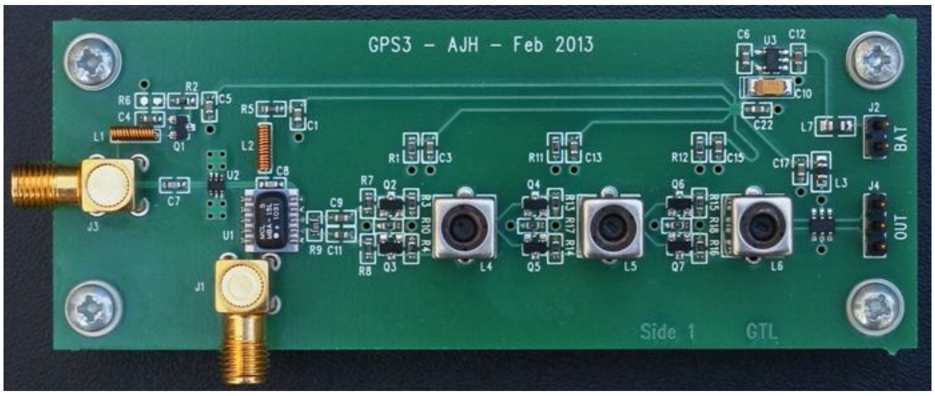
Homemade GPS Receiver

Back to projects



Pictured above is the front-end, first mixer and IF amplifier of an experimental GPS receiver. The leftmost SMA is connected to a commercial antenna with integral LNA and SAW filter. A synthesized first local oscillator drives the bottom SMA. Pin headers to the right are power input and IF output. The latter is connected to a Xilinx FPGA which not only performs DSP, but also hosts a fractional-N frequency synthesizer. More on this later.

I was motivated to design this receiver after reading the work [1] of Matjaž Vidmar, S53MV, who developed a GPS receiver from scratch, using mainly discrete components, over 20 years ago. His use of DSP following a hard-limiting IF and 1-bit ADC interested me. The receiver described here works on the same principle. Its 1-bit ADC is the 6-pin IC near the pin headers, an LVDS-output comparator. Hidden under noise but not obliterated in the bi-level quantised mush that emerges are signals from every satellite in view.

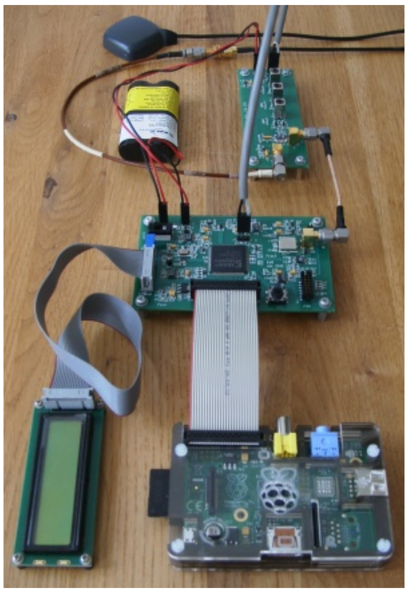
All GPS satellites transmit on the same frequency, 1575.42 MHz, using direct sequence spread spectrum (DSSS). The L1 carrier is spread over a 2 MHz bandwidth and its strength at the Earth's surface is -130 dBm. Thermal noise power in the same bandwidth is -111 dBm, so a GPS signal at the receiving antenna is ~ 20 dB below the noise floor. That any of the signals present, superimposed one on another and buried in noise, are recoverable after bi-level quantisation seems counter-intuitive! I wrote a simulation to convince myself.

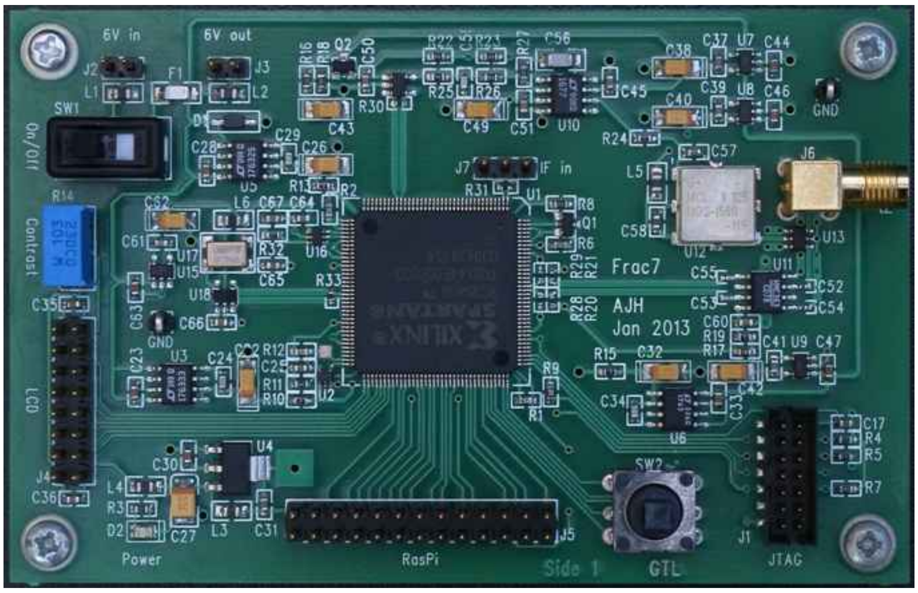
GPS relies on the correlation properties of pseudo-random sequences called Gold Codes to separate signals from noise and each other. Every satellite transmits a unique sequence. All uncorrelated signals are noise, including those of other satellites and hard-limiter quantisation errors. Mixing with the same code in the correct phase de-spreads the wanted signal and further spreads everything else. Narrow-band filtering then removes wideband noise without affecting the (once again narrow) wanted signal. Hard-limiting (1-bit ADC) degrades SNR by less than 3 dB, a price worth paying to avoid hardware AGC.

May 2013 Update

This is now a truly portable, battery-powered, 12-channel GPS receiver with turnkey software, which acquires and tracks satellites, and continuously recalculates its position, without user-intervention. The complete system (below, left) comprises: 16x2 LCD display, Raspberry Pi Model "A" computer, two custom printed-circuit boards, commercial patch antenna and Li-Ion battery. Total system current consumption is 0.4A for a battery life of 5 hours. The Raspberry Pi is powered through the ribbon cable linking its GPIO header to the "Frac7" FPGA board and requires no other connections.

Currently, the Pi is running Raspbian Linux. A smaller distro would shorten time to first fix. After booting from SD-Card, the GPS application software starts automatically. On exit, it provides a means to properly shutdown the Pi before powering-off. Pi software development was done "head-less" via SSH and FTP over a USB Wi-Fi dongle. Source code and documentation can be found towards the bottom of this page.





Both custom PCBs are simple 2-layer PTH boards with continuous ground planes on the bottom. Going clockwise around the Xilinx Spartan 3 on the "Frac7" FPGA board: from 12 o'clock to 3 o'clock are the loop filter, VCO, power splitter and prescaler of the microwave frequency synthesizer; bottom right are the joystick and JTAG connector; and, at 6 o'clock, a pin header for the Raspberry Pi ribbon cable. Far left is the LCD connector. Near left is a temperature-compensated voltage-controlled crystal oscillator (TCVCXO) providing a stable reference frequency, vital for GPS reception.

The TCVCXO is good; but not quite up to GPS standard when operating un-boxed in windy locations. Blowing on it displaces the 10.000000 MHz crystal oscillator by around 1 part in 10 million or 1 Hz, which is magnified 150 times by the synthesizer PLL. This is enough to momentarily unlock the satellite tracking loops, if done suddenly. The device is also slightly sensitive to infra-red e.g. from halogen bulbs and TV remotes!

When first posted in 2011, this was a four-channel receiver, meaning it could only track four satellites simultaneously. At least four are required to solve for user position and receiver clock bias; but greater accuracy is possible with more. In that original version, four identical instances of the "tracker" module filled the FPGA. But most of the flops were only clocked once per millisecond. Now, a custom "soft-core" CPU inside the FPGA serializes the processing and only 50% of the FPGA fabric is required for an 8-channel receiver or 67% for 12-channels. Number of channels is a parameter in the source and could go higher.

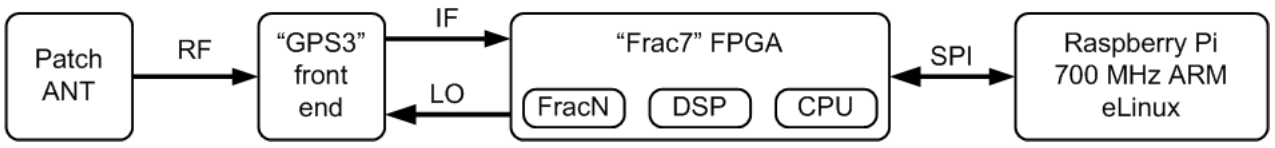
Positional accuracy is best when the antenna can see 360° of sky and receive signals from all directions. Generally, the more satellites in view, the better. Two or more satellites on the same bearing can lead to what is termed "bad geometry." The best fix so far was ±1 metres at a very open location using 12 satellites; but accuracy is typically ±5 metres in poorer locations with fewer satellites.

