FastMATH™ Overview

October 28, 2002

Overview

- Market and economics
- Intrinsity technology
- Architecture decisions leading to matrix unit
- Micro-architecture
- Programming examples
- Performance

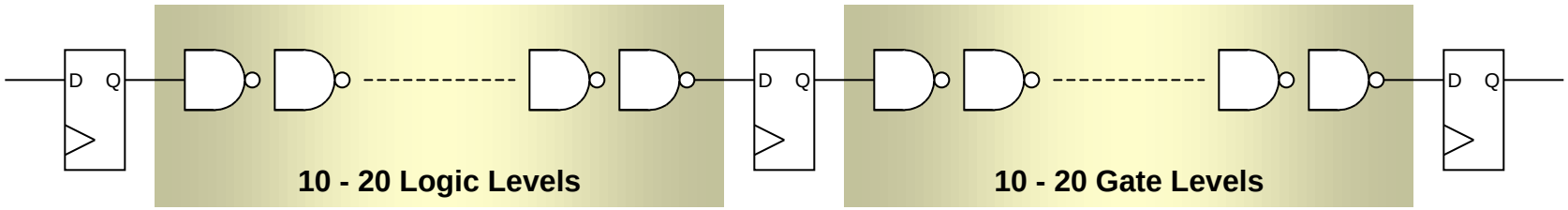
Challenge for a new design

- Intrinsity has Fast14 design methodology which is 3-5x faster than synthesized logic
- Intrinsity has small team compared to Intel, AMD and others
- Customers want standards, ease-of-use, performance

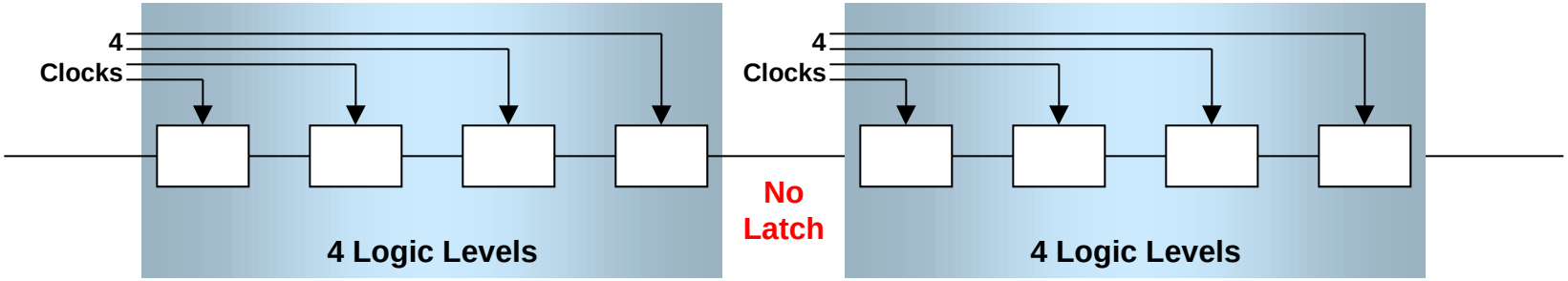
- Investors want 10X advantage over competitors

Static v. Fast14 Technology

Traditional Static Logic



NDL™ Dynamic Logic



Dynamic gates for speed.

More functions per gate (more complex Boolean functions possible).

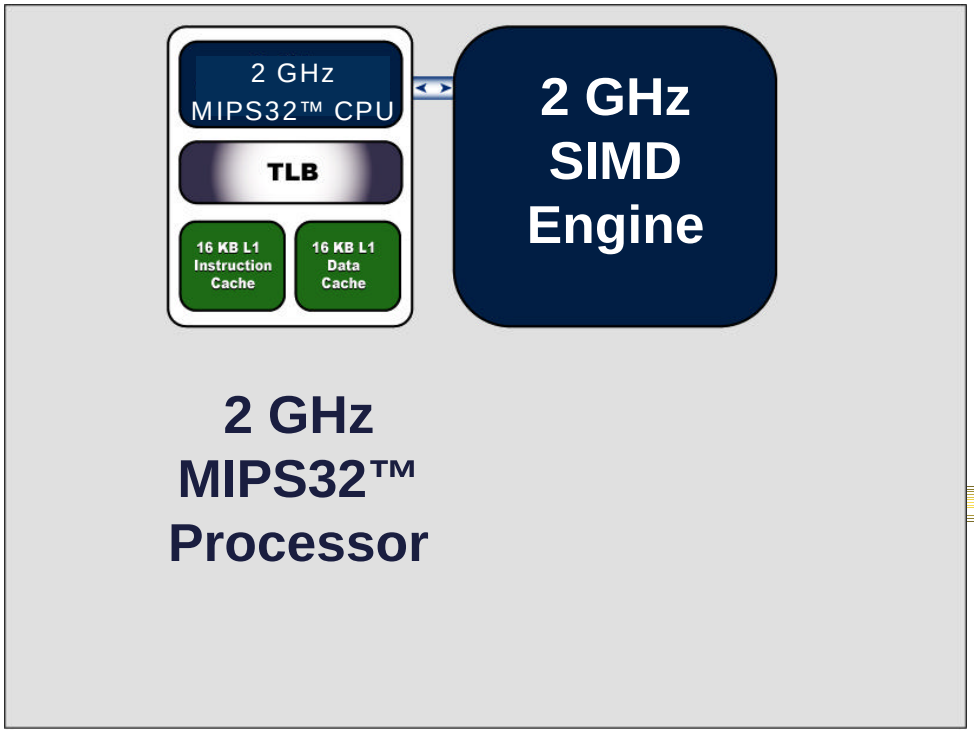
No latches needed in the pipeline.

Faster throughput, nearly 3x static logic.

Product strategy

- Fast14 Circuits are best at computational problems: focus on massive data parallelism in communications applications
- Parallel processing solutions:
  - VLIW: hard to program; little parallelism in ISA
  - SMP: synchronization overhead; complex to build
  - Superscalar: limited bandwidth; complex to build
  - SIMD: ideal for small kernel operations on a large sequence of data
- Customer still want ease of programming
  - Use MIPS™ ISA as industry standard with plenty of support
  - Integrate RISC core with a SIMD unit

