

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization International Bureau



(43) International Publication Date  
16 September 2004 (16.09.2004)

PCT

(10) International Publication Number  
**WO 2004/079978 A2**

(51) International Patent Classification<sup>7</sup>:

**H04L**

(21) International Application Number:

PCT/US2004/006064

(22) International Filing Date: 1 March 2004 (01.03.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:  
60/451,336 28 February 2003 (28.02.2003) US

(71) Applicant (for all designated States except US): **RGB NETWORKS, INC.** [US/US]; 2929 Campus Drive, Suite 165, San Mateo, CA 94403 (US).

(72) Inventor; and

(75) Inventor/Applicant (for US only): **MONTA, Peter** [US/US]; 1540 Oak Creek Drive, Apt. 405, Palo Alto, CA 94304 (US).

(74) Agent: **HOWARD, M., Cohn**; Patent & Trademark Attorney LLC, 21625 Chagrin Boulevard, Suite 220, Cleveland, OH 44122 (US).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

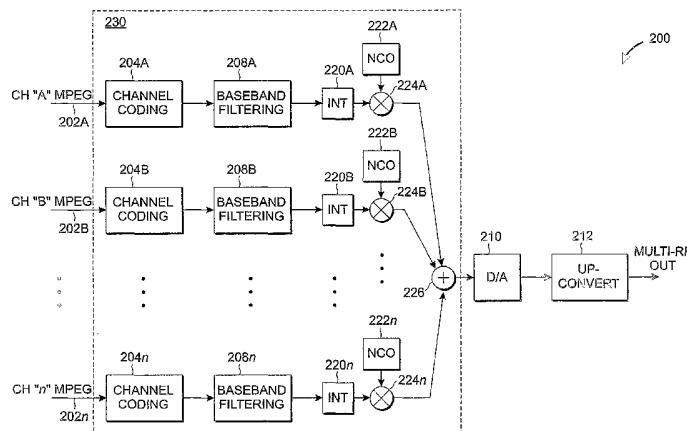
(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— without international search report and to be republished upon receipt of that report

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: COST-EFFECTIVE MULTI-CHANNEL QUADRATURE AMPLITUDE MODULATION



**WO 2004/079978 A2**

(57) Abstract: A highly-efficient, cost-effective technique for multi-channel QAM modulation is described. The technique employs an inverse fast-Fourier transform (IFFT) as a multichannel modulator.. QAM encoding expresses QAM symbols as constellation points in the complex plane such that each QAM symbol represents a specific phase and amplitude of a carrier frequency to which it is applied. In multi-channel systems, the carrier frequencies are generally uniformly spaced at a channel-spacing frequency (6MHz, for digital cable systems in the United States). The IFFT accepts a set of complex frequency inputs, each representing the complex frequency specification (i.e., phase and amplitude) of a particular frequency. The inputs are all uniformly spaced, so assuming that the IFFT is sampled at a rate to provide the appropriate frequency spacing between its frequency-domain inputs, the IFFT will produce a time domain representation of QAM symbols applied to its various inputs modulated onto carriers with the desired channel separation. Since the channel spacing and the symbol rate are different due to excess channel bandwidth, interpolation is used to rectify the difference. An efficient scheme for combining this interpolation with baseband filtering and anti-imaging filtering is described.

## **COST-EFFECTIVE MULTI-CHANNEL QUADRATURE AMPLITUDE MODULATION**

### **CROSS-REFERENCE TO RELATED APPLICATIONS**

[0001] This application claims the benefit of U.S. Provisional Patent Application No. 60/451,336 filed on February 28, 2003 which is incorporated herein by reference.

### **TECHNICAL FIELD OF THE INVENTION**

[0002] The present invention relates to digital data transmission systems, more particularly to multi-channel distribution of digitally-encoded data streams over a cable, optical fiber or similar transmission medium, and still more particularly to multi-channel QAM modulation of digital television data and related data sources.

### **BACKGROUND**

[0003] Over the last several years, there has been considerable growth in the availability of digital cable and satellite television broadcasting. As demand for digital programming continues to grow, cable television providers are transitioning from analog cable transmission systems and converters to mixed analog/digital and all-digital cable distribution systems. Increasing competition from digital satellite service providers has contributed to increased demand for more and different digital cable services including digital data services, interactive programming services and "on-demand" services like video-on-demand (VOD). A high-end variant of VOD, "everything-on-demand" (EOD) offers a dedicated, full-time video and audio stream for every user. An EOD stream can be used to view time-shifted TV, movies, or other content stored by content providers at the headend of the network, with full VCR-like controls such as pause, fast forward, random access with "bookmarks", etc.

- [0004] In combination with other services like interactive programming, cable Internet services, etc., these per-user services require considerably more infrastructure than do pure broadcast services. These newer, high-end services require a server subsystem to provide dynamically customized multi-program multiplexes on a per-user basis. Clearly, this requires a great deal of high-speed, high-performance processing, data routing, encoding and multiplexing hardware that would not otherwise be required.
- [0005] As demand continues to grow for these high-end, per-user services, there is a growing need for more efficient, more cost-effective methods of creating large numbers of custom program multiplexes.

## SUMMARY OF THE INVENTION

- [0006] The present inventive technique provides a highly efficient, cost-effective technique for multi-channel QAM modulation by employing an inverse fast-Fourier transform (IFFT) as a multi-channel modulator. QAM encoding expresses data symbols as constellation points in the complex plane space such that each QAM symbol represents a specific phase and amplitude of a carrier frequency to which it is applied. In multi-channel systems, the carrier frequencies are generally uniformly spaced at a channel-spacing frequency (6MHz, for digital cable systems in the United States). The IFFT, acting as a synthesis uniform filterbank, accepts a set of frequency domain inputs, each representing a 6MHz subband. The inputs are all uniformly spaced, so assuming that the IFFT is sampled at a rate to provide the appropriate frequency spacing between its frequency-domain inputs, the IFFT will produce a time domain representation of QAM symbols applied to its various inputs modulated onto carriers with the desired channel separation.
- [0007] Typically, baseband filtering is applied to the QAM input streams to shape the baseband spectrum and, in cooperation with the receiver filtering, control inter-symbol interference. Also, anti-imaging filters are applied to the IFFT output to ensure proper channel separation.
- [0008] According to an aspect of the invention, a typical multi-channel QAM modulator includes QAM encoding means, inverse FFT (IFFT) processing means, D/A conversion and upconversion. The QAM encoding means encode multiple digital input streams into multiple corresponding QAM symbol streams. The IFFT creates the desired modulation and channel spacing of the QAM symbol streams in an intermediate complex baseband, in digital form. The D/A conversion means convert the digital output from the IFFT conversion process into analog form, and the up-conversion means frequency shift the resultant multi-channel IF QAM signal up to a target frequency band to realize a multi-RF output for transmission.

