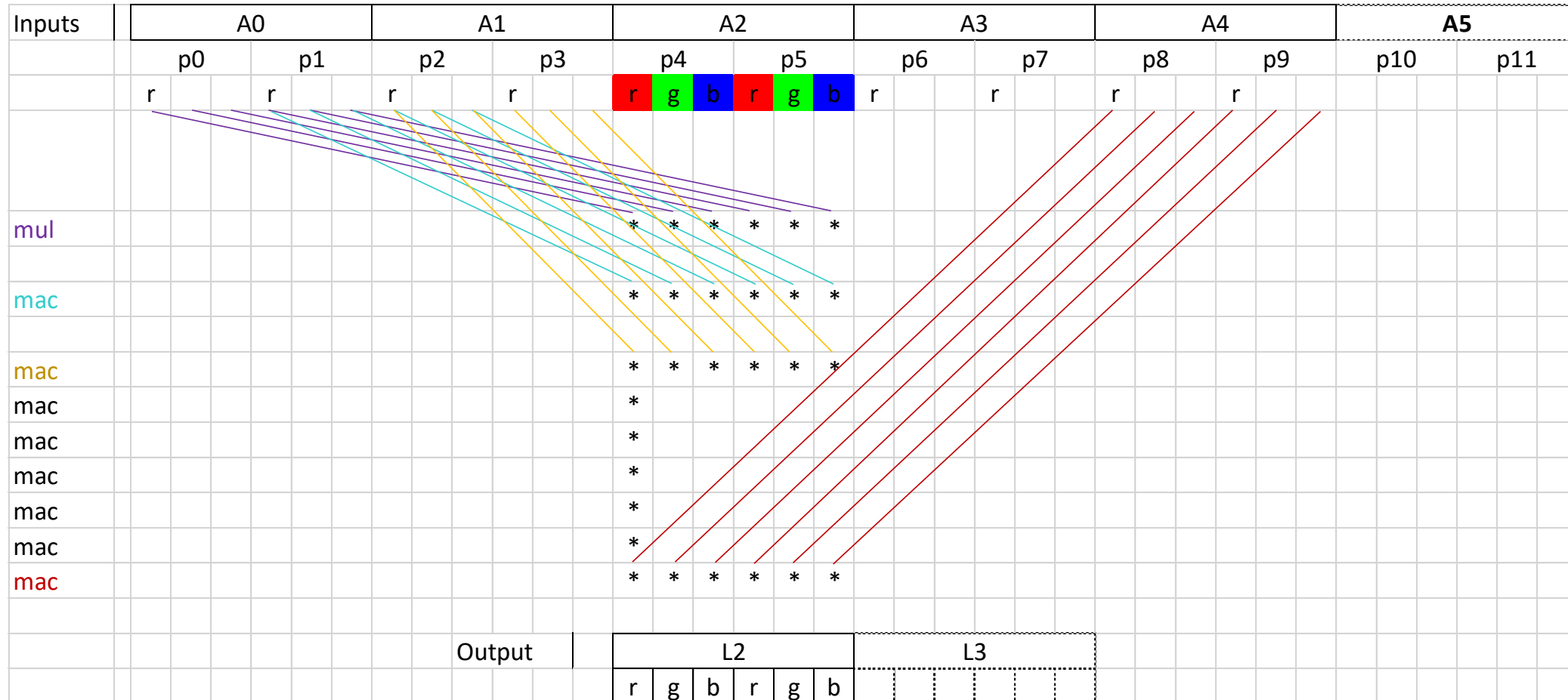


- Memory interfaces added for wait states

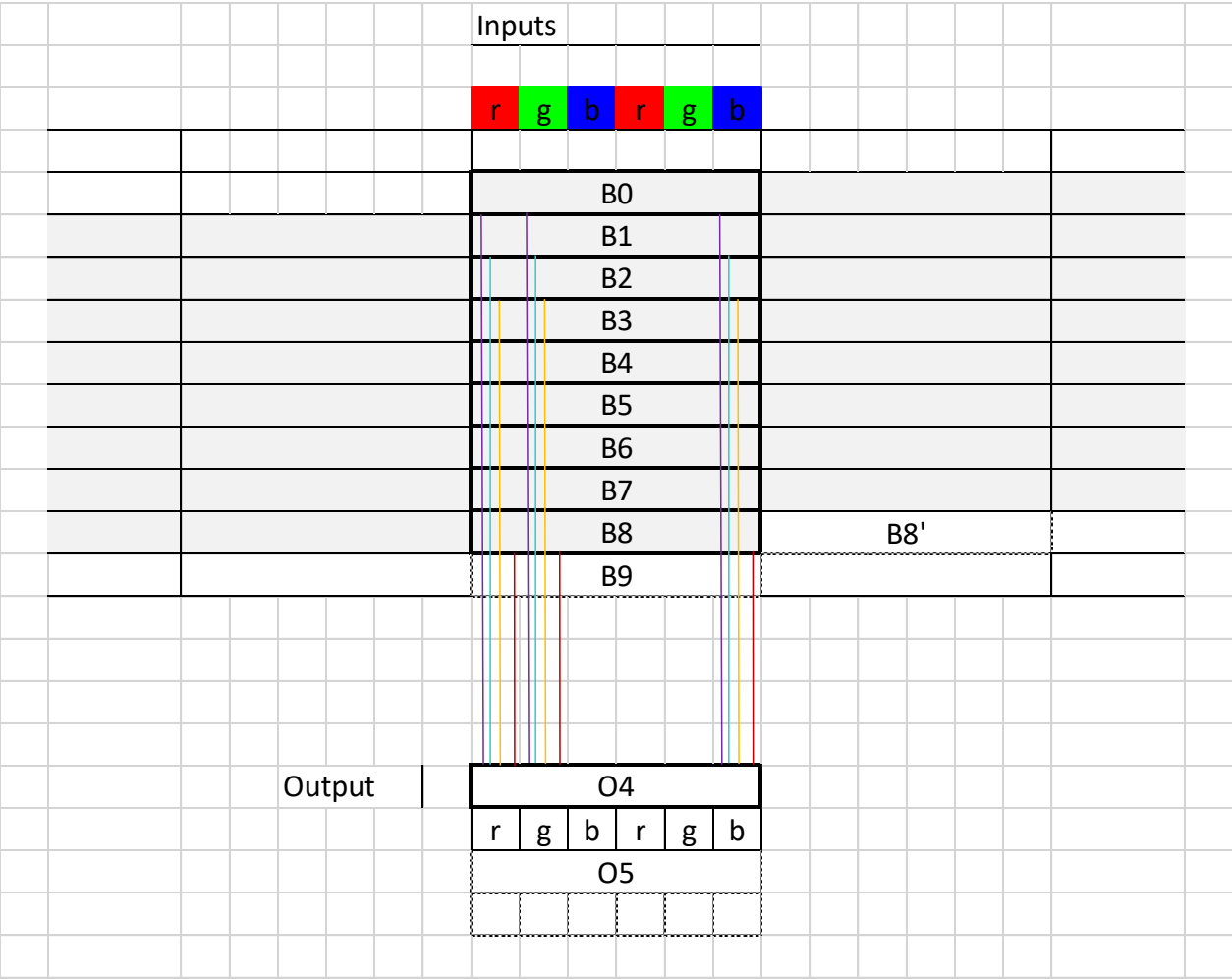
Tgauss data path



Separable filter – horizontal phase



Separable filter – vertical phase



Gaussian 9x9 filter – Version 1

- Separable filter: first pure horizontal filtering, then pure vertical filtering
- Version 1: re-use horizontal inputs, not vertical inputs