Basic Product Concept



MIPS32 Core and a wide SIMD engine

- 16-32 SIMD elements
- Decode done by MIPS core
- 2 GHz operation for both

SIMD Unit Influences

Communications applications operate on

- Vector data
- Matrix data; including large matrices

Imaging applications operate on 4x4 data

Inter-element communications is frequent

16/32 way SIMD parallelism is the knee of the curve

Communications restricted by wire delays

Memory accesses can be up 50% of the code stream

Memory accesses require high-bandwidth

Low latency loads reduce register pressure and ease programming

Architecture vs. physical tradeoffs

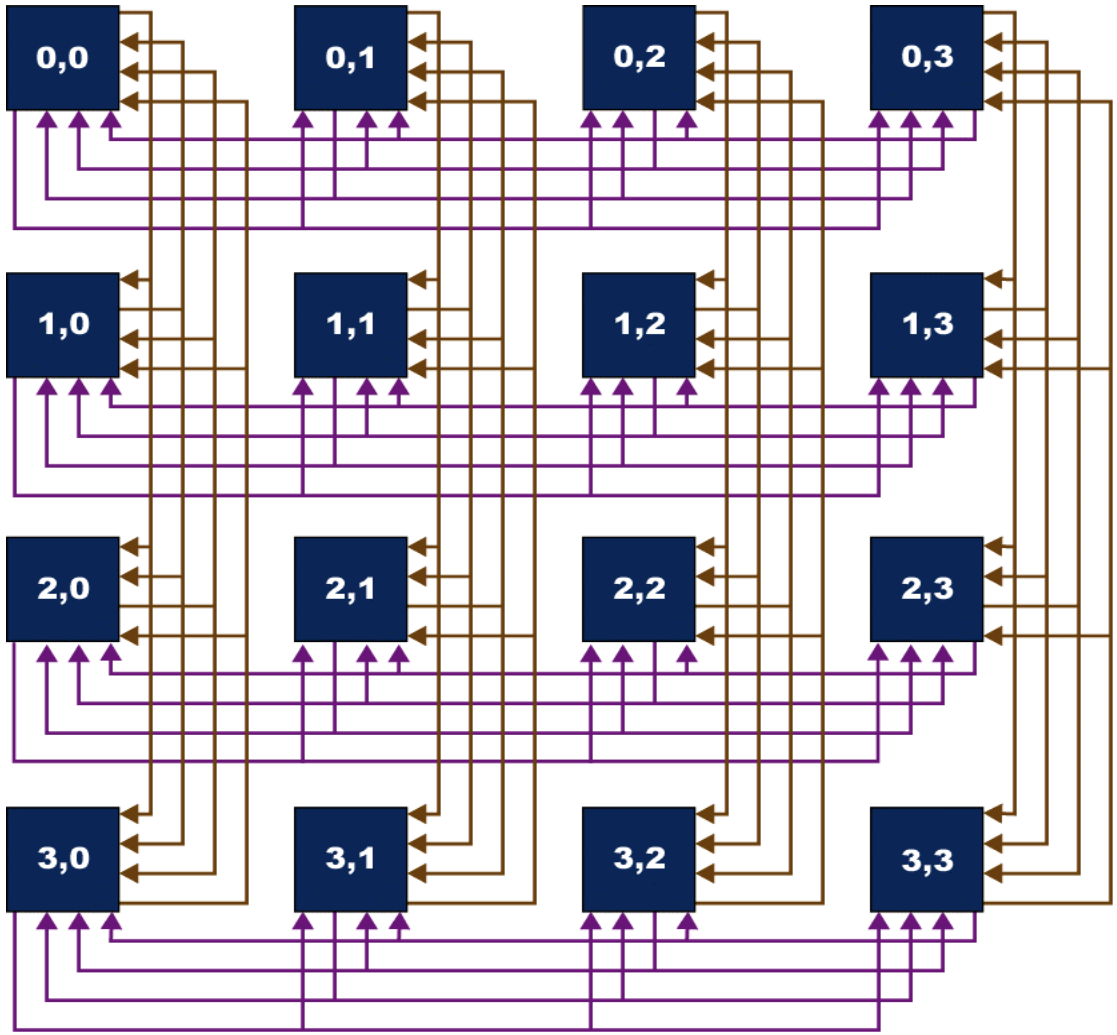
2GHz and physical constraints in .13µ CMOS led to

- 1 cycle adders must be 32b or smaller
- 1 cycle broadcast between elements limits distance to ~2mm
- Each matrix element estimated at .75x.75mm
- Memory paths limited to 1024 wires for 1 cycle accesses

These constraints and previous ones led to

- 16 matrix elements arranged 4x4 array
- Each element operates on 16b or 32b

Matrix Processing Elements Connections



- **Each element can broadcast a value to all the other elements in its row and column**
- **Each element can use operands from local registers or from a broadcast during each operation**

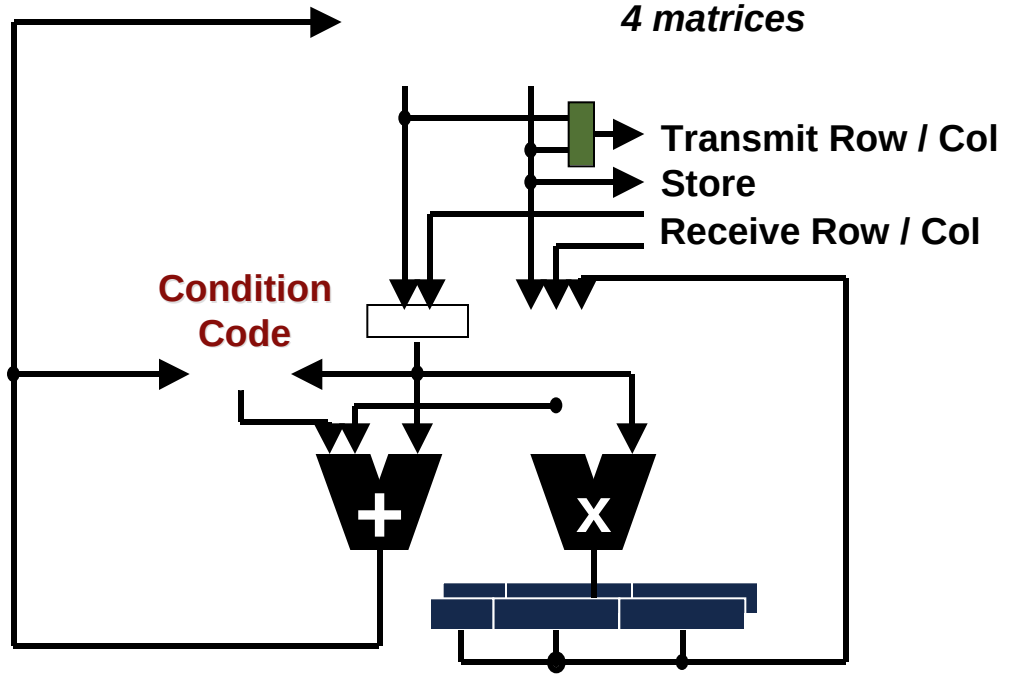
Instruction Set Architecture

Operation Classes

load	load	store
ALU	add subf and or nor xor andn set popc	cmp shifts mflo mfhi select pack unpack
simple broadcast	slc src sur sdr	selectrow selectcol
complex broadcast	alignword transpose	rotate block4
broadcast/operate	matmul table	sumrow sumcol
multiply	mul	madd msub