September 2014 Update

The source code for this project has been re-released under the GNU General Public License (GPL).

Architecture

Processing is split between FPGA and Pi by complexity and urgency. The Pi handles math-intensive heavy-lifting at its own pace. The FPGA synthesizes the first local oscillator, services high-priority events in real-time and tracks satellites autonomously. The Pi controls the FPGA via an SPI interface. Conveniently, the same SPI is used to load the FPGA configuration bitstream and binary executable code for the embedded CPU. The FPGA can also be controlled via a Xilinx Platform USB JTAG cable from a Windows PC and auto-detects which interface is in use.

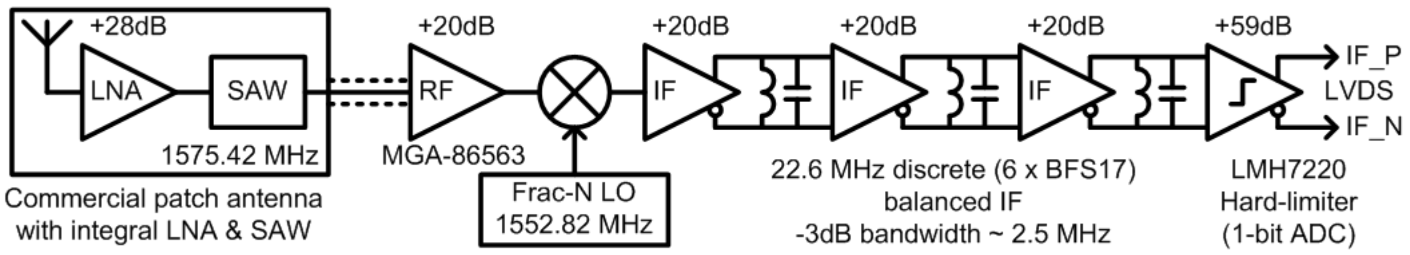


L1 frequencies are down-converted to a 1st IF of 22.6 MHz by mixing with a 1552.82 MHz local oscillator on the "GPS3" front-end board. All subsequent IF and baseband signal processing is done digitally in the FPGA. Two proportional-integral (PI) controllers per satellite, track carrier and code phase. NAV data transmitted by the satellites is collected in FPGA memory. This is uploaded to the Pi, which checks parity and extracts ephemerides from the bit stream. When all required orbital parameters are collected, a snapshot is taken of certain internal FPGA counters, from which time of transmission is computed to ± 15ns precision.

Much of the 1552.82 MHz synthesizer is implemented in the FPGA. One might expect jitter problems, co-hosting a phase detector with other logic, but it works. Synthesizer output spectral-purity is excellent, even though the FPGA core is toggling away furiously and not all on harmonically-related frequencies. This approach was taken because a board similar to "Frac7" already existed from an earlier synthesizer project. Adding a front-end was the shortest route to a prototype receiver. But that first version was not portable: it had inconvenient power requirements and no on-board frequency standard.

Front-end

Signal processing up to and including the hard-limiter:

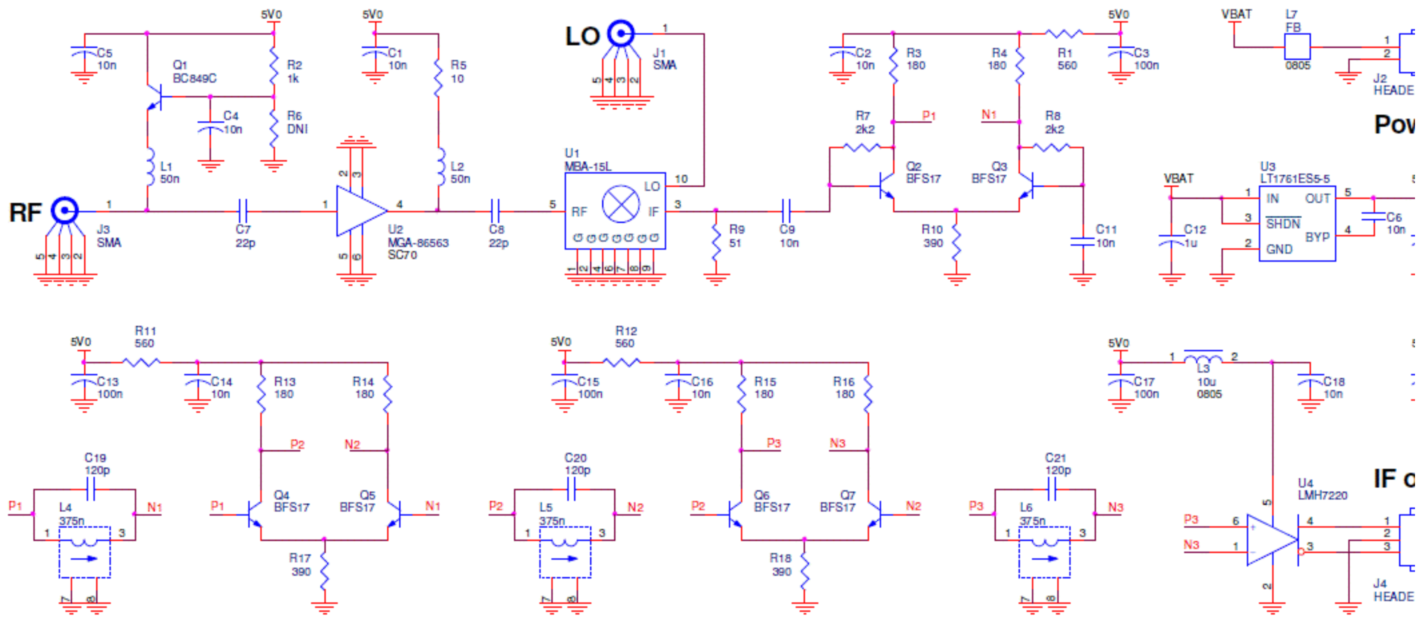

+28dB
LNA
SAW
1575.42 MHz
Commercial patch antenna with integral LNA & SAW
+20dB
RF
MGA-86563
Frac-N LO 1552.82 MHz
+20dB
IF
+20dB
IF
+20dB
IF
22.6 MHz discrete (6 x BFS17) balanced IF -3dB bandwidth ~ 2.5 MHz
+59dB
IF_P
LVDS
IF_N
LMH7220 Hard-limiter (1-bit ADC)

The LMH7220 comparator has a maximum input offset voltage of 9.5mV. Amplified thermal noise must comfortably exceed this to keep it toggling. Weak GPS signals only influence the comparator near zero crossings! They are "sampled" by the noise! To estimate noise level at the comparator input we tabulate gains, insertion losses and noise figures:

	LNA	SAW	Coax	RF	Mixer	IF	**Overall system noise figure**
Gain	+28	-1.5	-3.9	+20	-6		
NF	0.8	1.5	3.9	2	6	7	**0.8 dB**

In-band noise at the mixer output is -174+0.8+28-1.5-3.9+20-6+10*log10(2.5e6) = -73 dBm or 52µV RMS. The mixer is resistively terminated in 50-ohms and the stages thereafter work at higher impedance. The discrete IF strip has an overall voltage gain of 1000 so the comparator input level is 52mV RMS.

The LMH7220 adds 59 dB of gain making a total of 119 dB for the whole IF. Deploying so much gain at one frequency was a risk. To minimise it, balanced circuitry over a solid ground plane was used and screened twisted-pair carries the output to the FPGA. The motivation was simplicity, avoiding a second conversion. In practice, the circuit is stable, so the gamble paid-off.



Active decoupler Q1 supplies 5V for the remote LNA. MMIC amplifier U2 provides 20 dB gain (not at IF!) and ensures low overall system noise figure, even if long antenna cables are used. L1 and L2 are hand-wound microwave chokes with very high self-resonant frequency, mounted perpendicular to one another and clear of the ground plane. Wind 14 turns, air-cored, 1mm inside diameter from 7cm lengths of 32swg enamelled copper wire. Checked with the tracking generator on a Marconi 2383 SA, these were good to 4 GHz.

The Mini-Circuits MBA-15L DBM was chosen for its low 6 dB conversion loss at 1.5 GHz and low 4 dBm LO drive requirement. R9 terminates the IF port.

Three fully-differential IF amplifier stages follow the mixer. Low-Q parallel tuned circuits strung between collectors set the -3 dB bandwidth around 2.5 MHz and prevent build-up of DC offsets. L4, L5 and L6 are screened Toko 7mm coils. The BFS17 was chosen for its high (but not too high) 1 GHz f_T. I_e is 2mA for lowest noise and reasonable βr_e.