- [0009] According to an aspect of the invention, the digital data streams can be 256-QAM or 64-QAM encoded according to ITU specification J.83 Annex B.
- [0010] According to an aspect of the invention, baseband filtering, anti-imaging and interpolation are all combined into a single post-IFFT time-varying digital filter stage.
- [0011] In combination, then, one embodiment of a multi-channel QAM modulator for modulating a plurality of digital data streams onto a single multi-output is achieved by means of a set of QAM encoders, IFFT processing means, post-IFFT combined filtering means, D/A conversion means and up-converter means. The QAM encoders provide QAM symbol stream encoding of the digital data input streams. As described previously, IFFT processing performs parallel multi-channel QAM modulation in an intermediate frequency band. Post-IFFT combined filtering effectively combines baseband filtering, anti-imaging filtering and rate interpolation into a single filtering stage. The D/A conversion converts IF output from the Post-IFFT filtering means from digital to analog form and the up-converter means frequency shifts the resultant analog signal into a target frequency band on a multi-RF output.
- [0012] According to an aspect of the invention, digital quadrature correction means can be employed in the digital domain to pre-correct/pre-compensate for non-ideal behavior of the analog up-converter means.
- [0013] According to another aspect of the invention, digital offset correction can be employed in the digital domain to pre-correct for DC offsets in the analog D/A conversion and up-converter means.
- [0014] The present inventive technique can also be expressed as method for implementation on a Digital Signal Processor, FPGA, ASIC, or other processor.

- [0015] According to the invention, multi-channel QAM modulation can be accomplished by providing a plurality of digital data input streams, encoding each of the digital data streams into a set of QAM-encoded streams, processing the QAM-encoded streams via an inverse FFT (IFFT) to modulate the plurality of QAM-encoded streams into a single digital multi-channel IF stream encoding the multiple QAM encoded streams onto a set of uniformly spaced carrier frequencies in an intermediate frequency band, converting the digital multi-channel IF stream to analog form; and frequency-shifting the analog multi-channel IF stream to a target frequency band onto a multi-RF output.
- [0016] According to another aspect of the invention, the digital multi-channel IF stream can be post-IFFT filtered via a combined baseband and anti-imaging filter.
- [0017] According to another aspect of the invention, the digital multi-channel IF stream can be interpolated to compensate for any difference between the QAM symbol rate and the channel spacing (sample rate).
- [0018] According to another aspect of the invention, the digital multi-channel IF stream can be digitally quadrature corrected to pre-correct for non-ideal behavior of the frequency shifting process (in particular, the errors in an analog quadrature modulator).
- [0019] According to another aspect of the invention, digital offset correction can be applied to compensate for DC offsets, in the digital-to-analog conversion and frequency-shifting processes.

## BRIEF DESCRIPTION OF THE DRAWINGS

- [0020] Reference will be made in detail to preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. The drawings are intended to be illustrative, not limiting. Although the invention will be described in the context of these preferred embodiments, it should be understood that it is not intended to limit the spirit and scope of the invention to these particular embodiments.
- [0021] The structure, operation, and advantages of the present preferred embodiment of the invention will become further apparent upon consideration of the following description taken in conjunction with the accompanying drawings, wherein:
- [0022] **Figure 1** is a block diagram of a multi-channel Quadrature Amplitude Modulation (QAM) modulator, in accordance with the prior art.
- [0023] **Figure 2** is a block diagram of a direct translation of the multi-channel QAM modulator of Figure 1 to digital form.
- [0024] **Figure 3** is a block diagram of an all-digital multi-channel QAM modulator employing an Inverse Fast Fourier Transform, in accordance with the invention.
- [0025] **Figure 4** is a block diagram of a simplified version of the multi-channel QAM modulator of Figure 3, in accordance with the invention.
- [0026] **Figure 5** is a block diagram of a preferred embodiment of a 16-channel QAM modulator, in accordance with the invention.

**DETAILED DESCRIPTION OF THE INVENTION**

- [0027] The present inventive technique provides an efficient, cost-effective means of multiplexing multiple "channels" of digital television and other data onto a single transmission medium.
- [0028] Most prior-art multi-channel QAM modulators are generally organized as shown in Figure 1, which shows a system 100 of separate channel modulators being combined (summed) via an RF combiner 114 to produce a multi-channel RF output signal (Multi-RF Out). In Figure 1, MPEG data streams 102A, 102B, ..., 102 $n$  corresponding to " $n$ " separate program sources are each encoded by a respective channel coder 104A, 104B, ..., 104 $n$  to produce a respective QAM "symbol" stream 106A, 106B, ..., 106 $n$  representing the MPEG data streams 102A, 102B, ..., 102 $n$ . Each QAM symbol stream is encoded according to a suitable standard for digital cable television QAM stream encoding (e.g., ITU-T J.83 Annex A or Annex B, provided by the International Telecommunications Union, Geneva, Switzerland) whereby each QAM "symbol" represents one of a set of pre-defined phase/amplitude "constellation points" in complex frequency space. For example, 256-QAM defines a rectangular 16x16 array of constellation points in the complex plane. Each constellation point in the array represents a unique 8-bit binary value encoded at a specific carrier amplitude and phase angle. 64-QAM defines an 8x8 rectangular array of constellation points.
- [0029] According to the United States frequency plan for digital cable television, channels are spaced in 6 MHz intervals, and are encoded at a symbol rate of 5.360537 Mbaud in the case of 256-QAM. (Other QAM modulation schemes such as 64-QAM and 1024-QAM are encoded at different symbol rates). Baseband filters 108A, 108B, ..., 108 $n$  each receive a respective encoded 5.360537 Mbaud QAM symbol stream 106A, 106B, ..., 106 $n$  and perform general channel "shaping". (Most European systems operate at 8 MHz channel spacing). Outputs from the baseband filters 108A, 108B, ..., 108 $n$  are then converted by respective digital-to-

analog (D/A) converters 110A, 110B, ..., 110 $n$  from digital to analog. Analog outputs from the D/A converters 110A, 110B, ... 110 $n$  are each up-converted by a respective up-converter 112A, 112B, ..., 112 $n$  to a respective channel frequency. Each up-converter 112 $'x'$  frequency-shifts an analog QAM-encoded stream from a respective D/A converter 110 $'x'$  to a specific channel frequency. Outputs from the up-converters 112A, 112B, ... 112 $n$  are then combined (summed) onto a single multi-RF output by the RF combiner 114 for subsequent transmission over a suitable coaxial cable, fiber or hybrid fiber/coax (HFC) signal distribution network.