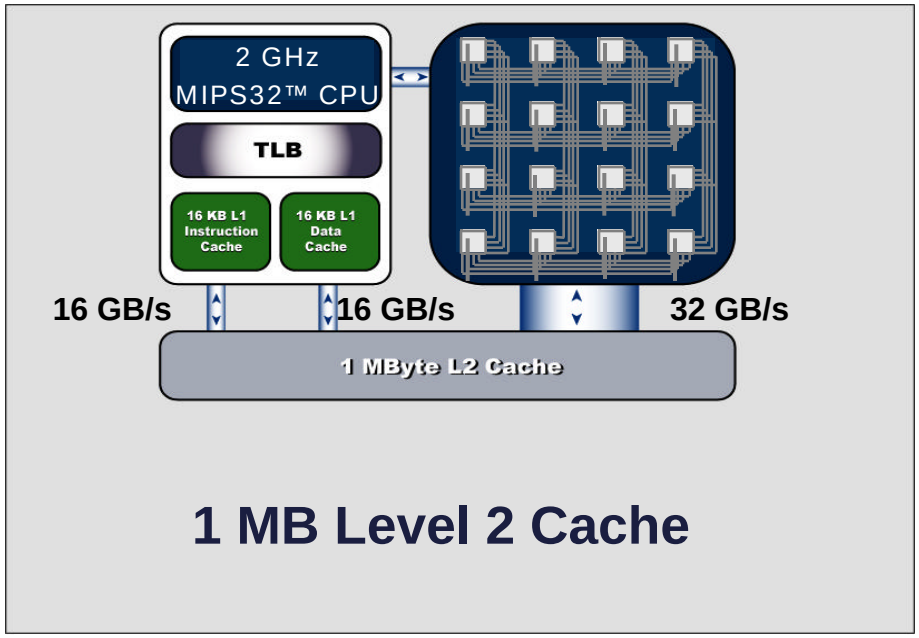
16 x 32-bit Register File

Complete array stores 16 32-bit 4 x 4 matrices



Dual independent 40-bit accumulators

Matrix Unit Memory Support

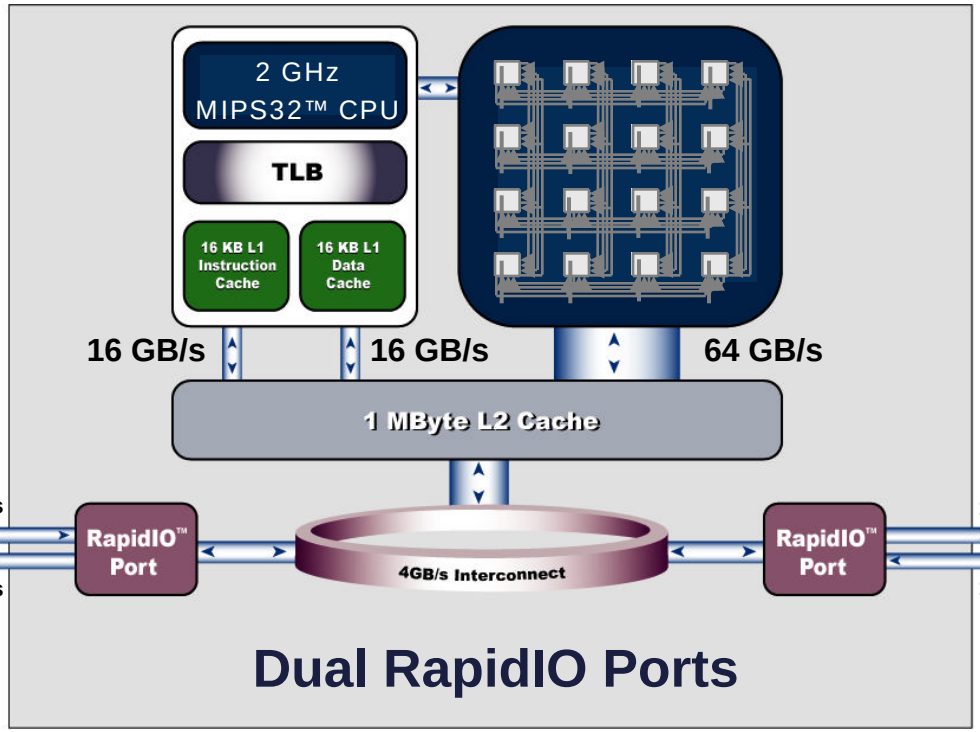


1 MB Level 2 Cache

1 MB Level 2 cache provides direct access to large data sets

- Is in essence a 1 MB L1 cache for matrix unit
- Loads appear to be 1 cycle
- 4-way set-associative
- Operates at 1 GHz
- 32 GB/s sustained
- Configurable as cache or SRAM in 512 KB increments
- Coherent to on-chip memory and I/O

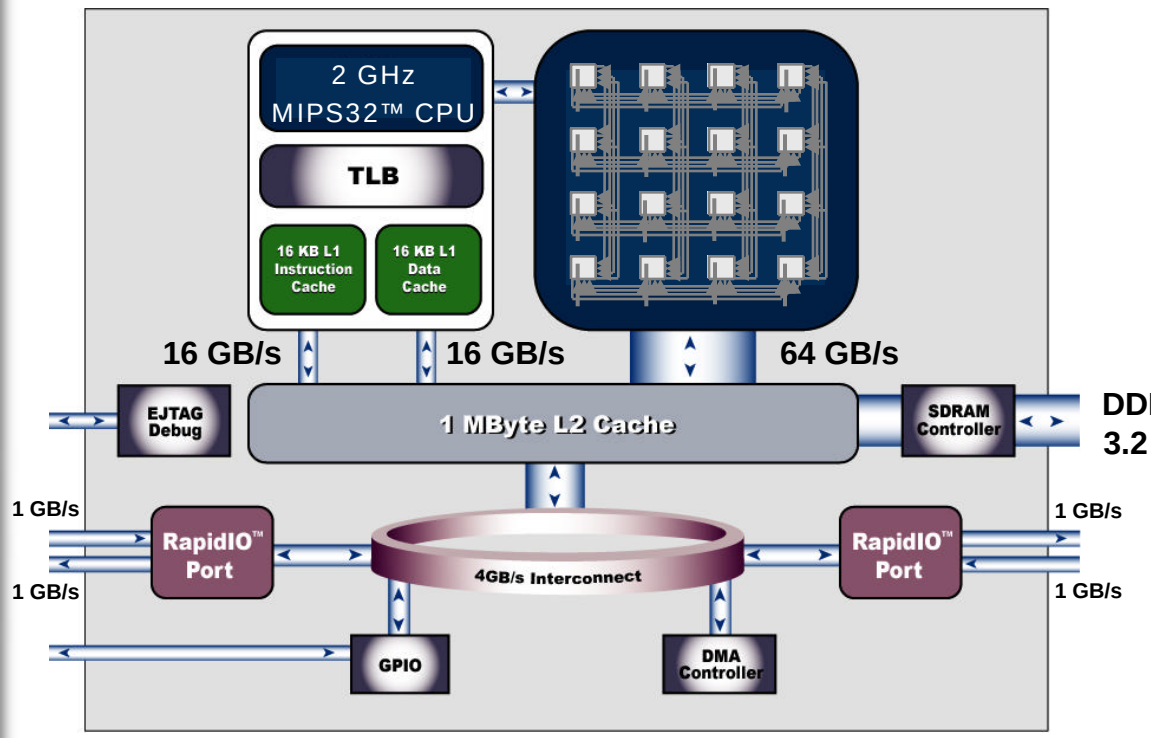
FastMath™ Adaptive Signal Processor



Dual RapidIO™ ports provide high-bandwidth system I/O

- Standard fabric interface with broad industry support
- 8-bit LVDS interface, up to 500 MHz operation
- Data transferred on each clock edge (DDR)
- Simultaneous 1 GB/s input and output per port
- Total bandwidth of 4 GB/s

FastMath™ Adaptive Signal Processor



Memory controller

- 64-bit, DDR-400
- 3.2 GB/s

General Purpose I/O

- 66 MHz
- 8- or 32-bit interface to ROM, Flash, SRAM, etc.