The 22.6 MHz 1st IF is digitally down-converted to 2.6 MHz by under-sampling at 10 MHz in the FPGA. 2.6 MHz lies close to the centre of the 5 MHz Nyquist bandwidth. It is best to avoid the exact centre, for reasons that will be explained later. Several other first IF frequencies are possible: 27.5 MHz, which produces spectrum inversion at the 2nd IF, has also been tried successfully. There is a trade-off between image problems at lower and available BFS17 gain at higher frequencies.

Search

Signal detection entails resolving three unknowns: what satellites are in view, their Doppler shifts and code phases. A sequential search of this three-dimensional space from a so-called "cold start" could take many minutes. A "warm start" using almanac data to predict positions and velocities still requires a code search. All 1023 code phases must be tested to find the maximum correlation peak. Calculating 1023 correlation integrals in the time-domain is very expensive and redundant. This GPS receiver uses an FFT-based algorithm that tests all code phases in parallel. From cold, it takes 2.5 seconds on a 1.7 GHz Pentium to measure signal strength, Doppler shift and code phase of every visible satellite. The Raspberry Pi is somewhat slower.

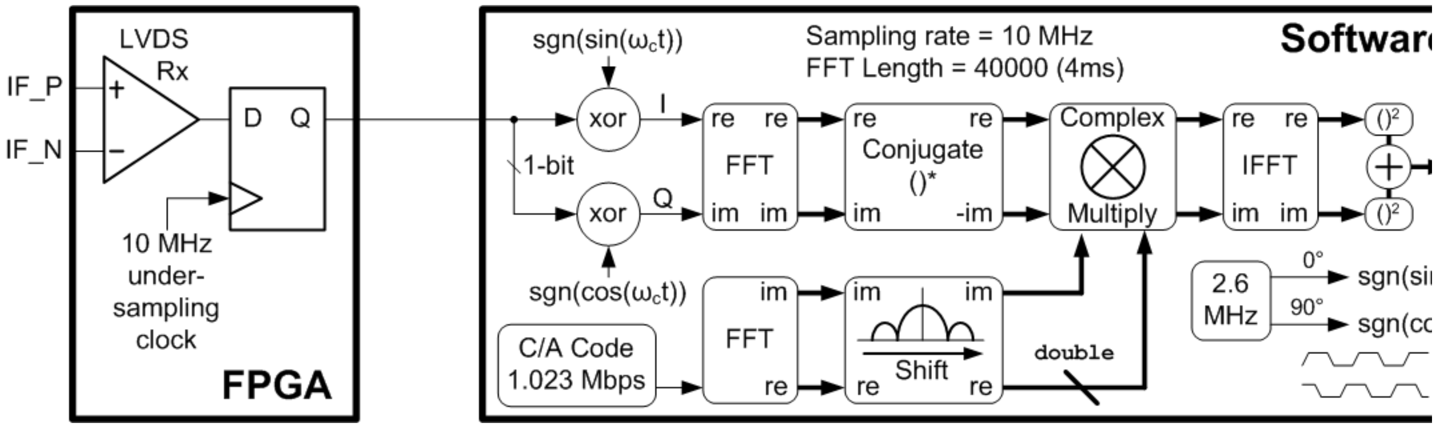
With over-bar denoting conjugation, the cross-correlation function y(T) of complex signal s(t) and code c(t) shifted by offset T is:

$$y(\tau) = \int_{-\infty}^{\infty} \bar{s}(t)\ c(\tau + t)\ dt$$

The Correlation Theorem states that the Fourier transform of a correlation integral is equal to the product of the complex conjugate of the Fourier transform of the first function and the Fourier transform of the second function:

```
FFT(y) = CONJUGATE(FFT(s)) * FFT(c)
```

Correlation is performed at baseband. The 1.023 Mbps C/A code is 1023 chips or 1ms long. Forward FFT length must be a multiple of this. Sampling at 10 MHz for 4 ms results in an FFT bin size of 250 Hz. 41 Doppler shifts must be tested by rotating the frequency domain data, one bin at a time, up to ±20 bins = ±5 KHz. Rotation can be applied to either function.

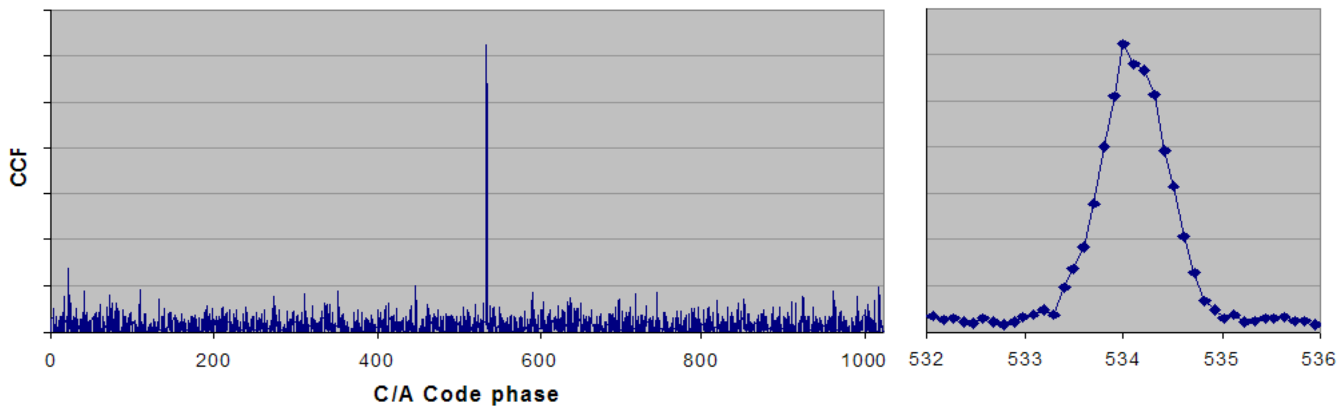


The 22.6 MHz 1st IF from the 1-bit ADC is under-sampled by a 10 MHz clock in the FPGA, digitally down-converting it to a 2nd IF of 2.6 MHz. In software, the 2nd IF is down-converted to complex baseband (IQ) using quadrature local oscillators. For bi-level signals, the mixers are simple XOR gates. Although not shown above, the samples are temporarily buffered in FPGA memory. The Pi is not able to accept them at 10 Mbps.

1.023 Mbps and 2.6 MHz are generated by numerically-controlled-oscillator (NCO) phase accumulators. These frequencies are quite large compared to the sampling rate, and are not exact sub-harmonics of it. Consequently, the NCOs have fractional spurs. The number of samples per code chip dithers between 9 and 10. Fortunately, DSSS receivers are tolerant of narrow-band interferers, external or self-generated.

Complex baseband is transformed to the frequency domain by a forward FFT which need only be computed once. An FFT of each satellite's C/A code is pre-computed. Processing time is dominated by the inner-most loop which performs shifting, conjugation, complex multiplication and one inverse-FFT per satellite-Doppler test. The Raspberry Pi's Videocore GPU could be leveraged to speed things up.

At 10 MHz sampling rate, code phase is resolved to the nearest 100ns. Typical CCF output is illustrated below:



Calculating peak to average power over this data gives a good estimate of SNR and is used to find the strongest signals. The following were received at 20:14 GMT on 4 March 2011 in Cambridge, UK with the antenna on an outside North-facing window ledge:

PRN	NAVSTAR	Doppler (Hz)	Code Phase	SNR	
9	33	1500	2.4	95.3	
17	57	500	364.5	98.4	
22	53	1000	844.7	54.1	
27	27	0	770.0	53.8	
28	48	-3000	103.9	99.1	

From northern latitudes, more GPS satellites will generally be found in the southern sky i.e. towards the equator.

Taking longer samples increases SNR, revealing weaker signals; but cancellation occurs when the capture spans NAV data transitions. Forward FFT length is an integral number of milliseconds; however, the inverse FFT can be shortened, simply by throwing away data in higher frequency bins. SNR is preserved; but code phase is not so sharply resolved. Nevertheless, a good estimate of peak position is obtained by weighted averaging the two strongest adjacent bins; and off-air tests suggest this could work even down to quite short inverse FFT lengths.

Tracking

Having detected a signal, the next step is locking on, tracking it and demodulating the 50 bps NAV data. This requires two inter-dependent phase locked loops (PLLs) to track code and carrier phase. These PLLs must operate in real-time and are implemented as DSP functions in the FPGA. Pi software has a supervisory role: deciding which satellites to track, monitoring the lock status and processing the received NAV data.