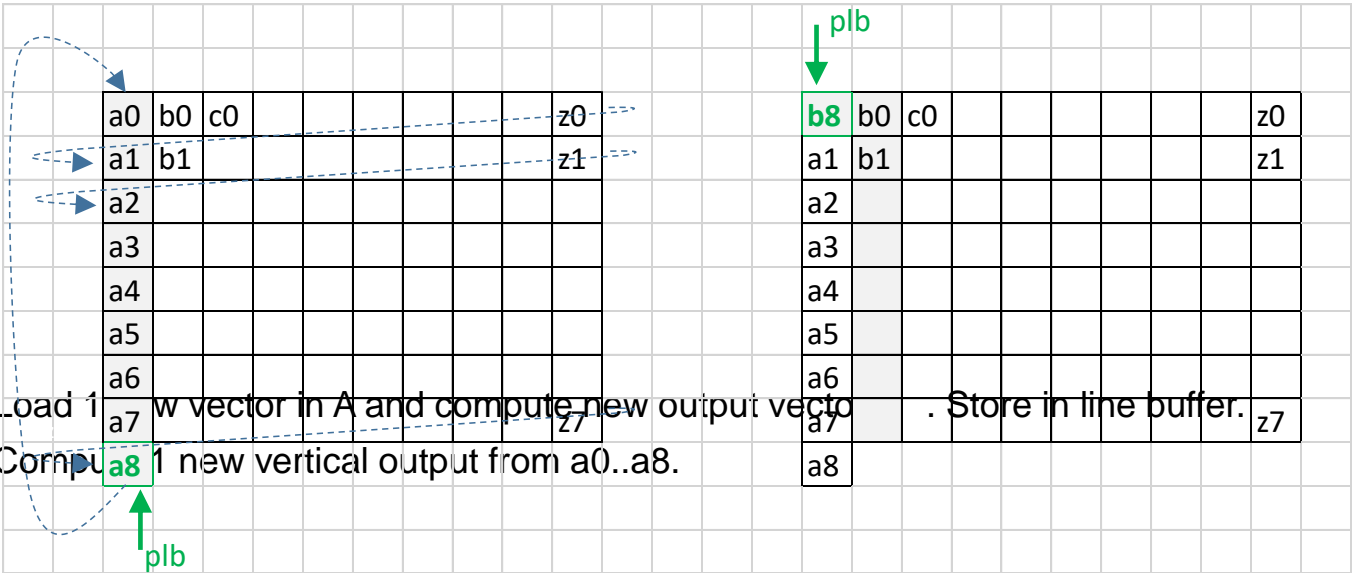
Line buffer = 8 lines + 1 vector of horizontal outputs

Reg A contains 5 vectors = 10 pixels

1. Init line buffer with horizontal filter outputs (vector contains 2 pixels =)



2. Load 1 row vector in A and compute new output vector . Store in line buffer.
3. Compute 1 new vertical output from a0..a8.



a8



Gaussian 9x9 filter – Version 1 (cont'd)

