SPEED COMPARISON: CORTEX-M0+ VS. CORTEX-M4F

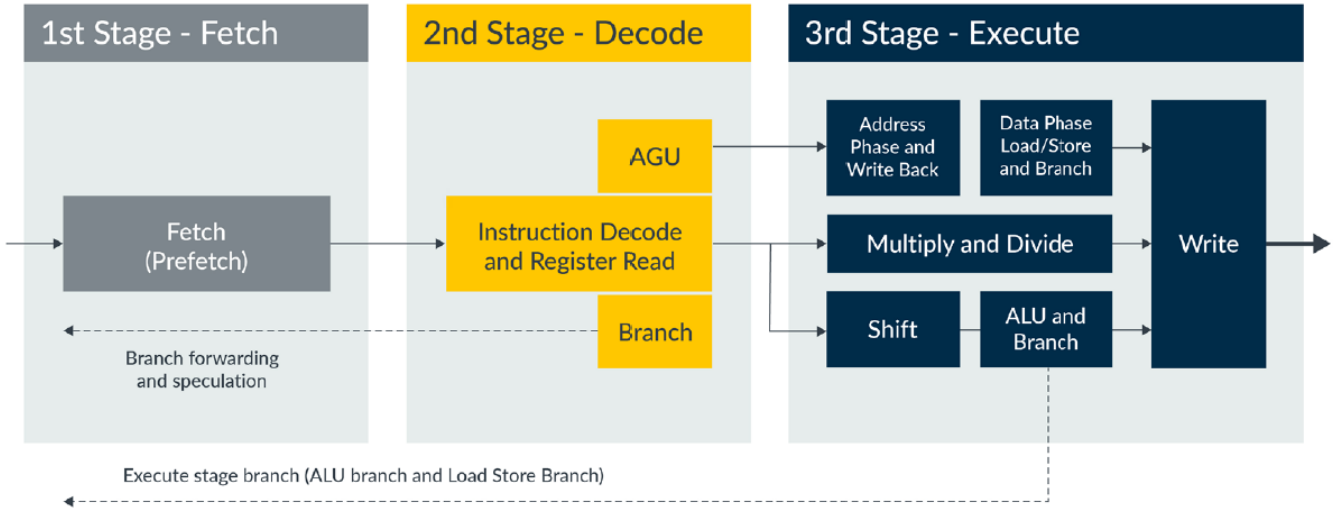
Overview

- M4F Performance Enhancing Features
- SG Optimization
  - Results
  - Analysis

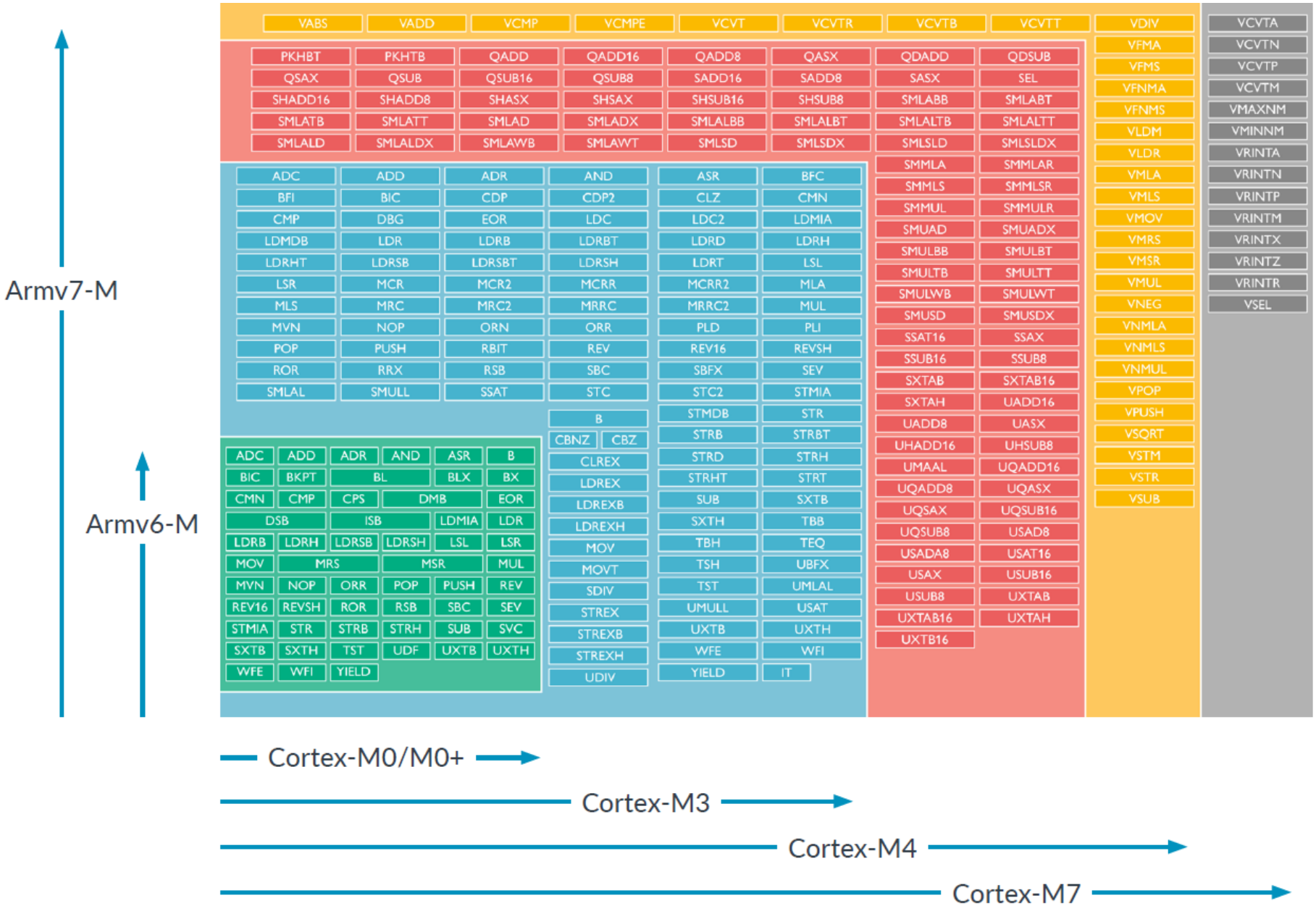
M4F Features

- Target platform: K64F on FRDM-K64F
  - Same generation as KL25Z
- Clock speed:
  - 120 MHz vs 48 MHz. 2.5x faster
- CPU Pipeline:
  - 3 stages instead of 2. Marginally slower for pipeline stalls
- Hardware floating-point math support (instructions and registers)
  - How much faster than software?
- More instructions