The tracking loops are good at maintaining lock, because they have very narrow bandwidths; however, this same characteristic makes them poor at acquiring lock without help. They cannot "see" beyond loop bandwidth to capture anything further away. Initial phases and frequencies must be preset to the measured code phase and Doppler shift of the target satellite. This is orchestrated under Pi control. The loops should be in-lock from the outset and remain so.

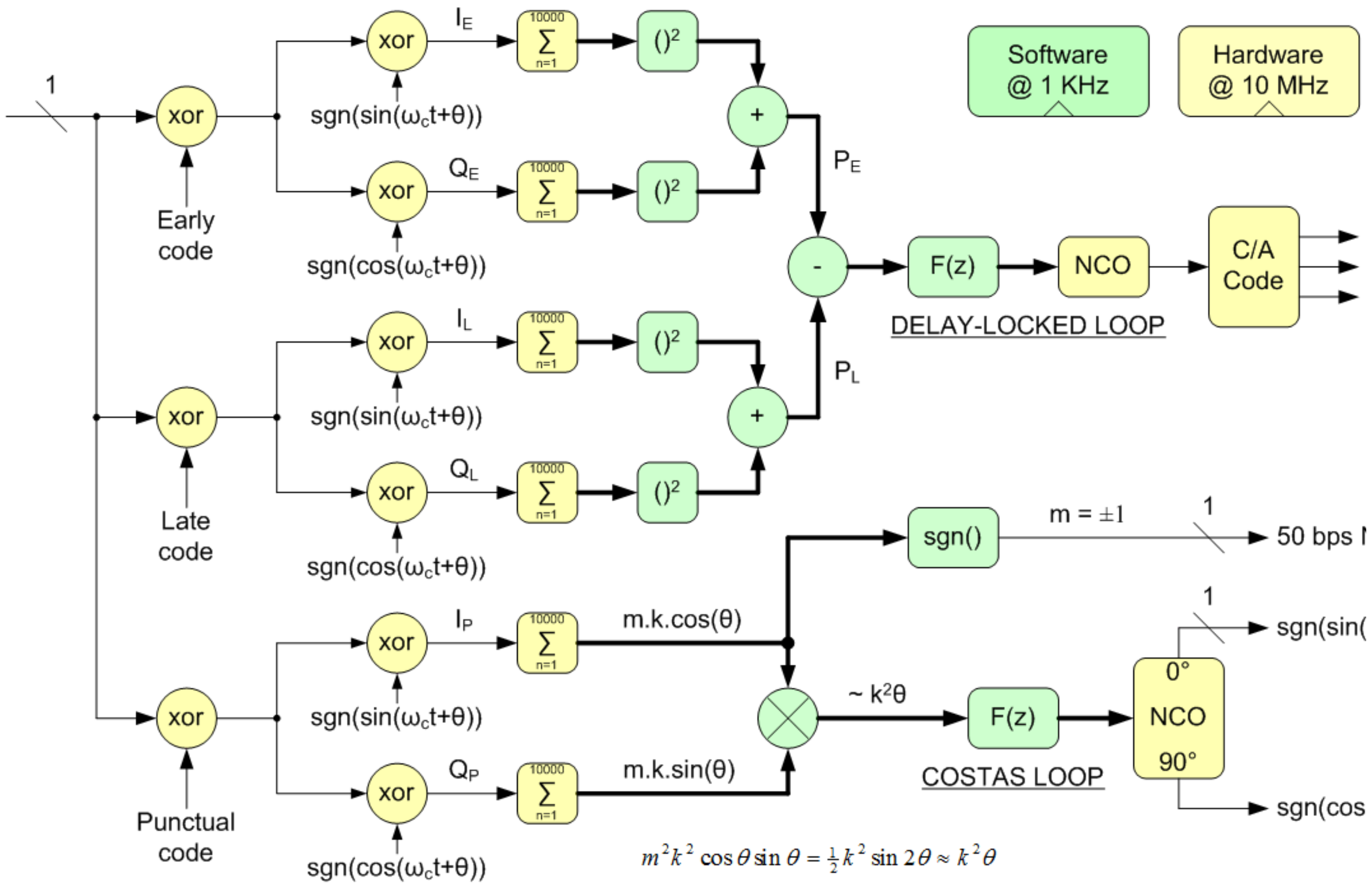
Code phase is measured relative to the FFT sample. The code NCO in the FPGA is reset at the start of sampling and accumulates phase at a fixed 1.023 MHz. It is later aligned with the received code by briefly pausing the phase accumulator. Doppler shift on the 1575.42 MHz carrier is ±5 KHz or ±3 ppm. It also affects the 1.023 Mbps code rate by ±3 chips per second. The length of the pause is adjusted for code creep in the time since the sample was taken. Fortunately, code Doppler is proportional to carrier Doppler for which we have a good estimate.

Hardware / software split

In the diagram below, colour-coding shows how the implementation of the tracking DSP is now split between hardware and software. Previously, this was all done in hardware, with identical parallel instances repeated for each channel, making inefficient use of FPGA resources. Now, the slower 1 KHz processing is done by software, and twice as many channels can be accommodated in half the FPGA real-estate.

The six integrate-and-dump accumulators (Σ) are latched into a shift register on the code epoch. A service request flag signals the CPU, which reads the data bit-serially. With 8 channels active, 8% of CPU time is spent executing the `op_rdBit` instruction! But there is plenty of time, and serial I/O uses FPGA fabric economically. Luxuries like RSSI and IQ logging (e.g. for scatter plots) can now be afforded.

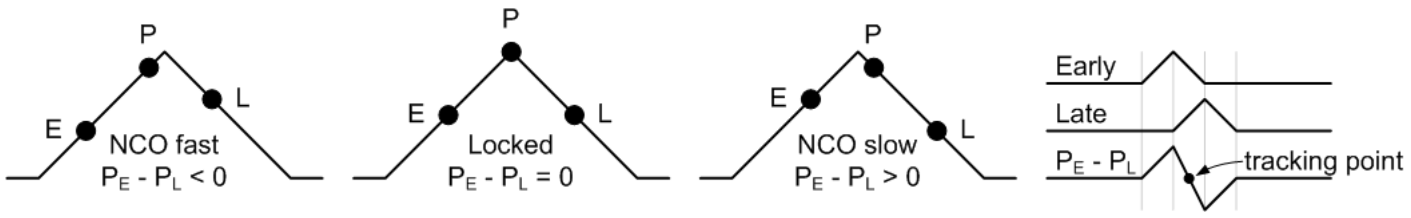
The F(z) loop filter transfer functions swallow 2% of CPU bandwidth per active channel. These are standard proportional-integral (PI) controllers: 64-bit precision is used and gain coefficients KI and KP, although restricted to powers of 2, are dynamically adjustable. Each channel having to wait its turn, NCO rate-updates can be delayed by tens or hundreds of microseconds after a code epoch; but this introduces negligible phase shift at frequencies where phase margin is determined.



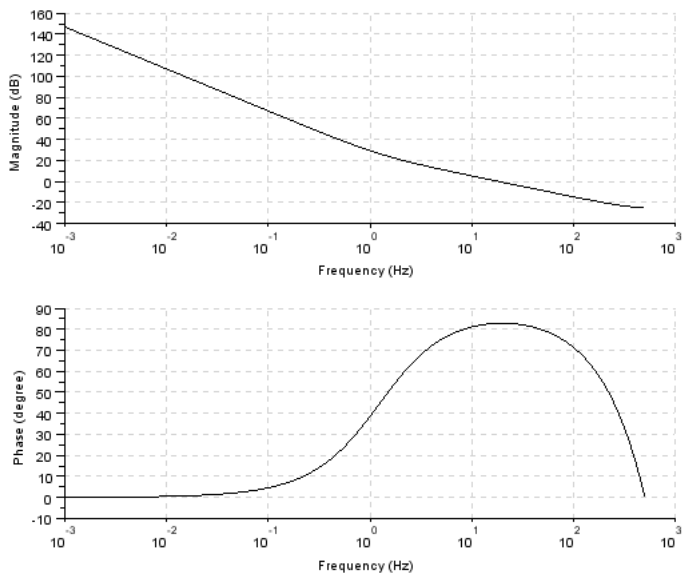
$$m^2k^2\cos\theta\sin\theta = \tfrac{1}{2}k^2\sin 2\theta \approx k^2\theta$$

Thin traces are 1-bit, notionally representing ±1. The 2.6 MHz carrier is first de-spread by mixing with early, late and punctual codes. I and Q complex baseband products from the second rank of XOR gate mixers are summed over 10000 samples or 1ms. This low-pass filtering dramatically reduces noise bandwidth and thereby raises SNR. Downsampling to 1 KHz necessitates wider onward data paths in the software domain.