- Critical loop: 24 cycles

```
//      HORIZONTAL FILTER
104      0x0484008a  lx  A[p8+=2%p0@p4],XM[r9+=6]      // load new A sample
105      0x05040009  sxh acc,XM[r10+=6]              // vert result from previous iteration
to output
106      0x06e00042  vmulf acc,r17,A[p8+=1%p0@p4]
107      0x04c00043  vmacf acc,rh6,A[p8+=1%p0@p4]
108      0x06a00043  vmacf acc,r15,A[p8+=1%p0@p4]
109      0x05c00043  vmacf acc,rh14,A[p8+=1%p0@p4]
110      0x06c00043  vmacf acc,r16,A[p8+=1%p0@p4]
111      0x04600043  vmacf acc,rh3,A[p8+=1%p0@p4]
112      0x06600043  vmacf acc,r13,A[p8+=1%p0@p4]
113      0x04e00043  vmacf acc,rh7,A[p8+=1%p0@p4]
114      0x06400083  vmacf acc,r12,A[p8+=2%p0@p4]
.swstall __RAW__VACC_39
115      0x00000000  nop
116      0x07080005  sh  acc,LM[r4++%r15]            // horiz result to line buffer
.swstall __conflict__lm_addr_8
117      0x00000000  nop

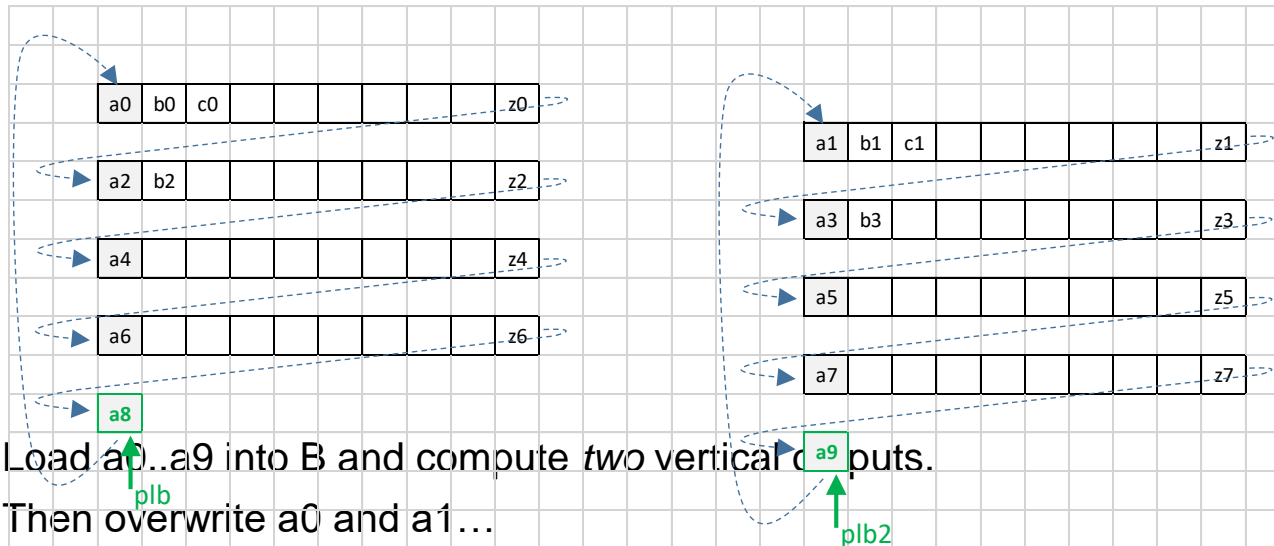
//      VERTICAL FILTER
118      0x07044007  ld  B[=p9+=0%p0@p4],LM[r4+=r8%r15]
119      0xcb002172  vmulf acc,r17,B[p9]  || ld B[=p9+=0%p0@p4],LM[r4+=r8%r15]
120      0xcb002063  vmacf acc,rh6,B[p9]  || ld B[=p9+=0%p0@p4],LM[r4+=r8%r15]
121      0xcb002153  vmacf acc,r15,B[p9]  || ld B[=p9+=0%p0@p4],LM[r4+=r8%r15]
122      0xcb0020e3  vmacf acc,rh14,B[p9] || ld B[=p9+=0%p0@p4],LM[r4+=r8%r15]
123      0xcb002163  vmacf acc,r16,B[p9]  || ld B[=p9+=0%p0@p4],LM[r4+=r8%r15]
124      0xcb002033  vmacf acc,rh3,B[p9]  || ld B[=p9+=0%p0@p4],LM[r4+=r8%r15]
125      0xcb002133  vmacf acc,r13,B[p9]  || ld B[=p9+=0%p0@p4],LM[r4+=r8%r15]
126      0xcb082073  vmacf acc,rh7,B[p9]  || ld B[=p9+=0%p0@p4],LM[r4++%r15]
.end_of_loop
127      0x06444003  vmacf acc,r12,B[=p9+=0%p0@p4]
```

Gaussian 9x9 filter – Version 1 (cont'd)

- Results:
 - 11.9 cycles/pixel
 - = 920 kcycles/frame (320 x 240)
 - = 9 ms/frame @ 100 MHz
 - 133 instructions
- Smaller filters need less cycles, e.g. 3x3 separable ~ 6 cycles/pixel
- Further speed-up possible by:
 - Increase vector size
 - Reduce loop size by more ILP, e.g. parallel store to XM, LM

Gaussian 9x9 filter – Version 2

- Compute two horizontal outputs, then two vertical outputs
- Requires two buffers in reg A double A size
- Use two half-size line buffers to avoid complex pointer management:



- Load a0...a9 into B and compute two vertical outputs.
- Then overwrite a0 and a1...