- [0030] Those of ordinary skill in the art will immediately recognize that although inputs to the multi-channel modulator of Figure 1 are shown as MPEG data streams, any suitable digital information source for which QAM or similar encoding can be defined may be employed. One example is DOCSIS data (Data Over Cable Service Interface Specification) whereby digital communications such as Internet communications can be encoded onto a digital cable television transmission medium. DOCSIS uses the MPEG transport stream as a convergence sublayer.
- [0031] This multi-channel modulator 100 of Figure 1 suffers from some inherent inefficiencies. First, the digital-to-analog (D/A) conversion happens too early in the process, and operates only on relatively low-bandwidth baseband streams. As a result, the relatively high sampling-rate capability of most modern D/A converters is wasted. Second, the up-converters each process only a single channel, occupying a tiny 6 MHz slice of the frequency spectrum. This results in poor converter utilization and high cost.
- [0032] While the availability of a separate up-converter for each 6 MHz channel allows for tremendous frequency agility in that each channel can be placed independently of the others, this agility is not required by present-day applications, and is not envisioned for any future digital cable applications. Blocks of contiguous channels provide adequate flexibility for spectrum planning. (A user's set-top box does not care which RF channel is carrying a program; RF channels can be allocated almost

completely arbitrarily among the spectrum channel slots, limited only by operational convenience.)

- [0033] One approach to improving the cost-effectiveness of the multi-channel modulator of Figure 1 is to translate as many of its analog components as possible – primarily the up-converters – into their digital equivalents and to move them back "behind" a single D/A converter. This greatly improves D/A converter utilization and eliminates the discrete up-converters. In this approach, numerically-controller oscillators (NCOs) would perform the function of local oscillators (LOs), digital multipliers would perform the function of doubly-balanced mixers, a digital adder would replace the analog RF combiner and digital filters would be employed to interpolate between the baseband channel QAM symbol rate (for example, 5.360537 Mbaud for 256-QAM) and a 6MHz digital conversion rate that facilitates implementation of the 6MHz channel spacing. This approach assumes that the additional cost of implementation of the new digital functions will be more than offset by the cost of the eliminated analog functions.
- [0034] Figure 2 is a block diagram of such an implementation. In Figure 2, a multi-channel QAM modulator 200 comprises a digital processing block 230, followed by a single D/A converter 210 and up-converter 212. In the digital processing block 230, channel coders 204A, 204B, ..., 204n (compare 104'x', Fig. 1) receive MPEG stream inputs (or other suitable digital stream data) and encode them according to a set of baseband QAM encoding rules (e.g., 256-QAM). QAM-encoded data from each channer coder 204A, 204B, ..., 204n is then processed by a respective digital baseband filter 208A, 208B, ..., 208n (compare 108'x', Fig. 1). The output of each baseband filter 208A, 208B, ..., 208n is then processed by a respective digital interpolator 220A, 220B, ..., 220n that compensates for the difference between the 5.360537 Mbaud QAM symbol rate and the 6n MHz D/A sample rate, where 'n' is the number of channels. Those of ordinary skill in the art will immediately understand that although the QAM symbol rate and channel

spacing would be different under the European frequency plan, the principles remain the same and the same techniques are readily applied.

- [0035] After interpolation, the output of each interpolator 220A, 220B, ..., 220 $n$  is processed by a respective digital up-converter comprising a respective numerically controlled oscillator (NCO) 222A, 222B, ..., 222 $n$  and a respective digital multiplier 224A, 224B, ..., 224 $n$ . Each NCO 222'x' behaves as a digital equivalent of a local oscillator (LO) and each digital multiplier 224'x' behaves as a digital equivalent of a doubly balanced modulator (DBM or "mixer"). In combination, each NCO/multiplier pair (222'x'/224'x') produces a digital output stream that digitally represents one QAM-coded channel upconverted to a different intermediate frequency. The outputs of the digital multipliers 224A, 224B, ..., 224 $n$  are then summed together in a digital adder 226 to produce a multi-channel digital stream, encoding multiple properly-spaced QAM channels, but in an intermediate frequency (IF) band. This multi-channel digital stream is then converted to analog form by the D/A converter 210. A final up-converter 212 is used to frequency shift the entire analog IF multi-channel stream into the correct frequency band for transmission (Multi-RF out).
- [0036] Two of the most significant factors in the cost of digital signal processing systems are the cost of the digital signal processors (DSPs) themselves and the cost of D/A converters. Semiconductor densities have exhibited an unabated exponential rate of increase for over 40 years. This trend predicts that any DSP-based or digital logic based technique will benefit over time from the increasing density and decreasing cost associated with digital circuitry. D/A converters are following similar density and cost curves, driven in part by the performance demands and high-volume production of digital cellular communications and wireless data communications markets.
- [0037] Digital signal processing techniques can be implemented in a wide variety of technologies, ranging from full-custom dedicated function integrated circuits to

ASICs (Application-Specific Integrated Circuits) to Field-Programmable Gate Arrays (FPGAs). Hardware description languages (HDLs) such as Verilog and VHDL in combination with logic synthesis techniques facilitate portability of digital designs across these various technology platforms. Each technology has its advantages and disadvantages with respect to development cost, unit pricing and flexibility, and all are capable of performing several hundred million digital operations per second.

- [0038] Wideband digital-to-analog converters (also "D/A converters", "D/As" or "DACs") have already reached advanced stages of development. For example, the AD9744 from Analog Devices can convert 165 Ms/s with spur-free dynamic range of 65 dB for a cost of \$11. This sample rate represents hundreds of video users, so the per-user cost is almost negligible.
- [0039] The multi-channel modulator approach shown in Figure 2 can be appropriate for situations where the channels are sparsely distributed over the spectrum, and it can be made fairly efficient by employing multi-rate techniques for the filters, for example, CIC (Cascade Integrator Comb) Filters. The cable-TV spectrum, however, is normally fully populated with uniformly spaced channels. This argues for a more efficient approach.
- [0040] A significant efficiency improvement can be realized by recognizing that QAM encoding on uniformly spaced channels is simply a representation of a plurality of uniformly spaced, independent complex frequency components. This suggests the use of a transform-based technique to accomplish simultaneous up-conversion of a uniformly-spaced array of complex frequencies to a time-domain representation of a composite, multi-channel multiplex, as has been done for many years in applications such as FDM/TDM (Frequency Division Multiplex/Time Division Multiplex) transmultiplexers. By way of example, Fast Fourier Transform (FFT) techniques, a special case of the Discrete Fourier Transform (DFT), are well-known, well-defined, computationally efficient techniques for transitioning

between time domain and frequency domain representations of signals. The Discrete Fourier Transform, which is in turn a special case of the more general continuous Fourier transform, represents a time-varying signal as the linear sum of a set of uniformly spaced complex frequency components. In its inverse form, the inverse DFT (IDFT) transforms a set of uniformly spaced complex frequency components (a frequency "spectrum" array) to its corresponding time-domain representation. The FFT and inverse FFT (IFFT) are computationally optimized versions of the DFT and IDFT, respectively, that take advantage of recursive structure to minimize computation and maximize speed.