Code phase is tracked using a conventional delay-locked loop or "early-late" gate. Power in the early and late channels is calculated using $P = I^2 + Q^2$ which is insensitive to phase. Early and late codes are one chip apart i.e. ½ chip ahead-of and behind punctual. This diagram helps to get the error sense correct:



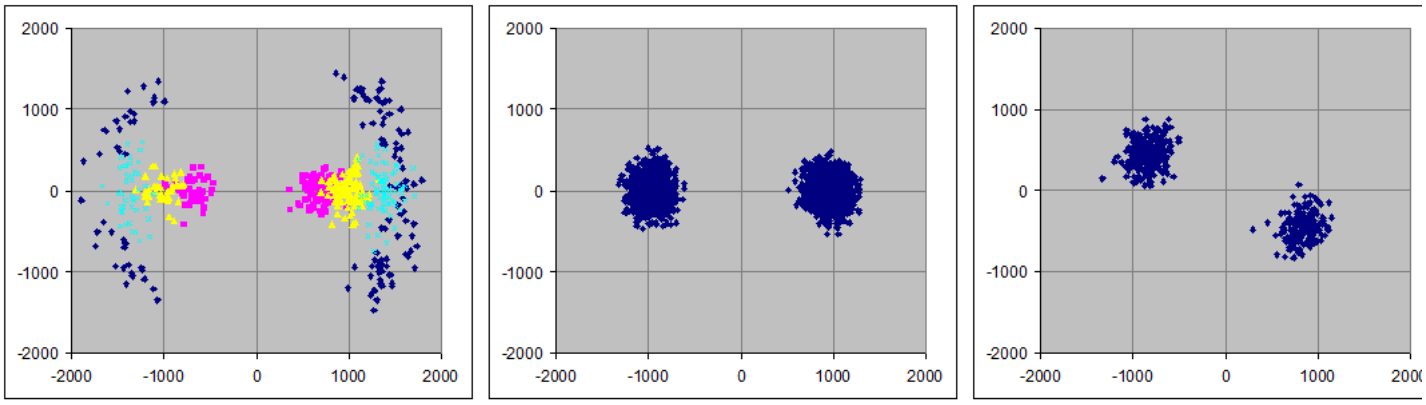
A Costas Loop is used for carrier tracking and NAV data recovery in the punctual channel. NAV data, m, is taken from the I-arm sign bit with 180° phase uncertainty. k is received signal amplitude and θ is phase difference between received carrier (sans modulation) and the local NCO. k varies from around 400 for the weakest recoverable signals up to over 2000 for the strongest. Notice how the error term fed back to the F(z) plant controller in the Costas Loop is proportional to received signal power k². Tracking slope, and therefore loop gain, also vary with signal power in the code loop. Below is a Bode plot of open-loop gain for the Costas Loop at k=500:



```
// Scilab script
// Bode plot: Costas carrier tracking loop

rssi = 500;  // amplitude
f1 = 10e6;   // 10 MHz
f2 = 1e3;    //  1 KHz

kPD = rssi^2;

kNCO = 2 * %pi * (f1/f2) / (%z-1);

kI = 2^(20-64);
kP = 2^(27-64);

G = kPD * kNCO * (kP + kI/(%z-1));
G.dt = 1/f2;

scf(0);
clf;
bode(-G, 1e-3, 500, 0.01);
```

Costas Loop bandwidth is around 20 Hz, which is about optimal for carrier tracking. Code loop bandwidth is 1 Hz. Noise power in such bandwidths is small and the loops can track very weak signals. The above kI and kP work for most signals, but need dropping one notch for the very strongest. Scilab predicts, and scatter plots confirm, the onset of instability at k≥1500. Parity errors do not occur unless samples stray into the opposite half of the IQ plane.

(i) Instability at k≥1500 (ii) Lock (iii) Phase error



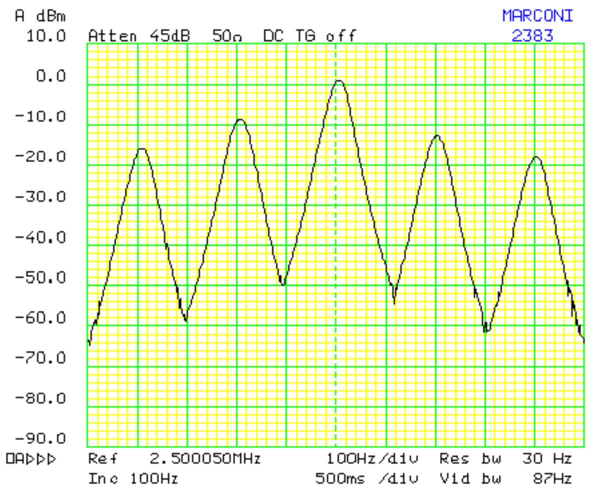
The amount of Doppler shift is always changing. Tracking a shifting carrier frequency requires a small, constant phase error at the loop filter input to drive the integrator. Insufficient kI integrator gain makes the phase error visible as a rotation of the scatter plot; and true or inverted NAV data appears in the sign-bit of the Q channel.

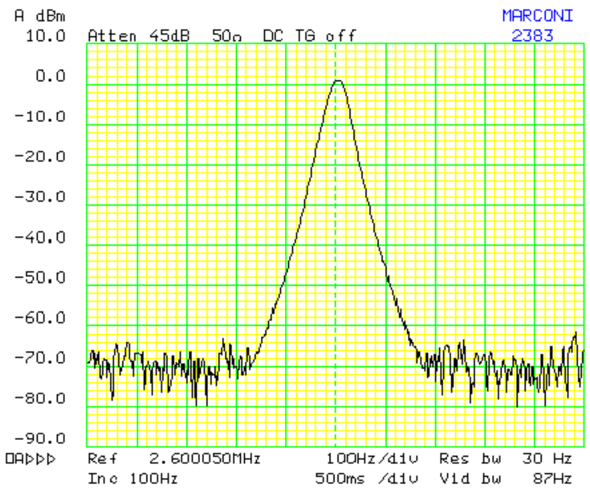
Acquisition

The code generator is aligned and both loop NCO frequencies are initially set using FFT search data. The initial carrier NCO can be up to 250 Hz (FFT bin size) off-frequency, placing it beyond loop capture range. Initial code rate error cannot exceed 0.16 Hz and the code loop is insensitive to carrier phase. If the signal is strong enough, the code loop always locks; but the carrier loop sometimes needs help. Fortunately, the exact carrier offset can be calculated from the locked code NCO, since both both exhibit the same Doppler shift. The carrier loop always locks once its NCO is so updated.

Before arriving at the above procedure, which seems to be 100% reliable, I just had to retry acquistion until the carrier loop locked. Fortunately, Doppler shift is constantly changing, and if one attempt failed, the next would often succeed. In stubborn cases, nudging the carrier NCO up or down by half an FFT bin-width proved effective.

Carriers close to the original IF centre frequency of 2.5 MHz were difficult to acquire, due to fractional spurs on the NCO. A huge improvement was obtained by shifting the IF frequency up 100 KHz. The first local oscillator was changed to 1552.82 MHz, moving the first and second IF frequencies to 22.6 MHz and 2.6 MHz respectively.

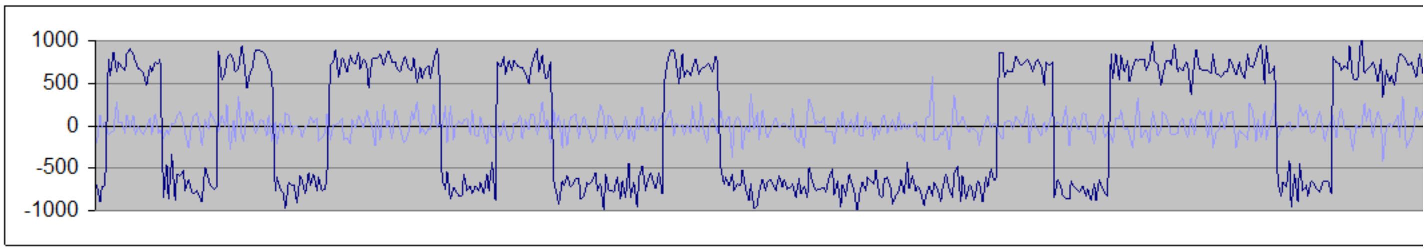




These spectra show the carrier NCO set 50 Hz above IF centres of 2.5 and 2.6 MHz. The original centre frequency was one quarter of the sampling rate. The spurs are safely further away when the frequencies are not in a simple ratio.