Cortex-M4 Pipeline



Newer, More Powerful Instructions



K64F Interconnection Networks

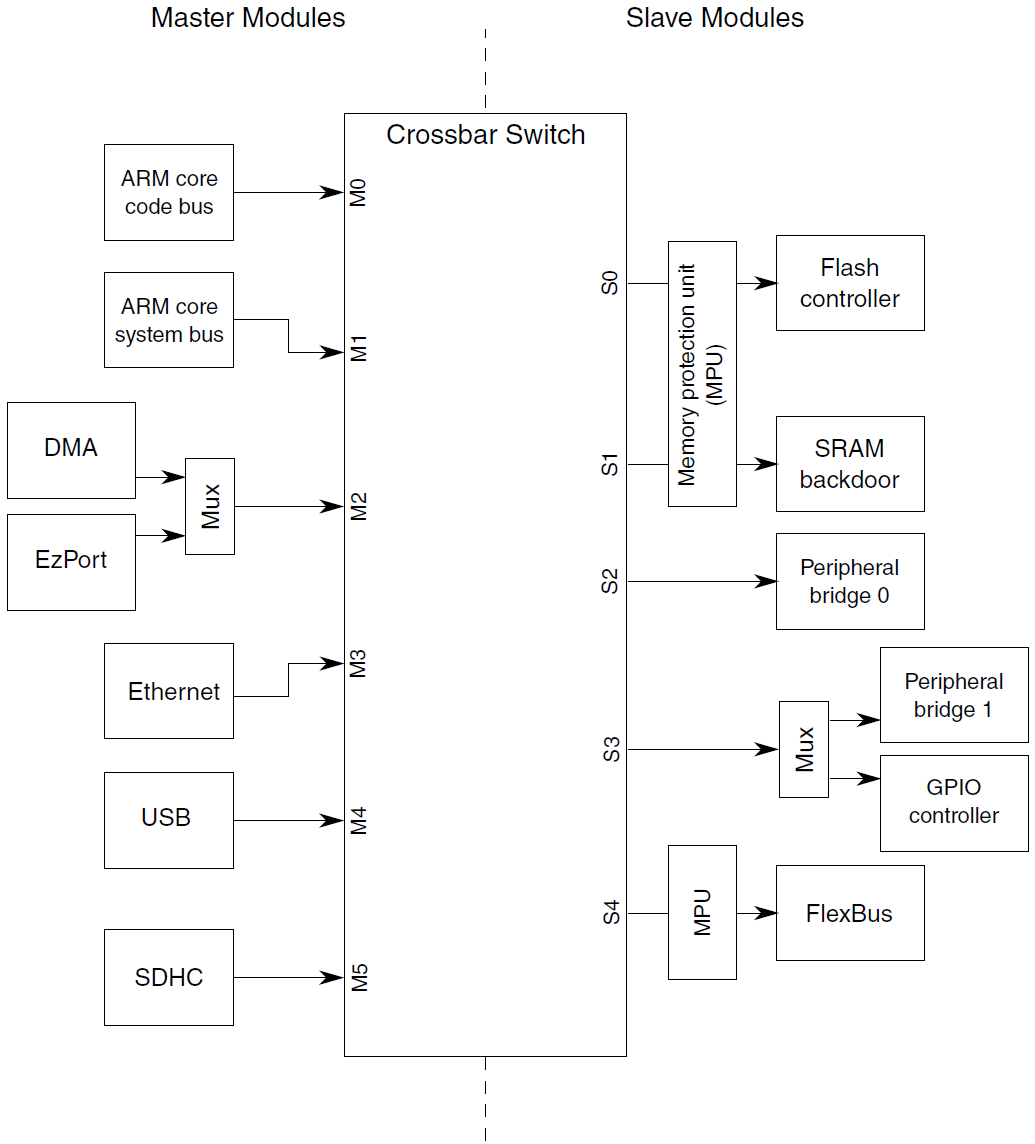


Figure 3-11. Crossbar switch integration

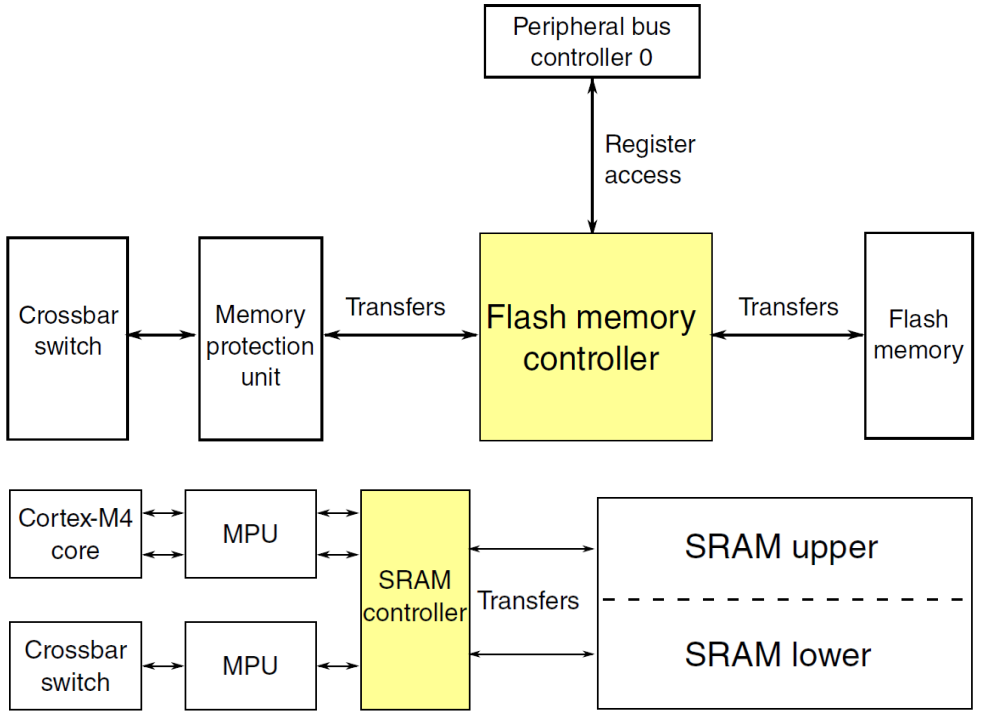