DMA engine reduces CPU overhead

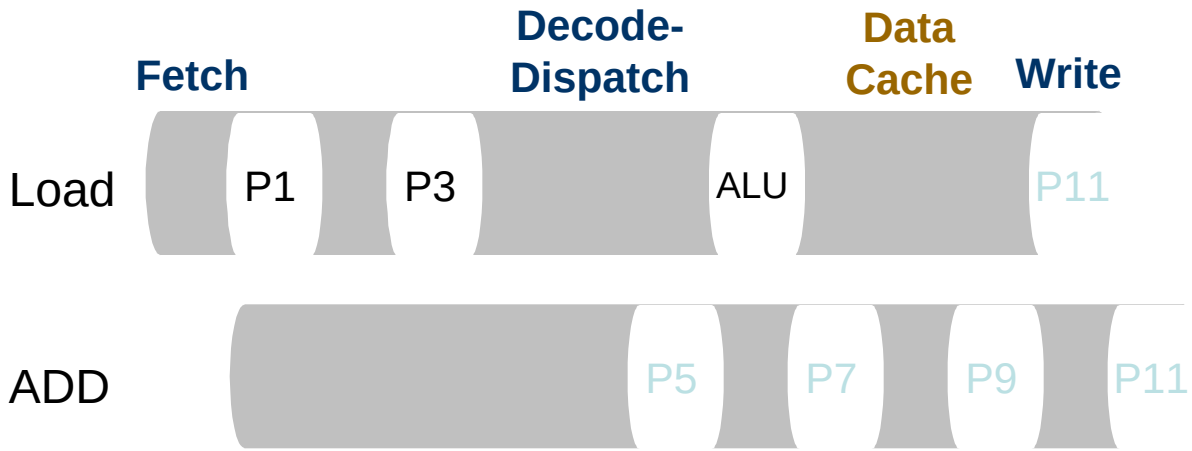
- 2-channel + inbound RapidIO port interface
- Descriptor-based
- Global shared memory

CPU Pipeline

	Fetch	Instruction Cache		Decode-Dispatch			Write
ALU Operation	P1	P3	P5	P7	P9	P11	

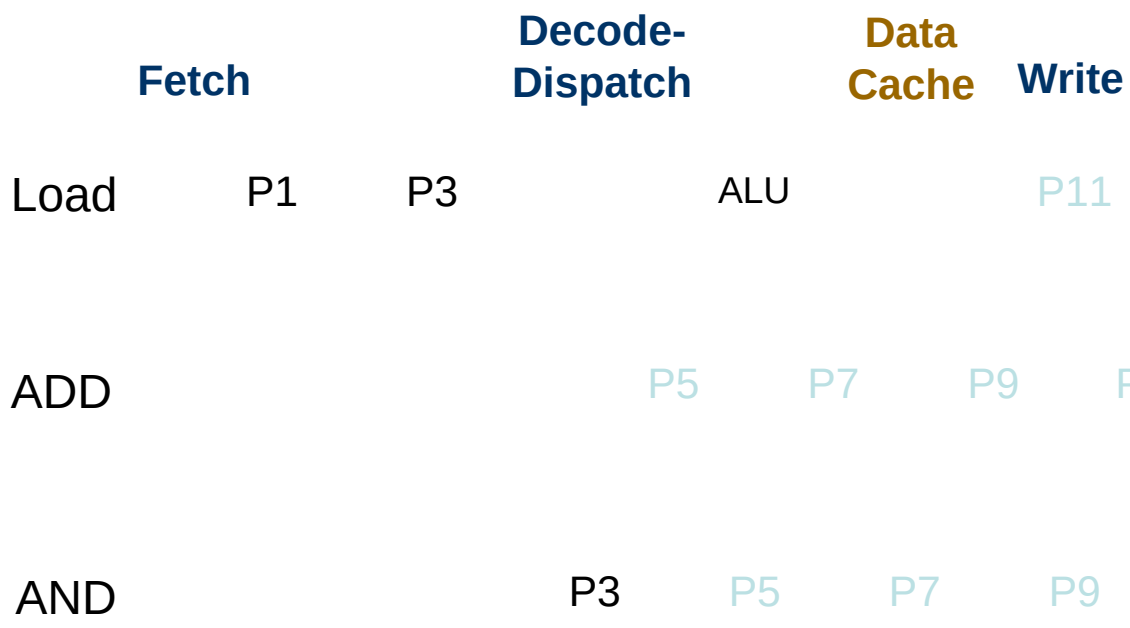
- **12 pipeline stages**

CPU Pipeline



- **12 pipeline stages**
- **ALU is staged for a 1 cycle load-to-use latency**

CPU Pipeline



- **12 pipeline stages**
- **ALU is staged for a 1 cycle load-to-use latency**
- **ALU operations complete in 1 cycle and run back-to-back**