NAV data

NAV data is taken from the sign-bit of the Costas Loop I-arm. The Q-arm should look like random noise. Below are 512 raw I_p, Q_p samples @ 1 KHz sampling rate:



The following fragment of NAV data was received at 21:46:45 GMT on Tuesday 1st February 2011:

```
10001011 00001001010101 00 110100 01010001110001111 01 001 11 101100 1001010101010000000000000001011110001011001111111111001011000101011111110101001101101001101111000101011011000011011100000101100
10001011 00001001010101 00 110100 01010001110010000 01 010 01 100000 001101111111100000011011110001000101100110010001001110000001000100011000110100111100000011110000110110101010111101111000010101
10001011 00001001010101 00 110100 01010001110010001 01 011 11 101000 0000000001100011110110100010001010100111100101100111000000010000000000011000011011000110010101001100010110001110011010110001000
10001011 00001001010101 00 110100 01010001110010010 01 100 01 010100 01111111101010011001100101011010101001101010011001100101000010011010100110011010100111010101010110011001100110011010011010011
```

```
10001011 00001001010101 00 110100 01010001110010011 01 101 11 011100 011100110110001101010101000001000000111111111111111111101111111111111111111111111111111111111111111111111111111111111111111
10001011 00001001010101 00 110100 01010001110010100 01 001 10 010100 10010101010100000000000000101111000101100111111111110010110001010111111101010011011010011011110001010110110000110111000001011100
10001011 00001001010101 00 110100 01010001110010101 01 010 11 111100 00110111111110000001101111000100010110011001000100111000001000100011000110100111100000011110000110110101010111101111000010101
10001011 00001001010101 00 110100 01010001110010110 01 011 10 001100 0000000001100011110110100010001010100111100101100111000000010000000000110000110110001100101010011000101100011100110101100010100
10001011 00001001010101 00 110100 01010001110010111 01 100 11 001000 0111100101011110000110000011000111100111010111000110001110000100010000000110100101110101011101111011111000111000101011011110
10001011 00001001010101 00 110100 01010001110011000 01 101 00 111100 0100000010101010101010101011001010101010101010101010101111001010101010101010101010101111001010101010101010101010101111001010
```

The above are 2 consecutive frames of 5 subframes each. Subframes are 300-bits long and take 6 seconds to transmit. Column 1 is the preamble 10001011. This appears at the start of every subframe; but can occur anywhere in the data. The 17-bit counter in column 5 is time-of-week (TOW) and resets to zero at midnight Sunday. The 3-bit counter in column 7 is the subframe ID 1 through 5. Subframes 4 and 5 are subcommutated into 25 pages each and a complete data message comprising 25 full frames takes 12.5 minutes to transmit. I am only using data in subframes 1, 2 and 3 at present.

Solving for user position

Every GPS satellite transmits its position and the time. Subtracting time sent from time received and multiplying by the speed of light is how a receiver measures distance between itself and the satellites. Doing so with three satellites would yield three simultaneous equations in three unknowns (user position: x, y, z) if the precise time was available. In practice, receiver clocks are not accurate enough, the exact time is a fourth unknown, four satellites are therefore required and four simultaneous equations must be solved:

$$
\begin{aligned}
PR_1 &= ct_b + \sqrt{(x-x_1)^2 + (y-y_1)^2 + (z-z_1)^2} \\
PR_2 &= ct_b + \sqrt{(x-x_2)^2 + (y-y_2)^2 + (z-z_2)^2} \\
PR_3 &= ct_b + \sqrt{(x-x_3)^2 + (y-y_3)^2 + (z-z_3)^2} \\
PR_4 &= ct_b + \sqrt{(x-x_4)^2 + (y-y_4)^2 + (z-z_4)^2}
\end{aligned}
$$

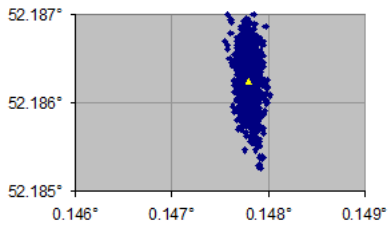
An iterative method is used because the equations are non-linear. Using earth's centre (0, 0, 0) and the approximate time as a starting point, the algorithm converges in only five or six iterations. The solution is found even if user clock error is large. The satellites carry atomic clocks; but these too have errors and correction coefficients in subframe 1 must be applied to the time of transmission. Typical adjustments can be hundreds of microseconds.

The uncorrected time of transmission is formed by scaling and adding several counters. Time-of-week (TOW) in seconds since midnight Sunday is sent every subframe. Data edges mark out 20ms intervals within 300-bit subframes. The code repeats 20 times per data bit. Code length is 1023 chips and chip rate is 1.023 Mbps. Finally, the 6 most significant bits of the code NCO phase are appended, fixing time of transmission to ± 15ns.

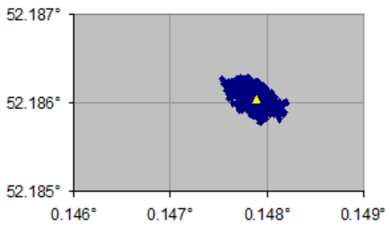
Satellite positions at the corrected transmission time are calculated using ephemeris in subframes 2 and 3. Orbital position at a reference time t_oe (time of ephemeris) is provided along with parameters allowing (x,y,z) position to be calculated up to a few hours before or after. Ephemerides are regularly updated and satellites only transmit their own. Long term orbits of the entire constellation can be predicted less accurately using Almanac data in subframes 4 and 5; however, this is not essential if a fast FFT-based search is used.

Solutions are computed in earth-centred, earth-fixed (ECEF) coordinates. User location is converted to latitude, longitude and altitude with a correction for eccentricity of the earth, which bulges at the equator. The scatter diagrams below illustrate repeatability, the benefit of averaging and the effect of poor satellite choices. Grid squares are 0.001° on each side. Blue dots mark 1000 fixes. Yellow triangles mark the centres of gravity:

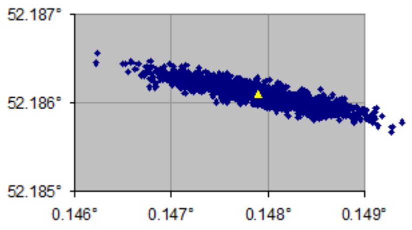
(i) North-facing window ledge



(ii) Rooftop antenna



(iii) East-facing window ledge



The tight cluster (ii) was obtained using satellites in four different quarters of the sky. Only the rooftop antenna had a clear view in all directions. But good fixes were obtained by averaging, even when half the sky was obscured. Rooftop fixes also exhibit spreading like (i) and (iii) if the wrong satellites are chosen.

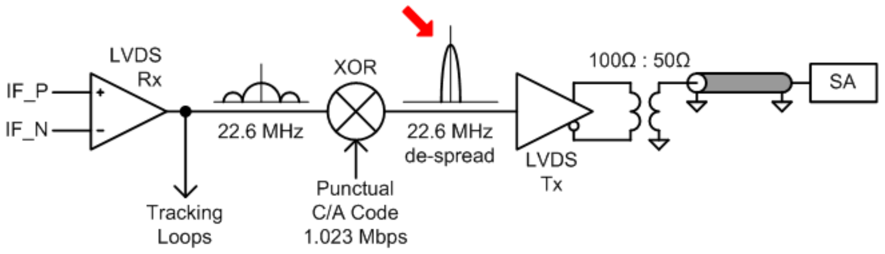
The above solutions were generated without compensating for ionospheric propagation delays using parameters in page 18 of subframe 4 which should be applied because this is a single frequency receiver. Ionospheric refraction increases path lengths between users and satellites.

In April 2012, I fixed a bug that caused significant errors in user-position solutions. Originally, by not transforming satellite positions from earth-centred-earth-fixed (ECEF) to earth-centred-inertial (ECI) coordinates, I was effectively ignoring Earth's rotation during the 60 to 80 ms that signals were in flight. I am now seeing positional solution accuracies of ± 5 metres after averaging, even with limited satellite visibility.

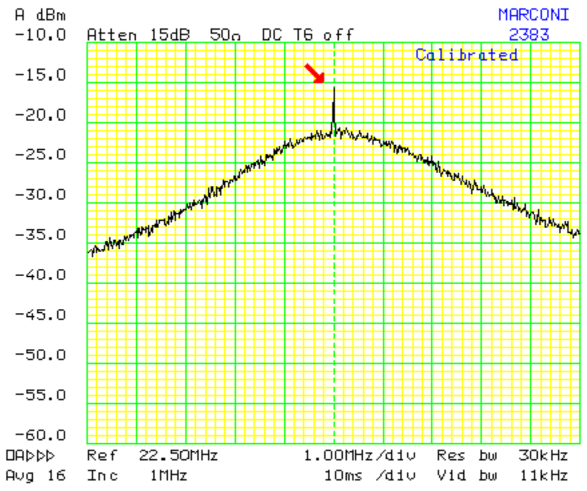
I've created an appendix showing how the iterative solution is developed, starting from a geometric range equation, which is linearised using a Taylor Series expansion, and solved by matrix methods, for the special case of four satellites or the general case of more, with the option of using weighted least-squares to control the influence of particular satellites. You'll find this and solution "C" source code in the links at the bottom of the page.