Gaussian filter – Version 2 (cont'd)

- Critical loop: 47 cycles executed ½ times

```
// TWO HORIZONTAL FILTERS
...
// TWO VERTICAL FILTERS:
139 0x04744007 ld B[=p9+=0%p0@p4],LM[r5+=r11%r12]
140 0x04880005 sh acc,LM[r6++%r12]
141 0x06844002 vmulf acc,r14,B[=p9+=0%p0@p4]
142 0x04b44067 ld B[=p9+=1%p0@p6],LM[r6+=r11%r12]
143 0xc8722223 vmacf acc,rh2,B[p9] || ld B[=p9+=1%p0@p6],LM[r5+=r11%r12] // mac and fill B
144 0xc8b22333 vmacf acc,r13,B[p9] || ld B[=p9+=1%p0@p6],LM[r6+=r11%r12]
145 0xc8722233 vmacf acc,rh3,B[p9] || ld B[=p9+=1%p0@p6],LM[r5+=r11%r12]
146 0xc8b223a3 vmacf acc,r110,B[p9] || ld B[=p9+=1%p0@p6],LM[r6+=r11%r12]
147 0xc87222a3 vmacf acc,rh10,B[p9] || ld B[=p9+=1%p0@p6],LM[r5+=r11%r12]
148 0xc8b223d3 vmacf acc,r113,B[p9] || ld B[=p9+=1%p0@p6],LM[r6+=r11%r12]
149 0xc84a22d3 vmacf acc,rh13,B[p9] || ld B[=p9+=1%p0@p6],LM[r5++%r12]
150 0xc88a2323 vmacf acc,r12,B[p9] || ld B[=p9+=1%p0@p6],LM[r6++%r12]
.swstall __RAW_VACC_39
151 0x00000000 nop
152 0x07040009 sxh acc,XM[r14+=6]
153 0x068440a2 vmulf acc,r14,B[=p9+=2%p0@p6]
    // REUSE B content
154 0x04444063 vmacf acc,rh2,B[=p9+=1%p0@p6]
155 0x06644063 vmacf acc,r13,B[=p9+=1%p0@p6]
156 0x04644063 vmacf acc,rh3,B[=p9+=1%p0@p6]
157 0x07444063 vmacf acc,r110,B[=p9+=1%p0@p6]
158 0x05444063 vmacf acc,rh10,B[=p9+=1%p0@p6]
159 0x07a44063 vmacf acc,r113,B[=p9+=1%p0@p6]
160 0x05a44063 vmacf acc,rh13,B[=p9+=1%p0@p6]
161 0x06444063 vmacf acc,r12,B[=p9+=1%p0@p6]
162 0x00406040 p9+=1%p0@p6
163 0x07840009 sxh acc,XM[r15+=6]
.end_of_loop
.swstall __conflict__xm_addr_9
164 0x00000000 nop
```

Gaussian 9x9 filter – Version 2 (cont'd)

- Results:
 - 11.7 cycles/pixel
 - = 899 kcycles/frame (320 x 240)
 - = 9 ms/frame @ 100 MHz
 - 176 instructions
- LM load traffic reduced by factor 2
- Reduce cycles: same as Version 1

IO Interface Examples

- Tmicro_tcm
 - Tightly coupled data and program memory
 - Memories can be access by external blocks
 - Policies:
 - PM: always responsive to processor requests
 - DM: req – ack signals, core may need to be stalled
- Tmicro_cache
 - DM cache
 - AXI interface