Figure 3-25. SRAM configuration

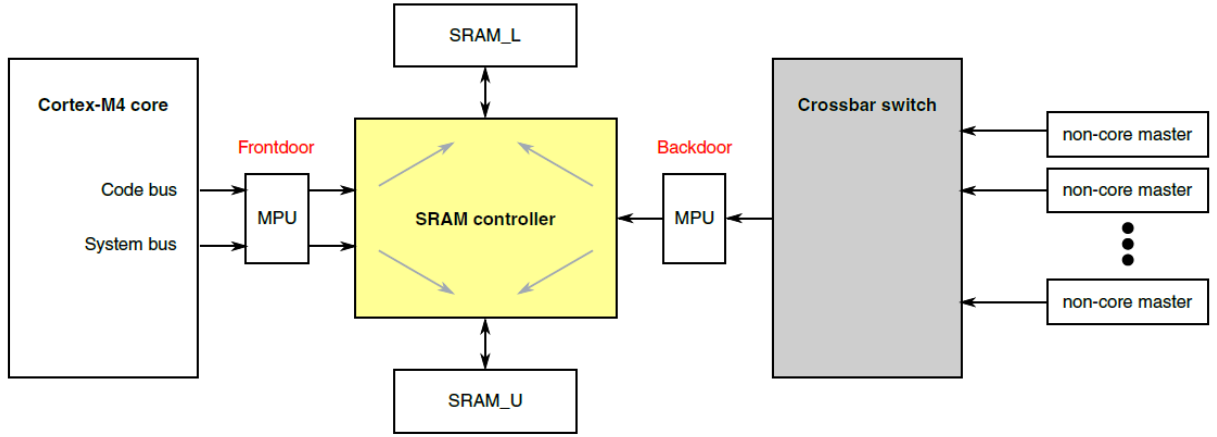


Figure 3-26. SRAM access diagram

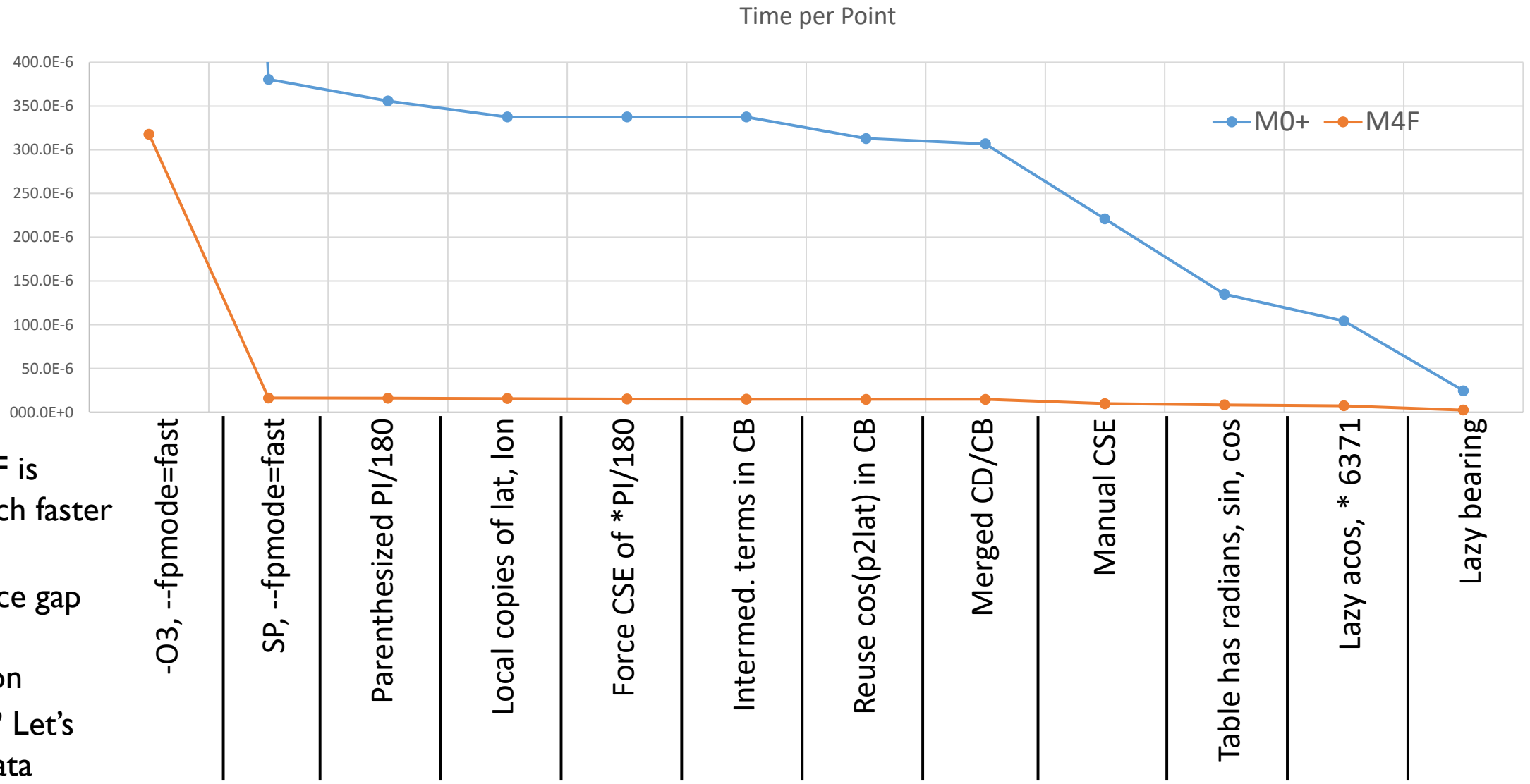
M4F Memory System

- Still no cache, but dual banks (flash, NVM) with 64-bit buses
- Controller has an accelerator (FMC, Chapter 28)
  - Prefetch: 8-byte speculation buffer (see 28.5.4)
  - 4-way, 4-set cache
  - 8-byte buffer for previously accessed data