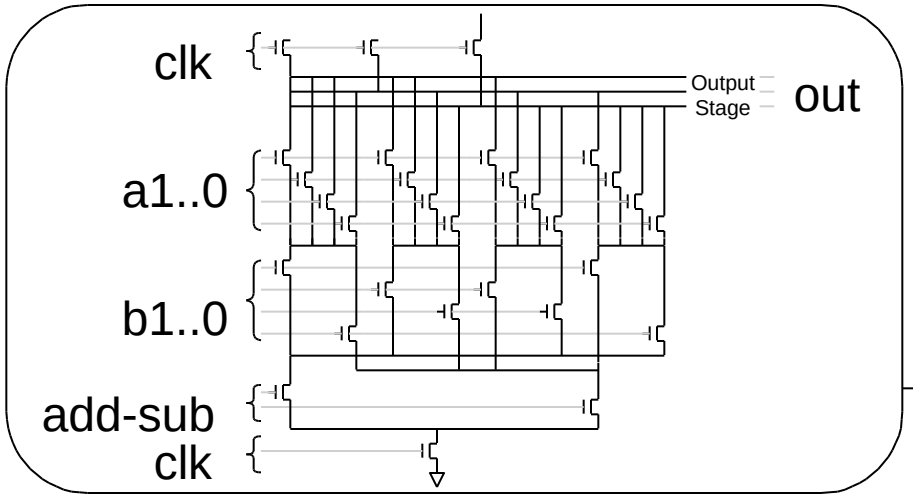
Fast14™ Technology for 2 GHz

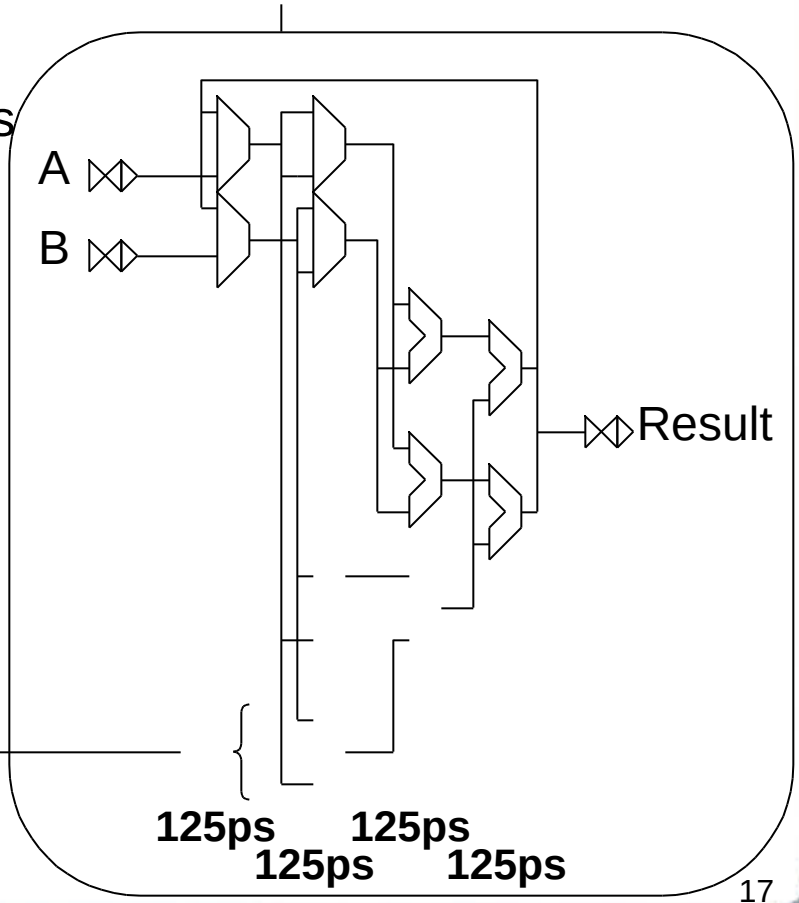
Fetch **Decode-Dispatch**

P1 P3 P7 P9 P11

Single-cycle ALU in 4 gate delays

High-speed, high-function NDL™ gates





Matrix Unit Instruction Classes

Memory Access

load.m	$m0,0($r1)	Load a 64-byte matrix
store.m	$m0,0($r1)	Store a 64-byte matrix

ALU / Logical / Comparison

add.m.m	$m0,$m1,$m2	Add 2 matrices element-wise
cmple.s.m	$mcc1,$m0,$r3	Compare each element of matrix 0 with scalar in r3

Multiply/Accumulate

mullh.m.m	$mac0,$m0,$m1	Mul element-wise low halfword of m0 w/ high of m1
maddhh.m.m	$mac1,$m0,$m1	Mul element-wise high halfwords of m0 and m1 and accumulate

Inter-Element Movement

trans.m	$m0,$m1	Transpose of elements in m1 to m0
block4.m	$m0block	Rearrange m0..m3 from 1x16 vectors to 4x4 matrices
srcol.i.m	$m0,$m1,0	Shift matrix m1 right by a column and fill w/ 0's

Inter-Element Computation

matmulhl.m.m	$mac1,$m1,$m2	Matrix multiply high halfwords of m1 and low of m2
sumrow.m	$m0,$m1sum	Elements of m1 across rows and store sums in m0

Application Example: Wireless Basestation

Goal: Increase system capacity through improved spectral efficiency

Technique: Parallel User Interference Cancellation

- Use knowledge of current CDMA spreading codes to determine correlated interference between users

Matrix-Matrix Multiplication

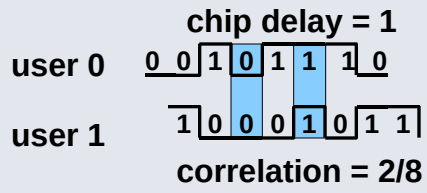
$$Y = B \times R$$

Y = Improved user stream (Nbits × Nuser)

B = Hard decision of user stream (Nbits × Nuser)

R = Multi-user correlation matrix (Nuser × Nuser)

Compute cross-correlation of all users' spreading codes:



Iteratively use correlation values to simultaneously cancel out interference between users:

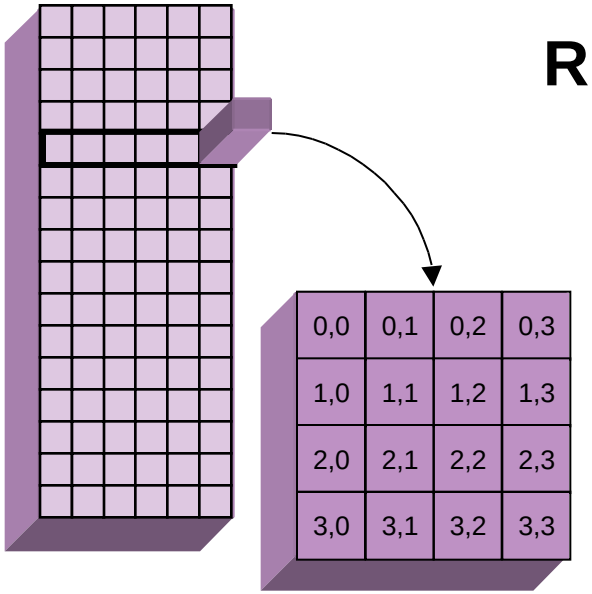
user 0 0.6 - (2/8 * -0.4) = 0.7 - (2/8 * -0.55) = 0.83

user 1 -0.4 - (2/8 * 0.6) = -0.55 - (2/8 * 0.7) = -0.73

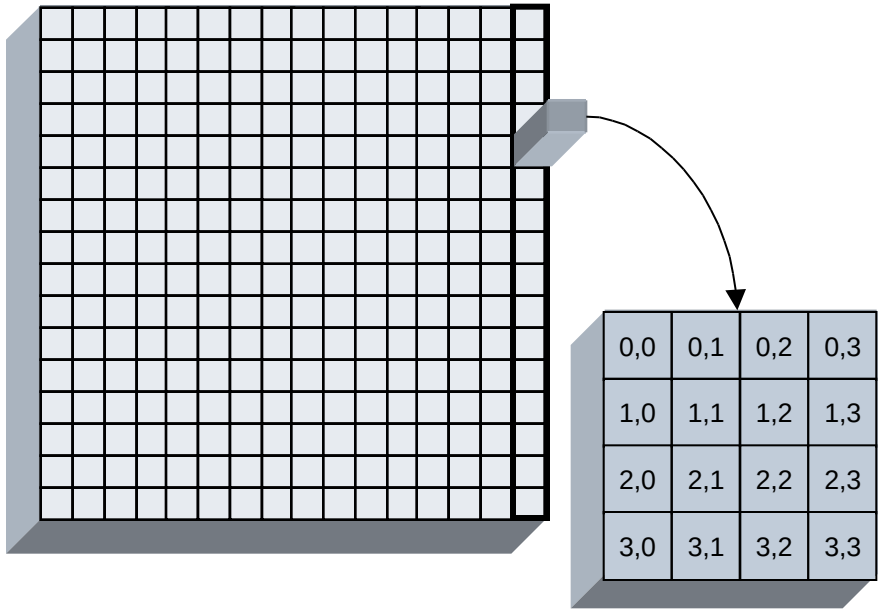
"Blockifying" Matrix Multiply into a 4x4 Sub-matrix