Release History

What's New in the M-2017.03 example models

- Tmicro:
 - Support of the libc++lite library
- Tdsp:
 - Support of LLVM front end
- DLX Family
 - New PSTALL model: implements precise stalling of the pipeline (stalling in a stage after the decode stage)
 - New MCLD model: IO interface that demonstrates a multi-cycle load into a wide register
 - New SHA256 model: acceleration of the SHA256 hashing function on TLX
- Tvliw
 - Add instruction truncation as alternative to elongation to lib2
 - Use compiler property `nop_instruction_class` instead of Preferred class Setting.
- SystemC example
 - New examples of cycle accurate and TLM integration in OSCI and Virtualizer
- ASIP Programmer Lab
 - Hands-on tour of the features of ChessDE, based on Tmcu and the JPEG application

What's New in the L-2016.09 example models

- Generic regression suite and Tcl driver scripts
 - Generic (processor independent) version of the C regression tests, which can be instantiated according to the type system of the target processor
 - New Tcl driver scripts to run the regression tests
 - See examples/regression/HOWTO.txt
- LLVM compiler run time libraries
 - There is a new version, 3.8.0 of the LLVM compiler run time and C++ libraries
- Instruction accurate ISS projects
 - The new IA-ISS technology is demonstrated on tmicro, tinycore, tdsp, tcom8
- Tmicro
 - The LLVM front end has been enabled, including a compiler-rt library
 - Generation of SystemVerilog/UVM classes for random instruction generation with the Risk tool
 - SystemC/TLM2 supported for instruction accurate ISS
 - Support for integration into Eclipse

What's New in the L-2016.09 example models

- Tnano
 - Optimization of multiplication (single primitive and compiler header improvements)
- RISC-V
 - New example processor model, implementing the open source RISC-V ISA
- Tmcu
 - More efficient mapping of > and >= operators onto gts, ges, gtu and geu
- Tvec lib6
 - A new variant, lib6 has been added. This variant has following features
 - Support 4x32-bit vlong type next to 8x16-bit vint type
 - Map both types on the same vector data path sharing hardware as much as possible
 - Vector operations for both types can be predicated
 - Scalar-based vector addressing

What's New in the L-2016.03 example models

- Tinycore:
 - Introduced uniform model for call instructions and compact control signals
- Tmicro
 - Latching and edge detection of interrupts moved to IO interface
 - Added memory address step parameter to debug client
- Tmcu
 - Changed constant load for R from 18 bit signed to 20 bit signed
 - Changed immediate stack pointer load from 18 bit unsigned to 24 bit unsigned
- DLX-like processors
 - Tightened hardware stalls for div instruction
 - Updated assembly init code
- Tvliw
 - Introduced uniform model for call instructions and compact control signals
 - Minor model updates

What's New in the K-2015.12 example models

- Tmcu:
 - Full-featured C/C++ stack based on libcxx and newlib library
 - Lightweight C/C++ stack based on libcxx-lite and runtime library
- Tmicro:
 - Improvements of the processor verification flow
- Tnano:
 - Fast context switching for interrupts based on shadow registers was added as an option
- Tvec5: Tvec lib5 is a variant of the Tvec model with
 - Full support for the sequential execution of control flow instructions
 - A new vector addressing scheme where each lane can access the complete vector memory
 - Support for OpenCL compilation ☐
- VLX:
 - Variant of the DLX model, with a limited set of SIMD instructions
 - Used to demonstrate SIMD features of LLVM front end

What's New in the K-2015.06 example models

Following example processor models have been added or modified in K-2015.06:

- Tmcu: 32 bit microcontroller
 - New configurations Release_LLVM and Debug_LLVM to enable the LLVM front end.
 - Updated compiler-rt project (see Readme-LLVM.txt).
 - New unsigned min and max instructions.
- libs-K-2015.06.zip
 - Updates of LLVM compiler-rt required for LLVM front end.
- runtime.zip
 - Updates of Chess runtime library.

All other example models are unchanged and retain the J-2014.03 version number.