Optimization of Example Program

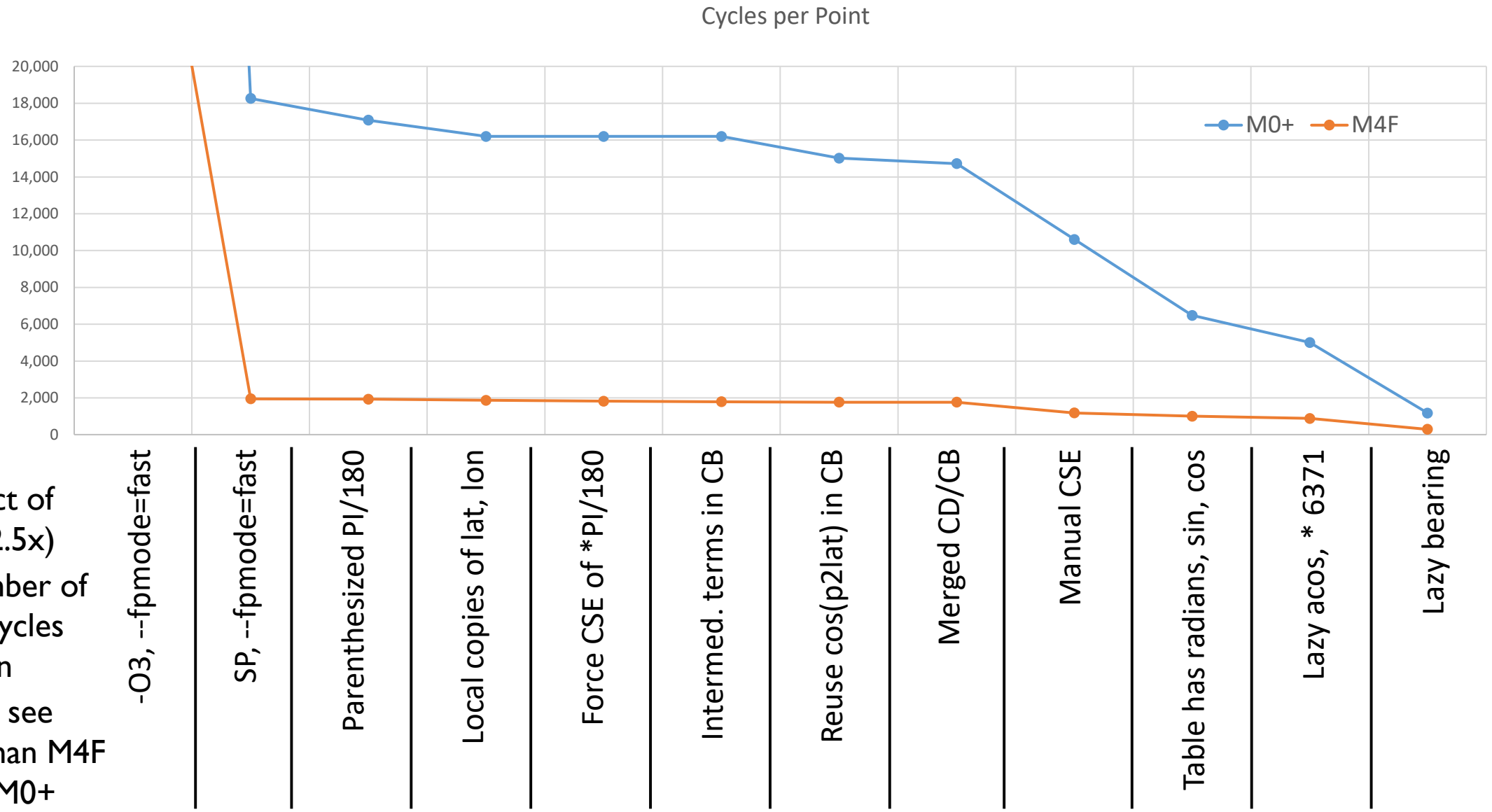
- Spherical Geometry – Find nearest point given coordinates lat/lon.
- Stepped through same tuning steps and optimizations as used on Cortex-M0+
- Program run time = # instructions * time per instruction * clock period
- KL25Z
  - 3,632 µs (174,331 CPU cycles) per point at start.
  - 380.4 µs (18,258 cycles): Switched from double-precision to single-precision floating-point math
  - 312.9 µs (15,018 cycles): Helping the compiler through code tweaks
  - 24.5 µs (1,178 cycles): precalculation, lazy execution
- How does K64F do?
  - Clock is 2.5x faster, so expect 24.5 µs → 10 µs
  - Floating point math in hardware. How much faster?