- [0041] If the QAM streams are expressed as a set of time-varying complex frequency coefficient pairs (i.e.,  $A \cos \omega_n t + jB \sin \omega_n t$ , represented as a complex number  $[A, jB]$ ) and assigned to a specific position in a complex IFFT's input array, and assuming that the IFFT is scaled and sampled such that the frequency spacing of its input array corresponds to the desired channel spacing, then the IFFT will produce a discrete time domain representation of all of the QAM streams modulated onto a set of uniformly spaced carriers and summed together. The IFFT, therefore, in a single computational block, effectively replaces all of the up-converters and local oscillators (NCOs/multipliers) of Figures 1 and 2.
- [0042] Figure 3 is a block diagram of an IFFT-based implementation of a multi-channel QAM modulator 300. In Figure 3, as in Figures 1 and 2, a plurality 'n' of MPEG input streams (or other suitable digital input stream) 302A, 302B, ..., 302n are QAM encoded by a respective plurality of channel coders 304A, 304B, ..., 304n and are subsequently processed by a respective plurality of baseband filters 308A, 308B, ..., 308n to perform per-channel shaping on QAM-encoded complex frequency symbol streams produced by the channel coders 304x, producing a set of complex frequency components. The resultant baseband-filtered QAM streams are then assigned to a respective complex frequency position in an IFFT input array and processed by an IFFT 340. While a number of transforms are suitable for realizing uniform filterbanks, (for example, discrete cosine transforms (DCTs)),

in the interest of brevity and simplicity only the IFFT is discussed herein. The results of the IFFT 340 are processed by a set of ' $n$ ' anti-imaging filters 342A, 342B, ..., 342 $n$  ( $h_0(z)$ ,  $h_1(z)$ , ...,  $h_{n-1}(z)$ ) to ensure proper channel isolation, and the outputs of the anti-imaging filters 342 $'x'$  are summed together by a digital adder 326 to produce a composite, multi-channel QAM-encoded digital time-domain stream, which is subsequently converted to analog by a D/A converter 310 and frequency shifted by an up-converter 312 into an appropriate frequency band to produce a multi-RF output.

- [0043] The design of the modulator 300 of Figure 3 employs two separate filtering stages: a baseband filtering stage (308 $'x'$  – pre-IFFT) and an anti-imaging filter stage (342 $'x'$  – post-IFFT). Although this scheme can be employed successfully, the split between the filtering stages is awkward and requires considerable attention to the design of the baseband and anti-imaging filters to ensure that their cascaded effect through the IFFT produces the desired results. Further, the use of two separate digital filtering stages is costly in circuitry and/or computations, requiring separate circuitry and/or computations for each stage.
- [0044] This deficiency can be addressed by combining the pre-IFFT baseband filters and post-IFFT anti-imaging filters into a single post-IFFT filter stage. Figure 4 shows a multi-channel QAM modulator implemented in this way.
- [0045] Figure 4 is a block diagram of an IFFT-based multi-channel QAM modulator 400 wherein two-stage baseband filtering and anti-imaging filtering have been combined into single-stage post-IFFT filtering. In Figure 4, as in Figures 1, 2 and 3, a plurality ' $n$ ' of channel MPEG (or other digital data) sources 402A, 402B, ..., 402 $n$  are QAM-encoded by a like plurality of respective channel coders 404A, 404B, ..., 404 $n$ . Unlike the implementation described hereinabove with respect to Figure 3, the QAM-encoded symbol streams are applied directly to the inputs of an IFFT 440, without baseband filtering; therefore the IFFT operates at the QAM symbol rate. Outputs of the IFFT are then processed by a set of ' $n$ ' time-varying

post-IFFT combined channel shaping and anti-imaging interpolation filters 444A, 444B, ..., 444n, ( $g_{0,t}(z), g_{1,t}(z), \dots, g_{n-1,t}(z)$ ) producing filtered outputs that are then summed together by a digital adder 426 to produce a composite digital multi-channel QAM-encoded multiplex in an intermediate frequency (IF) band. This multiplex is then converted to analog form via a D/A converter 410, and frequency shifted to an appropriate frequency band by an up-converter 412 to produce a multi-RF output.

- [0046] The multi-channel modulator 400 of Figure 4 requires that all input channels (402'x') have the same modulation format and symbol rate, since baseband shaping and anti-imaging are combined in a single filter stage. These are reasonable restrictions and are easily accommodated in any modern digital television transmission scenario.
- [0047] Attention is now directed to a preferred embodiment of the invention as shown and described hereinbelow with respect to Figure 5. It should be noted that complex quantities such as complex frequencies or complex time-domain signals (each having two values, a "real" part and an "imaginary" part) are represented in Figure 5 by double-headed arrows. Real values representing single values are represented in Figure 5 by single-headed arrows.
- [0048] Figure 5 is a block diagram of a 16-channel modulator 500 for multi-channel QAM-256 encoding of 16 MPEG signal streams (or any other suitable QAM-256 encodable digital data source, e.g., DOCSIS data) into a multi-channel RF signal for transmission via cable, optical fiber or HFC transmission medium. The converter 500 comprises a digital processing portion 530, a "complex" D/A converter 510 and an up-converter 512 which, in practice, would be implemented as two D/A converters (one for "real" and one for "imaginary") and a quadrature modulator.