$$Y = B^T \times R$$

B =

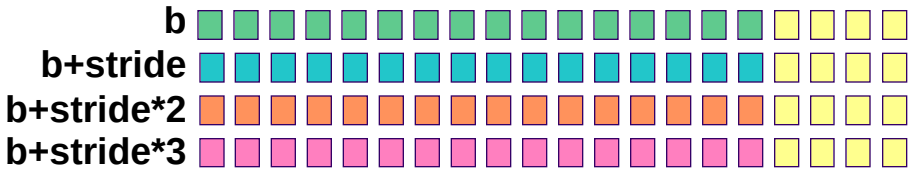


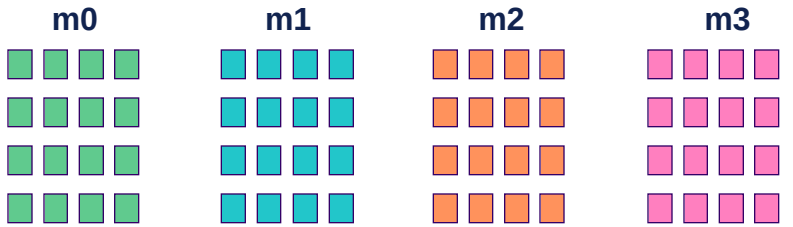
R =

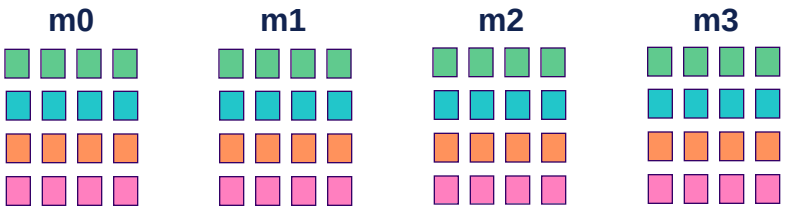


- Large matrix-multiply can be broken down into multiplications of smaller sub-matrix
- The ***FastMATH***™ processor provides an intrinsic 4 x 4 matrix-multiply operation as a building block

Converting to Block Form







Large matrix stored in memory in normal form

load.m gets 16 contiguous words of memory (vector), not a 4 x 4 submatrix

Solution:

1) Load 4 matrices that are separated by the large matrix stride (must be 64-byte aligned)
2) Perform block4 on the matrices (specified by first matrix: m0)
3) matrices are now in block form

```
load.m    m0, 0(b)
load.m    m1, stride(b)
load.m    m2, 2*stride(b)
load.m    m3, 3*stride(b)
block4    m0
```

Matrix-Multiply Instruction

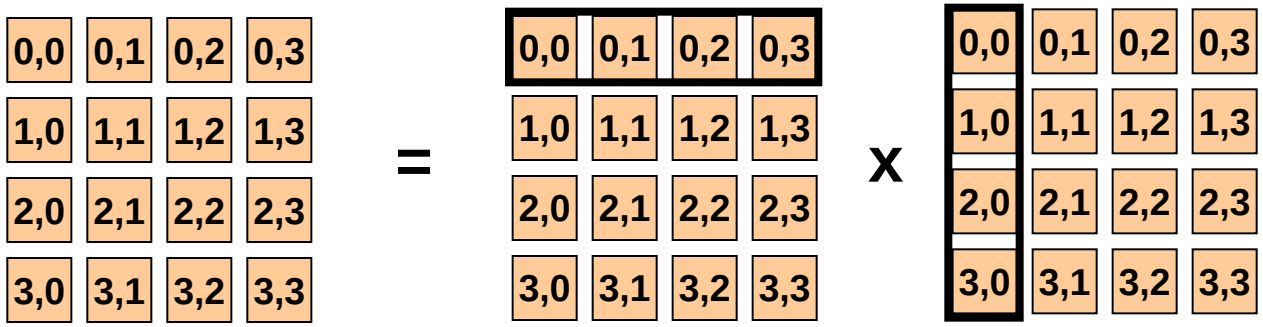
Matrix multiply of two 4x4 sub-matrices

- 1 instruction
- 4 cycles (2 ns @ 2 GHz)

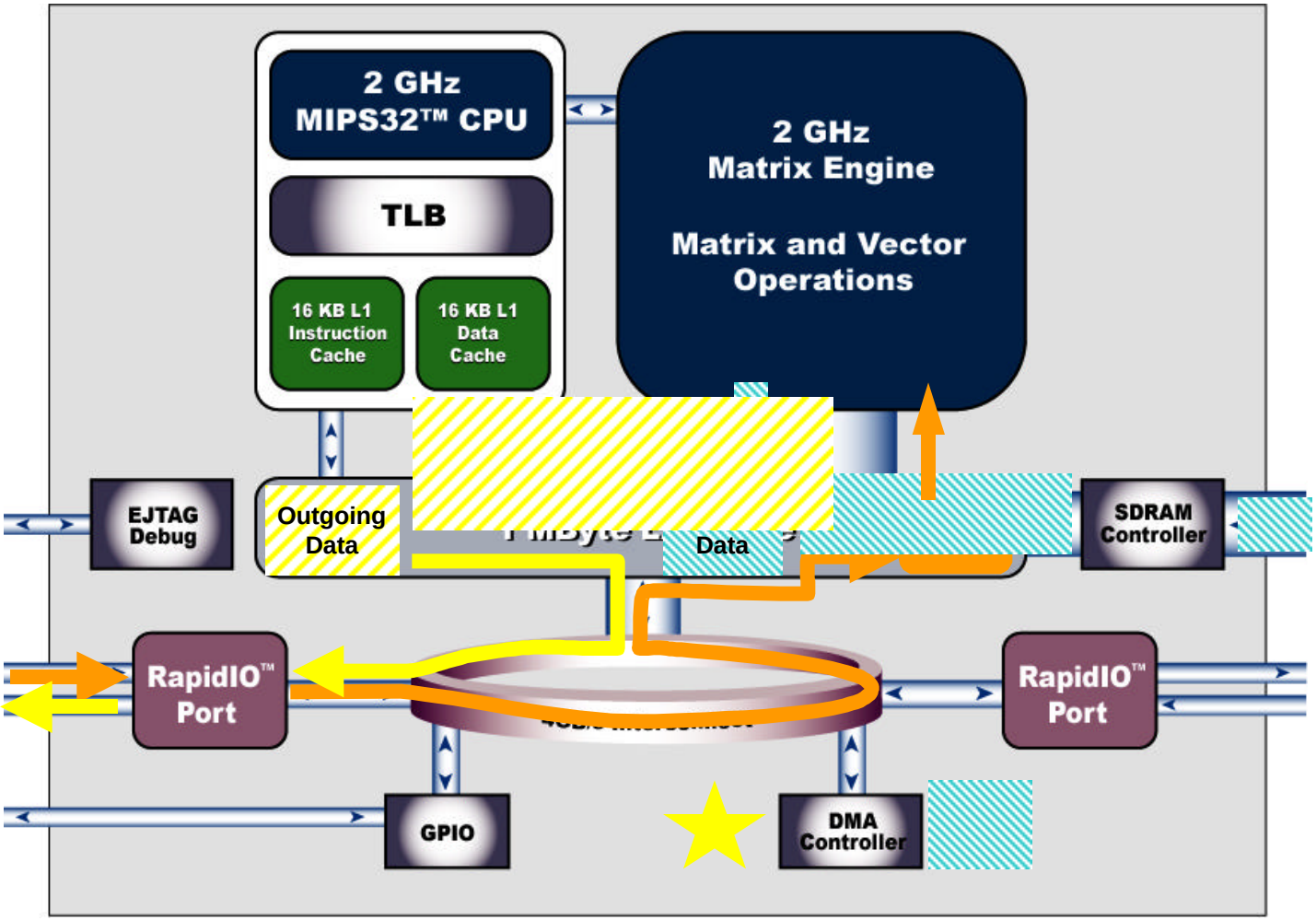
$$\sum_{k=0}^{3} M0_{0k} \times M1_{k0}$$