Time per Point



- Shows M4F is usually much faster than M0+
- Performance gap falls with optimization
- What else? Let's examine data differently

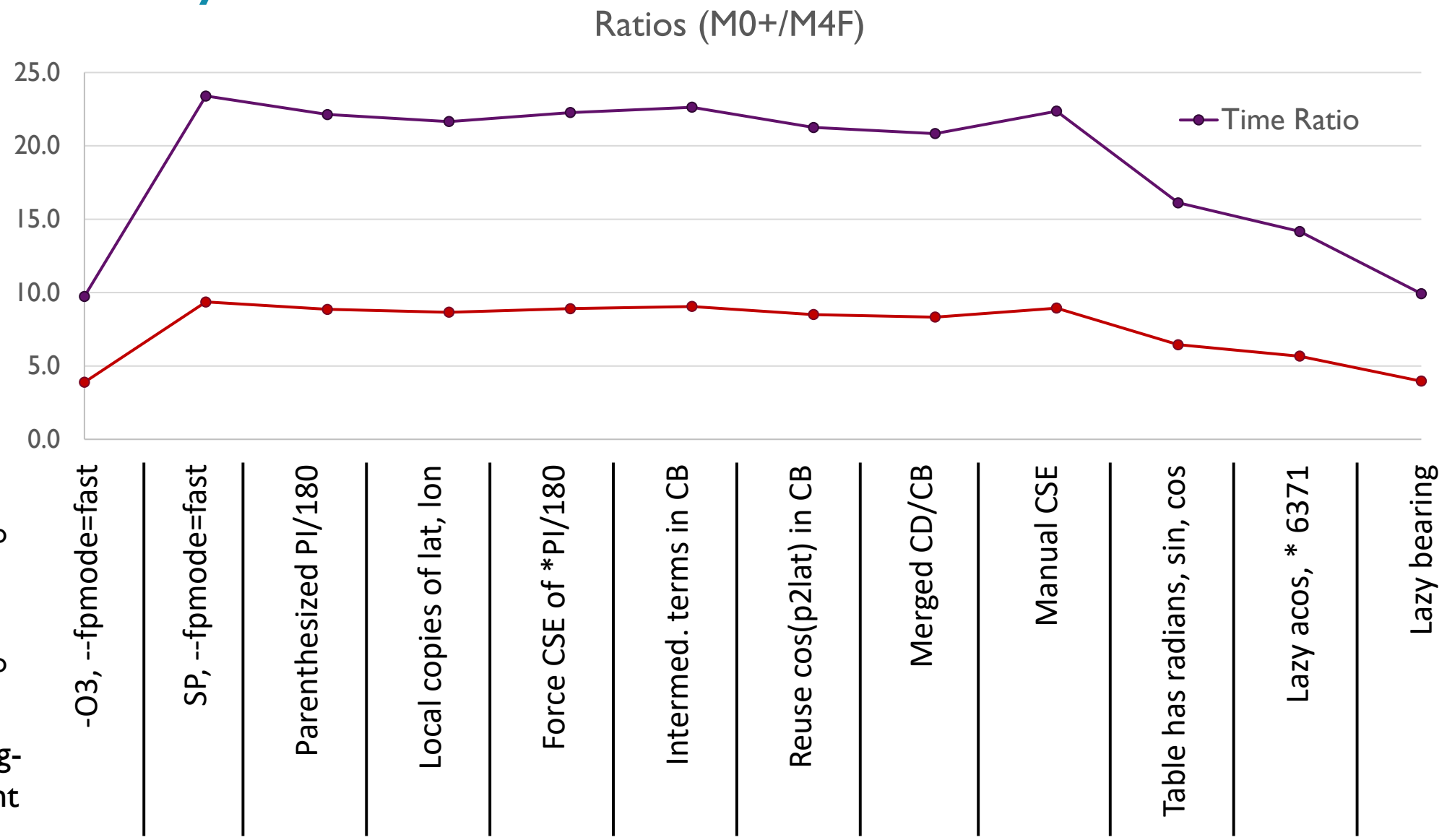
Cycles per Point



- Removes effect of faster clock (2.5x)
- Now just number of instructions, cycles per instruction
- OK, still don't see much other than M4F is faster than M0+

Time and Clock Cycle Ratios

- Look at M0+ vs. M4F ratio
- Time ratio includes ~2.5x performance gain from f_{clock}
- Clock cycle ratio shows impact of better instructions & microarchitecture
- Early opt. versions do lots of math
  - 9 to 1 clock cycle ratio
- Later versions reduce math
  - 4 to 1 clock cycle ratio
- Compare with floating-point math cycle count ratio?