- [0049] In Figure 5, a plurality of ' $n$ ' MPEG (or data) streams 502A, 502B, ..., 502 $n$  are QAM-256 encoded according to ITU J.83 annex B to produce a set of complex-frequency QAM symbol representations (indicated by double-headed arrows). A 24 point IFFT function 540 operates at the QAM symbol rate and is employed to convert 24 complex frequency domain inputs to the IFFT 540 into a like number of time-domain outputs. The first four and last four IFFT complex frequency inputs are set to a fixed value of complex "zero" (i.e.,  $(0, j0)$ ). while the complex QAM-encoded streams are applied to the 16 "middle" IFFT inputs. The zero channels create guard bands to ease the requirements on the analog anti-aliasing filters.
- [0050] The 24 outputs of the IFFT function 540 are serialized by a parallel-to-serial (P/S) function 550 that sequentially shifts out successive complex time-domain values (real/imaginary value pairs) from the IFFT. Each IFFT conversion constitutes an IFFT "frame", and the P/S function 550 is organized such that 24 shift-outs occur for each IFFT frame, producing a complex-serial stream output with a frame length of 24.
- [0051] The complex-serial output from the parallel to serial converter 550 is processed by an " $i^{\text{th}}$ " order FIR (Finite Impulse Response) digital filter comprising a plurality of  $i-1$  sequentially-connected delay elements 552, " $i$ " complex digital multipliers 554 and a digital adder 556. Each delay element 552 delays the complex serial output of the previous stage by exactly one complete IFFT frame (i.e., 24 complex values). The output from each of the serially connected delay elements 552, therefore, provides a specific delay tap. Each delay tap (and the input to the serially connected array) is multiplied by a real-valued coefficient ( $h_x$ ) via a respective one of the complex digital multipliers 554. Since the coefficients  $h_x$  are real-valued, the complex multipliers 554 need not deal with complex cross-products and can be simpler than "true" complex multipliers. (Whereas a "true" complex multiplier requires four multiplications and two additions, the simplified complex-times-real multiplier implementation requires only two multiplications

and no additions). The complex product outputs from these multipliers are summed together by the digital adder 556 to produce a filter output.

- [0052] A coefficient generator comprising a direct digital synthesizer 562 (DDS) acting as an address generator for a set of coefficient ROMs 564 cycles through coefficients for the FIR filter in IFFT frame-synchronous fashion, producing a set of "i" coefficient values ( $h_0, h_1, h_2, \dots, h_{i-2}, h_{i-1}$ ) in parallel. The DDS 562 updates the coefficient values for each step of the parallel-to-serial converter 550, repeating the sequence of coefficient values every IFFT frame. In combination, these elements produce an interpolating filter that acts as baseband filter, anti-imaging filter and interpolator (for compensating for the difference between the QAM symbol rate and the channel spacing).
- [0053] The output of the FIR filter is effectively a multi-channel QAM modulated stream with proper channel spacing in an intermediate frequency (IF) band, interpolated and ready for up-conversion. The output is processed first by a quadrature corrector 558 to pre-correct for non-ideal behavior of a final-stage up-converter 512. An offset is added to the output of the quadrature corrector 558 via a digital adder 560 to pre-compensate for subsequent DC offsets. The offset-compensated result is applied to a D/A converter 510 for conversion to analog form. Note that the FIR filter output, quadrature output, and offset-compensated output are all complex quantities. The digital adder 560 is a "double adder" and the offset is a complex quantity. The D/A converter 510 in fact consists of two converters for separately converting the real and imaginary portions of its complex input to analog form. The complex output of the D/A converter 510 is applied to the final-stage up-converter 512 to frequency-shift the fully compensated and corrected IF multi-channel QAM-encoded stream up to a desired final frequency band to produce a multi-RF output for transmission.
- [0054] A complete Verilog HDL description of the digital portions of the multi-channel design is provided as an Appendix to this specification.

- [0055] Those of ordinary skill in the art will immediately understand that the preferred embodiment shown in Figure 5 represents a specific implementation tailored to currently available digital signal processing, D/A converter and up-converter technologies and that adaptations to that embodiment are readily made to accommodate alternative technologies. For example, given sufficient speed, all or a portion of the multi-channel QAM modulator of Figure 5 could be implemented on a digital signal processor or general purpose processor, substituting equivalent computer code for digital logic. Such a system could be specifically designed to execute the functions of the present inventive technique or could be implemented on a commercially available processor. In such a system, the code would be stored as computer instructions in computer readable media. Examples of computer-readable media include, but are not limited to: magnetic media such as hard disks, floppy disks, and magnetic tape; optical media such as CD-ROMs and holographic devices; magneto-optical media such as floptical disks; and hardware devices that are specially configured to store and execute program code, such as application-specific integrated circuits ("ASICs"), programmable logic devices ("PLDs") and ROM and RAM devices. Examples of computer code include machine code, such as produced by a compiler, and files containing higher-level code that are executed by a computer using an interpreter. For example, an embodiment of the invention may be implemented using Java, C or other object-oriented programming language and development tools. Another embodiment of the invention may be implemented in hardwired circuitry in place of, or in combination with, machine-executable software instructions.
- [0056] Although the invention has been shown and described with respect to a certain preferred embodiment or embodiments, certain equivalent alterations and modifications will occur to others skilled in the art upon the reading and understanding of this specification and the annexed drawings. In particular regard to the various functions performed by the above described components (assemblies, devices, circuits, etc.) the terms (including a reference to a "means") used to describe such components are intended to correspond, unless otherwise

indicated, to any component which performs the specified function of the described component (i.e., that is functionally equivalent), even though not structurally equivalent to the disclosed structure which performs the function in the herein illustrated exemplary embodiments of the invention. In addition, while a particular feature of the invention may have been disclosed with respect to only one of several embodiments, such feature may be combined with one or more features of the other embodiments as may be desired and advantageous for any given or particular application.

## APPENDIX

*modulator\_256qam.v*

### ***APPENDIX***

#### ***modulator\_256qam.v***

```

// FFR-based multichannel modulator with polyphase interpolation
// and programmable I/Q compensation.
// Copyright 2002 rgb media, inc.
// Peter Monta
//
module modulator_256qam(clk, reset,
    sync_16,
    en,
    //fixme: serial programming for filter
    p, pq,
    daci, daccq);
    input clk, reset;

```

```

input sync_16;
output en;
input [4:0] pi, pq;
output [13:0] daci, daccq;
//local
// dds to generate enable
// interleave to spread out correlated symbols across 6 MHz channels,
// to void problems from globally synchronous FEC framing
// zeropad to groups of 24
// iFFT
// interpolator
// final filter: x/sin(x), compensate for IF ceramic BPF, compensate
// for I/Q phase and amplitude and frequency response, compensate for DC
// offset
// truncate/round/dither?
endmodule