Matmulhh.m.m M2,M0,M1

```
for i = 0 to 3
    for j = 0 to 3
        sum = 0
        for k = 0 to 3
            sum = sum + M0(i,k) × M1(k,j)
        M2(i,j) = sum
```



FastMATH™ Adaptive Signal Processor

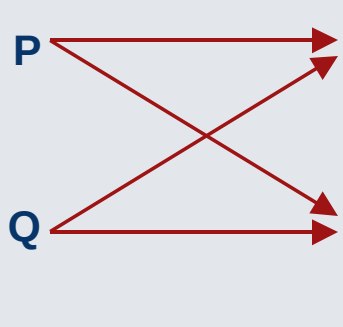


Radix-2 DIF FFT Code Example

Radix-2 "Decimation in Frequency" FFT

- With 16-bit complex (interleaved real and imaginary) data

Butterfly Operation

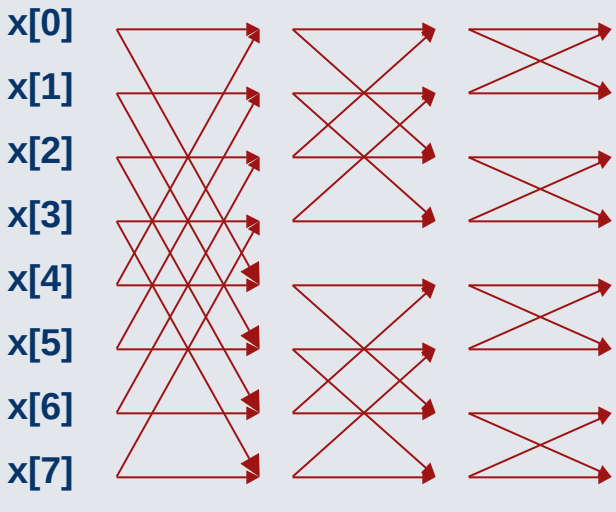


P + Q
Complex addition

w * (P - Q)
Complex multiply and subtraction

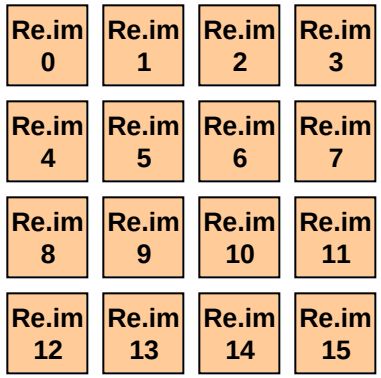
w = "twist" or "twiddle factor"

$\log_2 N$ stages of butterflies:

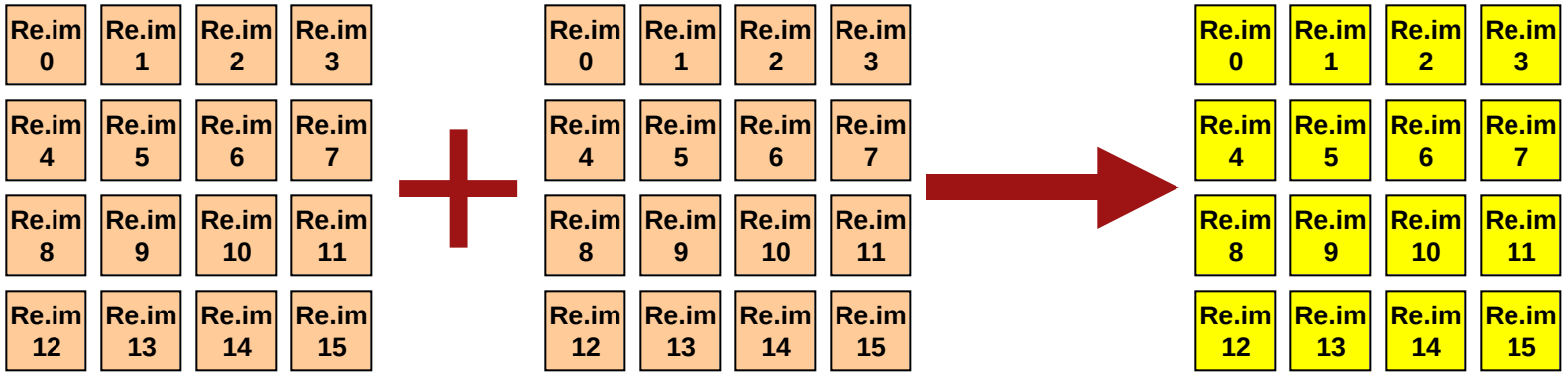


FastMATH Register File Usage

Each matrix register can hold 16 complex values, with the real and imaginary values kept in 16-bit partitions in a 32-bit element value:



In all but last four stages, a butterfly works on paired elements from two different matrices



Butterfly Implementation

Complex multiply in butterfly:

re = re1*re2 - im1*im2
im = re1*im2 + re2*im1

High/low variants of multiply used to form four products, and the accumulators to form the required sum and difference:

mulll.m	acc0, cplx1, cplx2
msubhh.m	acc0, cplx1, cplx2
mullh.m	acc1, cplx1, cplx2
maddhl.m	acc1, cplx1, cplx2
mflo.m	re, acc0
mflo.m	im, acc1

Pack word operation packs back into partitioned 16-bit real/imaginary format and rescale results:

packhhi.m.m	cplx, im, re

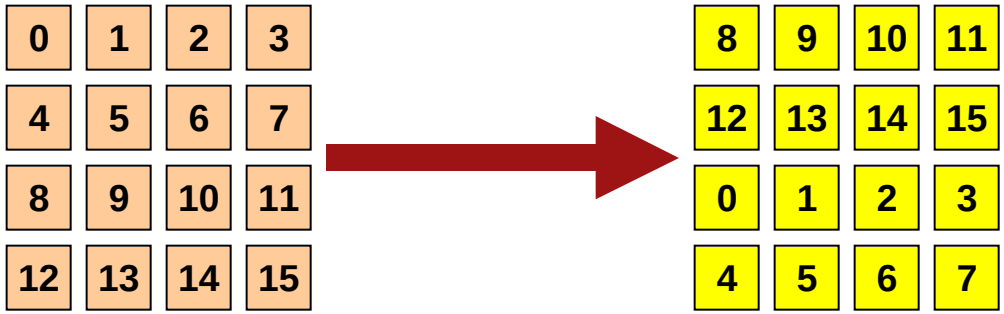
Intra-Matrix Butterflies
The Last Four Stages