Optimization of Example Program

	M0+		M4F	
	Time per pt (s)	Avg. Clock Cycles/Pt	Time per pt (s)	Avg. Clock Cycles/Pt
Base version, -O0	3.6E-3	174,331	325.0E-6	38,996
Calc_D,B -> Single Precision, --fpmode=fast, -O3	380.4E-6	18,258	16.3E-6	1,951
Reuse cos(p2lat) in CB	312.9E-6	15,018	14.7E-6	1,767
Don't do bearing until needed	24.5E-6	1,178	2.5E-6	297

Raw Performance Numbers

				Cortex-M0+				Cortex-M4F				
		Code Comments	Comments	Total Samples (fs = 1 kHz)	Total Time (s)	Time per pt (s)	Avg. Clock Cycles/Pt	Total Samples (fs = 1 kHz)	Loop Repeat Factor	Total Time per Loop (s)	Time per pt (s)	Avg. Clock Cycles/Pt
SG	1	Base version, -O0		592	0.592	3.6E-3	174,331	5297	100	0.05297	325.0E-6	38,996
	1	Base version, -O3, --fpmode=fast	Tuning	504	0.504	3.1E-3	148,417	5180	100	0.0518	317.8E-6	38,135
	2	Calc_D,B -> Single Precision, --fpmode=fast, -O3	Tuning	62	0.062	380.4E-6	18,258	265	100	0.00265	16.3E-6	1,951
	3	Parenthesized PI/180	Tuning	58	0.058	355.8E-6	17,080	262	100	0.00262	16.1E-6	1,929
	5	Local copies of lat and lon	Tuning	55	0.055	337.4E-6	16,196	254	100	0.00254	15.6E-6	1,870
	6	Force CSE of *PI/180	Tuning	55	0.055	337.4E-6	16,196	247	100	0.00247	15.2E-6	1,818
	7	Intermediate terms in CB	Tuning	55	0.055	337.4E-6	16,196	243	100	0.00243	14.9E-6	1,789
	8	Reuse cos(p2lat) in CB	Tuning	51	0.051	312.9E-6	15,018	240	100	0.0024	14.7E-6	1,767
	9	Merged CD/CB functions, left most CSE to optimizer		50	0.05	306.7E-6	14,724	240	100	0.0024	14.7E-6	1,767
	10	Merged CD/CB functions, extreme manual CSE		36	0.036	220.9E-6	10,601	161	100	0.00161	9.9E-6	1,185
SG2	11	Use radians in table, precalculate sin and cos for points	Data precomp.	22	0.022	135.0E-6	6,479	1365	1000	0.001365	8.4E-6	1,005
	12	Don't do acos or * 6371 until needed	Lazy execution	17	0.017	104.3E-6	5,006	1200	1000	0.0012	7.4E-6	883
	13	Don't do bearing until needed	Lazy execution	4	0.004	24.5E-6	1,178	403	1000	0.000403	2.5E-6	297

Final Profile

- Cosine approximation dominates time (~50%)
  - Can we use a faster, lower-degree polynomial approximation?
- Find_Nearest_Waypoint second
  - Should examine to see which code within loop dominates. Any inlined functions?
- Strcmp is a very close third
  - How does strcmp spend its time?
  - Can we replace it with a version which returns as soon as first difference (if any) is found?

profile_samples	380
num_lost	0
adx_lost	0x00000000
RegionTable[SortedRegions...	0x0000337C "__hardfp_cosf"
RegionCount[SortedRegion...	179
RegionTable[SortedRegions...	0x000031DC "Find_Nearest_Waypoint"
RegionCount[SortedRegion...	100
RegionTable[SortedRegions...	0x00002FDC "strcmp"
RegionCount[SortedRegion...	98
RegionTable[SortedRegions...	0x0000335C "__hardfp_atan2f"
RegionCount[SortedRegion...	2
RegionTable[SortedRegions...	0x0000339C "__hardfp_sinf"
RegionCount[SortedRegion...	1