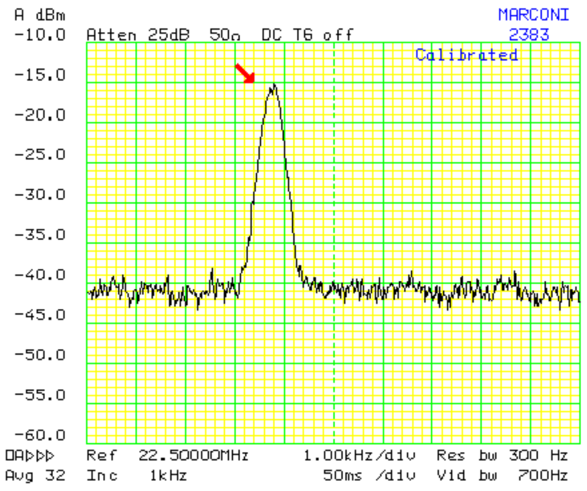
I'm grateful to Dan Doberstein for sending me an early draft of his GPS book [2] which helped me understand the solution algorithm. The official US government GPS Interface Specification [3] is an essential reference.

Signal monitor



The above circuit arrangement, mostly implemented in FPGA, de-spreads by taking the product of the 1-bit IF and punctual code, leaving 50 bps data modulation. A small notch due to BPSK carrier suppression can just be seen:





These spectra show the same de-spread transmission at different spans and resolution bandwidths (RBW). Doppler shift was -1.2 KHz. The noise floor is antenna thermal noise amplified and filtered by the IF strip. -3 dB bandwidth looks around 3 MHz, slightly wider than planned. The de-spread carrier is 5 dB above noise at 30 KHz RBW and 25 dB above at 300 Hz RBW. Received signal strength at the antenna can be estimated as -174+1+10*log10(30e3)+5 = -123 dBm.

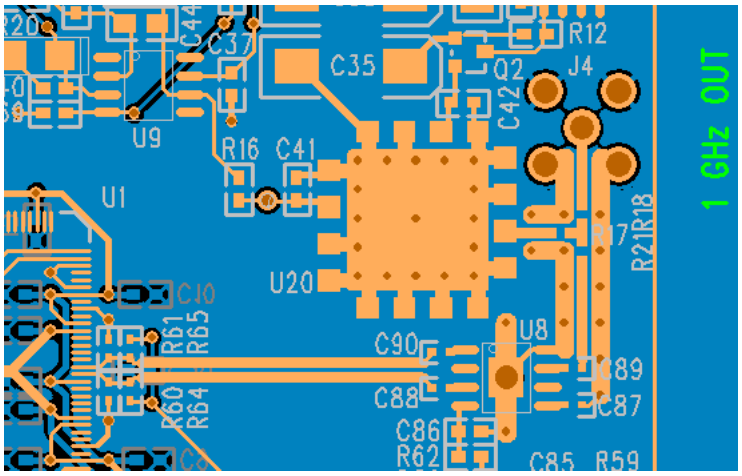
It still amazes me how well frequency domain information is preserved through hard-limiting! The LVDS transmitter has a constant output current of ~3mA which is ~1mW in 100 ohms. Peak power seen at the SA cannot exceed 0 dBm. Here, we see this available power spread across a range of frequencies. Wideband integrated power spectral density must be ~1mW.

First local oscillator

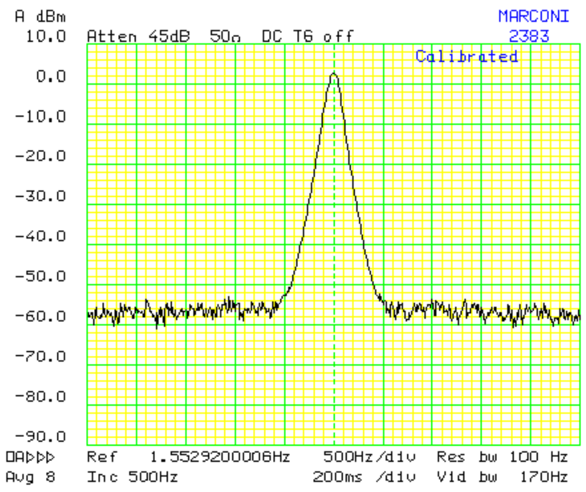
I've been building experimental fractional-N synthesizers using general-purpose programmable logic for several years:

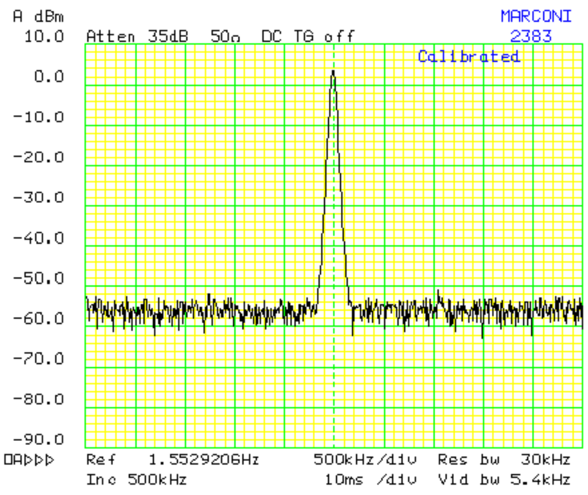
Project	Date	Technology	Frequency (MHz)
FracN	2004	Altera MAX 7000 CPLD	4.3
Frac2	2005	Altera MAX 7000 CPLD	15 - 25
Frac3	2009	Xilinx Spartan 3 FPGA	38 - 76
Frac4	2009	Xilinx Spartan 3 FPGA	38 - 76
Frac5	2010	Xilinx Spartan 3 FPGA	800 - 1600
Frac7	2013	Xilinx Spartan 3 FPGA	1500 - 1600

Frac7 was built for this purpose; but I had no idea Frac5 would be used in a GPS receiver when I originally designed it. The photo below shows how the ROS-1455 VCO output on Frac5 was resistively split between the output SMA and a Hittite HMC363 divide-by-8 prescaler. The 200 MHz divider output is routed (differentially) into the FPGA which phase locks it to a master reference using methods documented in my earlier projects. Microwave circuity on Frac7 is similar; but uses a Mini-Circuits 3dB splitter.



High stability and low phase noise are achieved, as can be seen in the VCO output spectra shown below. When Frac5 was originally developed, as a dedicated frequency synthesizer, simultaneous toggling on frequencies not harmonically related was avoided to minimise intermodulation spurs. The FPGA was static when clock pulses that toggled phase detector output crossed the fabric. No such luxury is practical when the FPGA is hosting a GPS receiver; however, fortunately, the local oscillator output is good enough:





The Marconi 2383 spectrum analyser's 50 MHz STD OUTPUT was used as the master reference source for Frac5 and all internal GPS receiver clocks. GPS receivers need accuracies better than 1 ppm (parts per million) to measure ±5 KHz Doppler shifts on the 1575.42 MHz L1 carrier. Any frequency uncertainty would necessitate a wider search range.

Embedded CPU

My original GPS receiver could only track 4 satellites. The available fabric was not used efficiently and the FPGA was full. Identical logic was replicated for each channel and only clock-enabled at the 1 KHz code epoch. GPS update rates are quite un-demanding and most of the "parallel" processing can easily be done sequentially. Embedding a CPU for this task has both increased the number of channels and freed space in the FPGA.

This CPU directly executes FORTH primitives as native instructions. Visitors to my Mark 1 FORTH Computer page will already be aware of my interest in the language. FORTH is not mainstream; and its use here might be an esoteric barrier; however, I could not resist doing another FORTH CPU, this time in FPGA, after seeing the excellent J1 project, which was an inspiration.

FORTH is a stack-based language, which basically means the CPU has stacks instead of general purpose registers. Wikipedia has a good overview.

Features

- FPGA resources: 360 slices + 2 BRAMs
- Single-cycle instruction execution

- FORTH-like dual-stack architecture

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

- 32-bit stack and ALU data paths
- 64-bit double-precision operations
- Hardware multiplier
- 2k byte (expandable to 4k byte) code and data RAM
- Macro assembler code development

Memory and I/O

Two BRAMs are used: one for main memory, the other for stacks. Xilinx block RAM is dual ported, allowing one instance to host both data and return stacks. Each stack pointer ranges over half of the array. Dual porting of the main memory permits data access concurrent with instruction fetch. One memory port is addressed by the program counter, the other by T, the top of stack. Writes to the PC-addressed port are also used for code download, the program counter providing incrementing addresses.

Code and data share the main memory, which is organised as 1024 (expandable to 2048) 16-bit words. Memory accesses can be 16-, 32- or 64-bits, word-aligned. All instructions are 16-bit. Total code plus data size of the GPS application is less than 750 words, despite all loops being unrolled.