Last four stages perform "intra-matrix" butterflies

- Operations occur between elements of the same matrix

Use select row and select column operations to create a copy with swapped elements

- Stage 6: swap top and bottom halves
- Stage 7: swap even/odd rows
- Stage 8: swap left/right halves
- Stage 9: swap even/odd columns



Selectcol dst, src, M_SWAP_TOP_BOTTOM_HALVES
M_SWAP_TOP_BOTTOM_HALVES = 10 10 10 10 11 11 11 11 00 00 00 00 01 01 01 01

Inner-Loop Implemented Without Software Pipelining

16-Cycle Inner Loop
(first 6 of 10 stages)

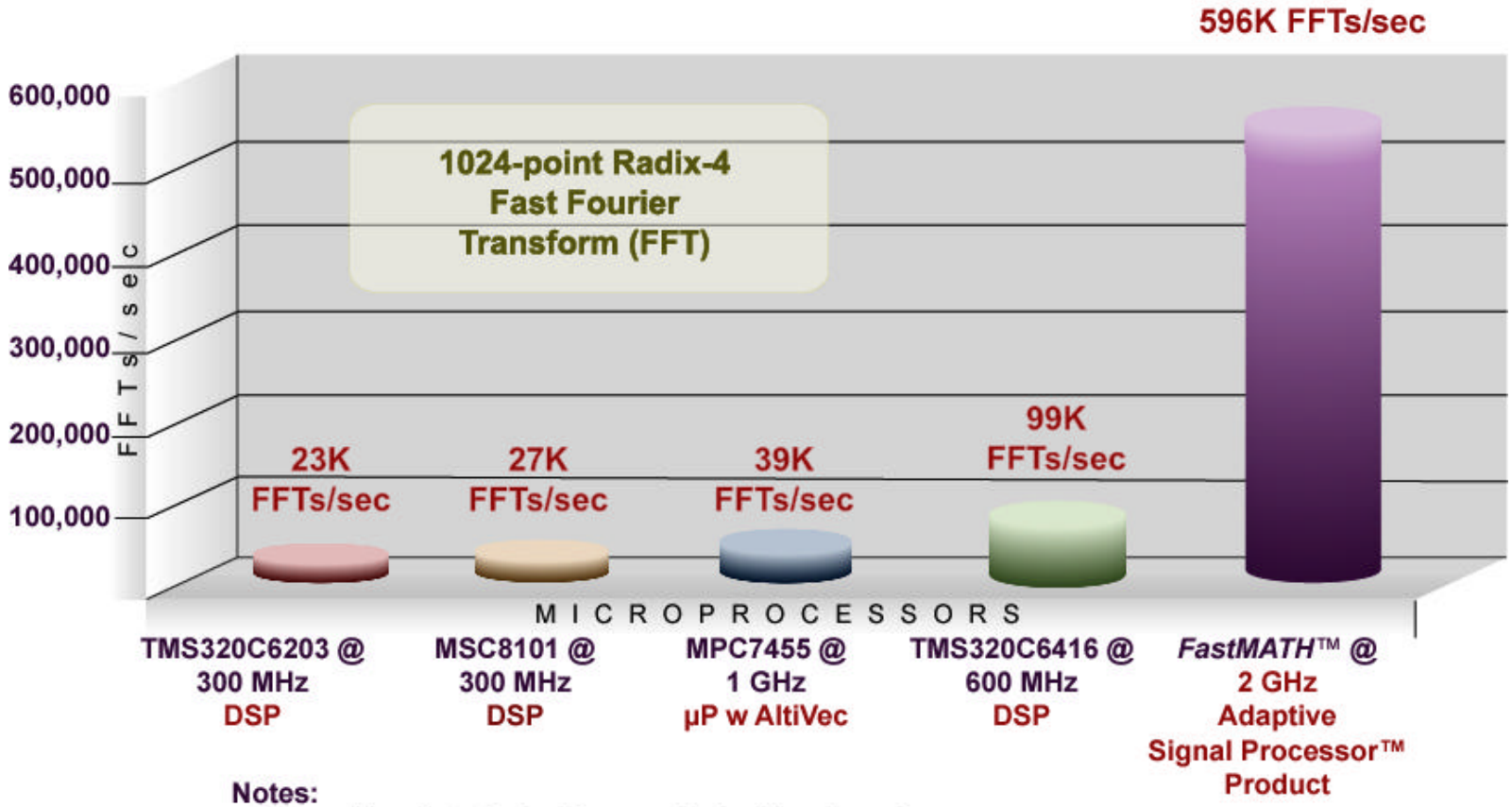
fftloop:		Cycle	
load.m	m0, 0(r3)	0	
srah.m	m0, m0	0	
addu	r3, r3, 64	1	
load.m	m1, 0(r4)	2	
srah.m	m1, m1	2	
addh.m.m	m3, m0, m1	3	
subfh.m.m	m2, m1, m0	4	
load.m	m1, 0(r5)	4	
mulhh.m	a0, m2, m1	5	
msubll.m	a0, m2, m1	6	
addu	r4, r4, 64	6	
mulhl.m	a1, m2, m1	7	
maddlh.m	a1, m2, m1	8	
store.m	m3, -64(r3)	9	
mflo.m	m0, a0	10	
addu	r5, r5, 64	10	
mflo.m	m1, a1	12*	(pipeline stall)
packhh.m	m0, m0, m1	13	
blt	r3, r6, fftloop	13	
store.m	m0, -64(r4)	14	(15 for loop to first load instr)

Software Pipelining Removes Stalls

13-Cycle Inner Loop (first 6 of 10 stages)

			Cycle	
	load.m	m2, 0(r3)	0	
	srah.m	m2, m2	0	
	load.m	m3, 0(r4)	2*	(pipeline stall)
	srah.m	m3, m3	2	
fftloop:				
	addh.m	m2, m2, m3	0	
	store.m	m2, 64(r3)	1	
	subfh.m.m	m2, m1, m0	2	
	addu	r3, r3, 64	2	
	load.m	m1, 0(r5)	3	
	mulhh.m	a0, m2, m1	3	
	msubll.m	a0, m2, m1	4	
	mulhhl.m	a1, m2, m1	5	
	maddhlh.m	a1, m2, m1	6	
	addu	r4, r4, 64	6	
	load.m	m2, 0(r3)	7	(load and scaling operations inserted from
	srah.m	m2, m2	7	next loop iteration to hide mul latency)
	mflo.m	m0, a0	8	
	addu	r5, r5, 64	8	
	load.m	m3, 0(r4)	9	(load and scaling operations inserted from
	srah.m	m3, m3	9	next loop iteration to hide mflo latency)
	mflo.m	m1, a1	10	
	packhh.m	m0, m0, m1	11	
	blt	r3, r6, fftloop	11	
	store.m	m0, -64(r4)	12	

FastMATH™ Performance Example: FFT



Notes:

- Competitive data derived from published benchmarks
- MPC7455 FFTs are radix-2; others are radix-4
- Competitive clock rates are highest announced

Intrinsity Is...

A fabless semiconductor company based in Bee Caves, Texas

- Down the road from Jim-Bob's BarBQ
- Formerly largest employer

Staffed by about 90 farmers and cedar choppers

Invented $Fast_{14}^{TM}$ Technology

www.intrinsity.com