```

## mpeg\_checksum.v

## mpeg\_checksum.v

```

// Compute MPEG-TS checksum byte to replace 0x47 sync byte
// as per ITU-T J.83 section B.4. One byte per cycle.
// Copyright 2002 rgb media, inc.
// Peter Monta
//



module mpeg_checksum(clk, reset,
                     in, in_sync_16_188, in_req, in_ack,
                     out, out_sync_16, out_req, out_ack);
    input clk;
    input reset;
    input [7:0] in;
    input in_sync_16_188;
    input in_req;
    output in_ack;
    output [7:0] out;
    output out_sync_16;
    input out_req;
    output out_ack;
    reg out_sync_16;
    //local
    wire [7:0] r;
    wire [7:0] x0;
    wire en;
    wire parity;
    wire first;
    wire [7:0] rfir;
    wire [7:0] riir;
    fir_rfir#(8, r, rfit) fir_rfir;
    fir_riir#(in, riir) fir_riir;
    assign in_ack = out_req;
    assign out_ack = in_req;
    assign en = in_ack & in_req;
    delay #(15) sro(clk, reset, in_sync_16_188, en, parity, first);
    delay #(15) srl(clk, en, (parity ? 8'b0 : riir), r);
    flag_gen#(clk, reset, in_sync_16_188, en, parity, first);
    assign out = parity ? rfir : in;
    reg [3:0] c;

```

```

always @ (posedge clk)
    if (reset) begin
        c <= #1 0;
        out_sync_16 <= #1 0;
    end
    else begin
        if (en) begin
            if (in_sync_16_188 && (c==4'd15))
                $display($time,"**** mpeg_checksum: unexpected in_sync_16_188");
            c <= #1 in_sync_16_188 ? 0 : (c+1);
            out_sync_16 <= #1 (c==4'd14);
        end
    end
endmodule
// Feedback
//



module fir(r, q);
    input [7:0] r;
    output [7:0] q;
    assign q = {
        r[0]^r[1]^r[2]^r[3]^r[4]^r[5]^r[6]^r[7],
        r[3]^r[4]^r[5]^r[6],
        r[2]^r[5]^r[6],
        r[1]^r[4]^r[5]^r[7],
        r[0]^r[3]^r[4]^r[6]^r[7],
        r[0]^r[1]^r[5]^r[6],
        r[1]^r[2]^r[3]^r[4]^r[5],
        r[0]^r[1]^r[2]^r[3]^r[4]
    };
endmodule
// Feedforward
//



module fir(x, r, q);
    input [7:0] x;
    input [7:0] r;
    output [7:0] q;
    assign q = {
        r[7]^r[1]^b0,
        r[6]^r[7]^r[1]^b1,
        r[5]^r[6]^r[7]^r[1]^b1,
        r[4]^r[5]^r[6]^r[1]^b0,
        r[3]^r[4]^r[5]^r[6]^r[7]^b0,
        r[2]^r[3]^r[4]^r[5]^r[6]^r[7]^b1,
        r[1]^r[2]^r[3]^r[5]^r[6]^r[7]^b1,
        r[0]^r[1]^r[2]^r[3]^r[4]^r[5]^b1
    };
endmodule
// Generate parity and first-symbol signals
//



module flag_gen(clk, reset, sync, en, parity, first);
    input clk, reset, sync, en;
    output parity, first;
endmodule

```

## mpeg\_checksum.v

```

reg parity, first;
//local
reg [3:0] c;
parameter IDIS = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
reg [1:0] state;
always @ (posedge clk)
if (reset) begin
state <= #1 S1;
c <= #1 0;
parity <= #1 1;
first <= #1 0;
end else
if (en)
case (state)
IDLE: if (sync) begin parity <= #1 1; state <= #1 S1; end
S1: begin c <= #1 c + 1; if (c==4'd15) begin parity <= #1 0;
first <= #1 1; state <= #1 S2; end end
S2: begin c <= #1 c + 1; if (c==4'd15) begin first <= #1 0; state
<= #1 IDLE; end end
endcase
endmodule

```

pad\_16\_to\_24.v

**pad\_16\_to\_24.v**

```
// Make groups of 24 bauds out of incoming groups of 16, to interface
// with 24-point IFFT. Zero pad for guard channels.
// Copyright 2002 rgb media, inc.
// Peter Monta
//
```

pipe.v

**pipe.v**

```

module pipe_control_1(clk, reset,
in_req, in_ack,
out_req, out_ack,
en);

input clk, reset;
input in_req;
output in_ack;
output out_req;
input out_ack;
output [0:0] en;

//local
reg [0:0] v;

assign in_ack = !v[0] || out_ack;
assign en[0] = in_req & (!v[0] || out_ack);
assign out_req = v[0];

always @ (posedge clk)
if (reset) begin
v <= #1 0;
end else begin
v[0] <= #1 v[0] ? (en[0] ? (out_ack) : en[0]);
end
endmodule

```

ram.v

```

// Single-port RAM, sync write, async read
// 
module ram_sp(clk,addr,wdata,we,rdata);
parameter WDATA = 8;
parameter WADDR = 4;
input clk;
input [(WADDR-1):0] addr;
input [(WDATA-1):0] wdata;
input we;
output [(WDATA-1):0] rdata;
//local reg [(WDATA-1):0] mem[0:(1<<WADDR)-1];
always # (posedge clk) begin
  if (we)
    mem[addr] <= #1 wdata;
end
assign rdata = mem[addr];
endmodule

// Dual-port RAM, sync write, async read
// 
module ram(clk,waddr,wdata,we,raddr,rdata);
parameter WDATA = 8;
parameter WADDR = 4;
input clk;
input [(WADDR-1):0] waddr;
input [(WDATA-1):0] wdata;
input we;
input [(WADDR-1):0] raddr;
output [(WDATA-1):0] rdata;
//local reg [(WDATA-1):0] mem[0:(1<<WADDR)-1];
always # (posedge clk) begin
  if (we)
    mem[waddr] <= #1 wdata;
end
assign rdata = mem[raddr];

```

```

endmodule
// Synchronous dual-port memory with read port and write port.

module ram_rw(clk,waddr,wdata,we,raddr,rdata,en);
parameter WDATA = 8;
parameter WADDR = 11;
input clk;
input [(WADDR-1):0] waddr;
input [(WDATA-1):0] wdata;
input we;
input [(WADDR-1):0] raddr;
output [(WDATA-1):0] rdata;
input en;
//local reg [(WDATA-1):0] mem[0:(1<<WADDR)-1];
reg [(WADDR-1):0] qaddr;
always # (posedge clk) begin
  if (we)
    mem[waddr] <= #1 wdata;
  if (en)
    qaddr <= #1 raddr;
end
assign rdata = mem[qaddr];
endmodule