I/O is not memory-mapped, occupying its own 36 bit-select space (12 in + 12 out + 12 events). One-hot encoding is used to simplify select decoding. I/O operations are variously 1-bit serial, 16- or 32-bit parallel. Serial data shifts 1 bit per clock cycle. Events are used mainly as hardware strobes and differ from writes by not popping the stack.

Instruction format

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op_push	0	literal [14:0]														
op_*	1	0	0	opcode [12:8]					ret	operand(s)						
op_call	1	0	1	0	destination_address [11:1]											0
op_branch	1	0	1	0	destination_address [11:1]											1
op_branchZ	1	0	1	1	destination_address [11:1]											0
op_branchNZ	1	0	1	1	destination_address [11:1]											1
op_rdReg	1	1	0	0	input_select [11:0]											
op_wrReg	1	1	0	1	output_select [11:0]											
op_wrEvt	1	1	1	0	event_select [11:0]											

24 instructions out of a possible 32 are currently allocated in the opcode space h80XX - h9FXX. These are mostly zero-operand stack / ALU operations. The "ret" option, which performs return from subroutine, executes in parallel, in the same cycle. Add-immediate is the only one-operand instruction. A carry-in option extends (stack, implied) addition precision. hF0000 - hFFFF is spare.

Precision

Stack and ALU data paths are 32-bit; however, 16-, 32- and 64-bit operations are supported. 64-bit values occupy two places on the stack, with least significant bits on top. Top of stack, T, and next on stack, N, are registered outside the BRAM for efficiency. Apart from the 64-bit left shift (op_shl64) which is hard-wired for single-cycle execution, all other double precision functions are software subroutines.

Assembly language

The GPS embedded binary was created using Microsoft's Macro Assembler MASM. This only supports x86 mnemonics; but opcodes are declared using equ and code is assembled using "dw" directives. MASM not only provides label resolution, macro expansion and expression evaluation but even data structures! The MASM dup() operator is used extensively to unroll loops e.g. `dw N dup(op_call + dest)` calls a subroutine N times.

This fragment gives some flavour of source style. Stack-effect is commented on every line:

```
op_store64      equ op_call + $             ; [63:32] [31:0] a         17 cycles
                dw op_store32               ; [63:32] a
                dw op_addi + 4              ; [63:32] a+4
                ; drop through
op_store32      equ op_call + $             ; [31:0] a                  8 cycles
                dw op_over                  ; [31:0] a [31:0]
                dw op_swap16                ; [15:0] a [31:16]
                dw op_over                  ; [15:0] a [31:16] a
                dw op_addi + 2              ; [15:0] a [31:16] a+2
                dw op_store16, op_drop      ; [15:0] a
                dw op_store16 + opt_ret     ; a
```

- `op_fetch16` and `op_store16` are primitives.
- `op_store32` and `op_store64` are subroutines or "compound instructions" usable as if they were primitives.
- T is actually `[15:0,31:16]` after `op_swap16`, but we don't care about the upper 16-bits here.
- `op_store16` leaves the address; stack depth can only change ±1 per cycle.
- Purists might prefer: `dw N + addi`

Host serial interfaces

The FPGA can be controlled via SPI by the Raspberry Pi, or by a Windows PC using a Xilinx Platform USB JTAG cable. There are two levels of request priority:

Priority	SPI select	JTAG IR	Function
Highest	SPI_CS[0]	USER1	Halt embedded CPU and load new code image
Lowest	SPI_CS[1]	USER2	Send new command and poll for response to previous

New code images are copied to main memory via a third BRAM which bridges the CPU and serial clock domains. Thus downloaded, binary images execute automatically. Host commands are captured in the bridge BRAM and the CPU is signalled to action them. Its responses are collected by the host from the bridge on the next scan.

The top-level main loop polls for host service requests. The first word of any host message is a command code. Requests are dispatched through the `Commands` jump table:

```
                dw op_rdReg + JTAG_RX       ; cmd
                dw op_shl                   ; offset
                dw Commands, op_add         ; &Commands[cmd]
                dw op_fetch16               ; vector
                dw op_to_r, op_ret
```

- `op_to_r` moves `vector` to the return stack.

Some host requests (e.g. CmdGetSamples) elicit lengthy responses. Data ports on the CPU side of the bridge are 16-bit. The CPU can read and write these via the data stack; however, more direct paths exist for uploading main memory and GPS IF samples. The instruction `op_wrEvt + GET_MEMORY` transfers a memory word directly to the bridge, using T as an auto-incrementing pointer. GET_MEMORY is the only event which has stack effect. The instruction `op_wrEvt + GET_SAMPLES` transfers 16 bits from the IF sampler:

```
UploadSamples:  dw 16 dup (op_wrEvt + GET_SAMPLES)      ;  16*16 =  256 samples copied
                dw op_ret


CmdGetSamples:  dw op_wrEvt + JTAG_RST                  ; addr = 0
                dw 16 dup (op_call + UploadSamples)     ; 256*16 = 4096 samples copied
                dw op_ret
```

Unrolling loops at assembly time with `dup()` trades code size for performance, avoiding a decrement-test-branch hit; and the entire application binary is still tiny; however, long loops must be nested, as illustrated above.

CHANNEL data structure

An array of structures holds state variables and buffered NAV data for the channels. MASM has excellent support for data structures. Field offsets are automatically defined as constants and the `sizeof` operator is useful.

```
MAX_BITS        equ 64

CHANNEL         struct
ch_NAV_MS       dw ?                        ; Milliseconds 0 ... 19
ch_NAV_BITS     dw ?                        ; Bit count
ch_NAV_PREV     dw ?                        ; Last data bit = ip[15]
ch_NAV_BUF      dw MAX_BITS/16 dup (?)      ; NAV data buffer
ch_CA_FREQ      dq ?                        ; Loop integrator
ch_LO_FREQ      dq ?                        ; Loop integrator
ch_IQ           dw 2 dup (?)                ; Last IP, QP
ch_CA_GAIN      dw 2 dup (?)                ; KI, KP
ch_LO_GAIN      dw 2 dup (?)                ; KI, KP
CHANNEL         ends

Chans:          CHANNEL NUM_CHANS dup (<>)
```

The epoch service routine (labelled `Method:`) is called with a pointer to a CHANNEL structure on the stack. Affecting OO-airs, stack-effect comments refer to it as "this" throughout the routine. A copy is conveniently kept on the return stack for accessing structure members like so:

```
                dw op_r                     ; ... this
                dw op_addi + ch_NAV_MS      ; ... &ms
                dw op_fetch16               ; ... ms
```

The `Chans` array is regularly uploaded to the host.

Raspberry Pi application software

The Raspberry Pi software is multi-tasked using what are variously known as coroutines, continuations, user-mode or light-weight threads. These co-operatively yield control, in round-robin fashion, using the "C" library `setjmp/longjmp` non-local goto, avoiding the cost of a kernel context-switch:

```
void NextTask() {
    static int id;
    if (setjmp(Tasks[id].jb)) return;
    if (++id==NumTasks) id=0;
    longjmp(Tasks[id].jb, 1);
}
```

Up to 16 threads can be active:

Source	Instances	Purpose
main.cpp	1	Initialisation; polling joystick; exit
user.cpp	1	User interface
search.cpp	1	Signal detection
channel.cpp	12	Acquisition, tracking and NAV data collection
solve.cpp	1	User-position solution

Coding as threads, each responsible for one task, produces more readable code. Other source files include:

Source	Purpose
auto.sh	Auto-start; and shutdown properly on exit
cacode.h	PRBS generator
coroutines.cpp	User-mode threading
ephemeris.*	Ephemeris database
gps.h	Main header file
peri.cpp	BCM2835 peripherals
Print.h	Base class for LCD driver
spi.*	SPI interface to FPGA

There is no Arduino in this project, but its LCD driver files LiquidCrystal.cpp and LiquidCrystal.h are used.

Source code

2013 (Latest, re-released under GNU General Public License)

- ASM Embedded CPU FORTH
- Verilog Spartan 3 FPGA
- C++ Raspberry Pi

Older versions (Win32 C++)

- 2012
- 2011

Schematics

- GPS3 front-end
- Frac7 FPGA

Links and resources

- User position solution derivation
- N2YO live tracking of GPS satellites above your horizon
- Celestrak daily GPS ephemeris TLE (two line element) updates
- SPACETRACK Report No. 3 mathematical models for processing orbital elements
- STSPlus orbital tracking software
- How to locate the preamble in NAV data
- PyEphem Python library for astronomical computations
- Doppler.py Python script for predicting GPS satellite Doppler shifts
- www.gps.gov/technical/icwg GPS documentation

References

1. GPS/GLONASS receiver Matjaž Vidmar, S53MV
2. PRINCIPLES OF GPS RECEIVERS - A HARDWARE APPROACH by Dan Doberstein
3. IS-GPS-200E GPS Interface Specification

andrew@aholme.co.uk