```

rmult.v

***rmult.v***

```

// Registered signed 14xN multiplier; product truncated to 18 bits. Latency
// two clocks.
//
module rmult(clk, a, b, p);
parameter W = 5;
input clk;
input [13:0] a;
input [(W-1):0] b;
output [17:0] p;

//local
reg [13:0] ra;
reg [(W-1):0] rb;
wire [35:0] rp;

assign p = rp[24:7];

// registered signed 18x18 Xilinx multiplier primitive
MULTI8X18S m(.P(rp), .A({(4{ra[13]}),ra}), .B({((18-W){rb[W-1]}),rb}),
.C(clk), .CE(1'b1), .R(1'b0));
always @ (posedge clk) begin
  ra <= #1 a;
  rb <= #1 b;
end
endmodule

```

from v

100

TOM.V

Appendix -

rom.v

from.v



TOMMY

Appendix -

from.v

Appendix -





rom.v



ROM.V

Appendix -

rom.v

rom.v

Appendix .



iron.v

ROM V



rom.v

Appendix -

from v



rom.v

from v



TOMMY

Tom. V

Appendix -

rom.v

from.v



rom.v

```

// synthesis attribute INTP_06 of r is
"ad000000aaaaaa000000aaaaaa000000aaaaaa000000aaaaaa"
// synthesis attribute INTP_07 of r is
"aa00000002aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
endmodule

```

Appendix -

fft24.v

**fft24.v**

```

// 24-point DIF pipeline inverse FFT. One complex sample per clock.
// Copyright 2002 rgb media, inc.
// Peter Monta
//



module ifft24(clk, reset, en, sync_24,
xi, xq, yi, yq);
input clk, reset;
input en;
input sync_24;
input [4:0] xi, xq;
output [11:0] yi, yq;

//local
wire [4:0] b00i, b00q, b01i, b01q, b02i, b02q;
wire [15:0] b0i, b0q, b02i, b02q;
wire [15:0] b10i, b10q, b11i, b11q, b1i, b1q;
wire [15:0] b20i, b20q, b21i, b21q, b2i, b2q;
wire [15:0] b30i, b30q, b31i, b31q, b3i, b3q;

reg [22:0] s;
wire [13:0] sync = {s, sync_24};
always @ (posedge clk)
if (en)
  s <= #1 {s[21:0], sync_24};

commutator_s1 #(10) c0(clk, reset, en, sync[0], {xi,xq}, {b00i,b00q},
{b01i,b01q}, {b02i,b02q});
butterfly_3#(b0(clk, reset, en, sync[1], b00i, b00q, b01i, b01q, b02i, b02q,
b0i, b0q);
rotate_24 r24(clk, reset, en, sync[5], b0i, b0q, b1i, b1q, b2i, b2q);
commutator_s2 #(32) c1(clk, reset, en, sync[16], {b0i,b0q}, {b10i,b10q},
{b11i,b11q});
butterfly_2 #(16) b1(clk, reset, en, sync[17], b10i, b10q, b11i, b11q, b1i,
b1q);
rotate_8 r8(clk, reset, en, sync[18], b1i, b1q, b1i, b1q);
commutator_s3 #(32) c2(clk, reset, en, sync[22], {b11i,b11q}, {b20i,b20q},
{b21i,b21q});
butterfly_2_ns #(16) b2(clk, reset, en, sync[7], b20i, b20q, b21i, b21q,
b2i, b2q);
rotate_4 r4(clk, reset, en, sync[8], b2i, b2q, b21i, b21q);
commutator_s4 #(32) c3(clk, reset, en, sync[9], {b21i,b21q}, {b30i,b30q},
{b31i,b31q});
butterfly_2_ns #(16) b3(clk, reset, en, sync[14], b30i, b30q, b31i, b31q,
b3i, b3q);

```

```

wire [15:0] yi, yyq;
bitrev24 #(32) b24(clk, reset, en, sync[15], {b3i,b3q}, {yyi,yyq});
delay16 #(6) add0(clk, en, yyi, yi);
delay16 #(6) add1(clk, en, yyq, yq);

endmodule
// 3-point serial butterfly
//


module butterfly_3(clk, reset, en, sync,
x0i, x0q, x1i, x1q, x2i, x2q,
yi, yq);
input clk, reset;
input en, sync;
input [4:0] x0i, x0q, x1i, x1q, x2i, x2q;
output [15:0] yi, yq;
//local
reg [11:0] p;
wire cal, cb3, cc2;
assign ca1 = {p==2'd1};
assign cb3 = {p==2'd1};
assign cc2 = {p==2'd2};
always @ (posedge clk)
if (reset)
  p <= #1 0;
else begin
  if (en)
    p <= #1 (sync || (p==2'd2)) ? 0 : p+1;
end
b3sum #(1) b0(clk, reset, en,
x0i, x1i, x2i, x1q, x2q,
cal, cb3, cc2,
yi);
b3sum #(1) b1(clk, reset, en,
x0q, x1q, x2q, x2i, x1i,
cal, cb3, cc2,
yq);
endmodule
// half of a 3-point butterfly datapath
//


module b3sum(clk, reset, en,
a, b, c, d, e,
cal, cb3, cc2,
out);

```

## fft24.v

```

parameter R = 0;
parameter clk, reset, en;

input clk, reset, en;
input [4:0] a, b, c, d, e;
input ca1, cb3, cc2;
output [15:0] out;
reg [15:0] out;

//local
reg [4:0] t1, t2;
reg [5:0] t3, t5;
reg [6:0] t4;
wire [14:0] t6;
reg [14:0] t7;
reg [7:0] t8;
sqrt3 s3(clk, en, t5, t6);

always @ (posedge clk)
if (en) begin
t1 <= #1 t1;
t2 <= #1 t1;
t3 <= #1 {b[4],b} + {c[4],c};
t4 <= #1 ca1 ? {t3,1'b0} : {~{t3[5],t3} + 1};
t5 <= #1 {d[4],d} - {e[4],e};
t6 <= #1 cc2 ? 0 : t6;
t8 <= #1 {(2*(t2[4])),t2,1'b0} + {t4[6],t4};
out <= #1 cb3 ? {t8,8'd0} - {t7[14],t7} : {t8,8'd0} + {t7[14],t7};
end
endmodule

// Multiply by sqrt(3). 6-bit input (2's comp Q6.0), 15-bit output (Q7.8).
//
```

```

module sqrt3(clk, en, x, y);
input clk, en;
input [5:0] x;
output [14:0] y;
reg [14:0] y;

//local
reg [14:0] q;
always @ (x)
  case (x)
    6'd0: q = 15'd0;
    6'd1: q = 15'd443;
    6'd2: q = 15'd887;
    6'd3: q = 15'd1330;
    6'd4: q = 15'd1774;
    6'd5: q = 15'd2217;
    6'd6: q = 15'd2660;
    6'd7: q = 15'd3104;
    6'd8: q = 15'd3547;
    6'd9: q = 15'd3991;
    6'd10: q = 15'd4344;
    6'd11: q = 15'd4877;
    6'd12: q = 15'd5321;
    6'd13: q = 15'd5764;
    6'd14: q = 15'd6208;
    6'd15: q = 15'd6651;
    6'd16: q = 15'd7094;
    6'd17: q = 15'd7538;
    6'd18: q = 15'd981;
    6'd19: q = 15'd8425;
    6'd20: q = 15'd8668;
    6'd21: q = 15'd912;
    6'd22: q = 15'd9755;
    6'd23: q = 15'd10198;
    6'd24: q = 15'd10642;
    6'd25: q = 15'd11085;
    6'd26: q = 15'd11529;
    6'd27: q = 15'd11972;
    6'd28: q = 15'd12415;
    6'd29: q = 15'd12859;
    6'd30: q = 15'd13302;
    6'd31: q = 15'd13746;
    6'd32: q = 15'd14879;
    6'd33: q = 15'd15022;
    6'd34: q = 15'd19466;
    6'd35: q = 15'd19097;
    6'd36: q = 15'd23535;
    6'd37: q = 15'd2796;
    6'd38: q = 15'd22397;
    6'd39: q = 15'd21683;
    6'd40: q = 15'd22126;
    6'd41: q = 15'd25707;
    6'd42: q = 15'd2013;
    6'd43: q = 15'd2456;
    6'd44: q = 15'd2900;
    6'd45: q = 15'd23343;
    6'd46: q = 15'd24787;
    6'd47: q = 15'd2230;
    6'd48: q = 15'd22674;
    6'd49: q = 15'd21117;
    6'd50: q = 15'd25560;
    6'd51: q = 15'd27004;
    6'd52: q = 15'd27447;
    6'd53: q = 15'd2891;
    6'd54: q = 15'd23334;
    6'd55: q = 15'd2877;
    6'd56: q = 15'd23221;
    6'd57: q = 15'd23664;
    6'd58: q = 15'd30108;
    6'd59: q = 15'd30551;
    6'd60: q = 15'd30994;
    6'd61: q = 15'd34438;
    6'd62: q = 15'd34881;
    6'd63: q = 15'd3325;
```

## fft24.v

```

endcase
always @ (posedge clk)
if (en) y <= #1 q;
endmodule
// 2-point serial butterfly
module butterfly_2(clk, reset, en, sync,
xi, x0q, x1i, x1q,
yi, yq);
parameter W = 16;
input clk, reset;
input en, sync;
input [(W-1):0] xi, x0q, x1i, x1q;
output [(W-1):0] yi, yq;
reg [(W-1):0] yi, yq;
//local
reg phase;
reg lsb1, lsbq;
always @ (posedge clk)
if (reset)
phase <= #1 0;
else
if (en) begin
Phase <= #1 sync ? 0 : ~Phase;
{y, lsb1} <= #1 phase ? {x0i[(W-1)], x0i} - {x1i[(W-1)], x1i} :
{x0i[(W-1)], x0i} + {x1i[(W-1)], x1i};
{yq, lsbq} <= #1 phase ? {x0q[(W-1)], x0q} - {x1q[(W-1)], x1q} :
{x0q[(W-1)], x0q} + {x1q[(W-1)], x1q};
end
endmodule
// 2-point serial butterfly, no scaling
//
```

```

output [(W-1):0] yi, yq;
reg [(W-1):0] yi, yq;
//local
reg phase;
always @ (posedge clk)
if (reset)
phase <= #1 0;
else
if (en) begin
Phase <= #1 sync ? 0 : ~Phase;
yi <= #1 phase ? x0i - x1i : x0i + x1i;
yq <= #1 phase ? x0q - x1q : x0q + x1q;
end
endmodule
// Rotate by multiple of 2*pi/24, bias -6*pi/24.
//
```

```

module rotate_24(clk, reset, en, sync,
xi, xq,
yi, yq);
input clk, reset;
input en, sync;
input [15:0] xi, xq;
output [15:0] yi, yq;
//local
wire [14:0] c;
wire cy;
counter #(5,24) cs(clk, reset)(sync6ten), en, c, cy);
reg [4:0] cr;
wire cr24, cr12, cr8, cr2, cr1;
assign {cr24, cr12, cr8, cr2, cr1} = cr;
reg [4:0] ncr;
always @ (c)
case (c)
5'b23: ncr = 5'b00000;
5'b0: ncr = 5'b00000;
5'b1: ncr = 5'b00000;
5'b2: ncr = 5'b00000;
5'b3: ncr = 5'b00000;
5'b4: ncr = 5'b00000;
5'b5: ncr = 5'b00000;
5'b6: ncr = 5'b00000;
5'b7: ncr = 5'b10100;
5'b8: ncr = 5'b00000;
5'b9: ncr = 5'b11000;
5'b10: ncr = 5'b00000;
5'b11: ncr = 5'b00000;
```

## fft24.v

```

5'd12: ncr = 5'b1010;
5'd13: ncr = 5'b01010;
5'd14: ncr = 5'b00000;
5'd15: ncr = 5'b01100;
5'd16: ncr = 5'b10110;
5'd17: ncr = 5'b00000;
5'd18: ncr = 5'b00010;
5'd19: ncr = 5'b00001;
5'd20: ncr = 5'b00000;
5'd21: ncr = 5'b10101;
5'd22: ncr = 5'b01001;
default: ncr = 5'bxx;
endcase
always @ (posedge clk)
if (reset)
cr <= #1 5'b0000;
else begin
if (en)
cr <= #1 ncr;
end
wire cr12_d3, cr1_d6, cr2_d9, cr1_d10;
delay1 # (2) d0 (clk, en, cr12, cr12_d3);
delay1 # (5) d1 (clk, en, cr8, cr8_d6);
delay1 # (8) d2 (clk, en, cr2, cr2_d9);
delay1 # (9) d3 (clk, en, cr1, cr1_d10);

wire [15:0] t0i, t0q, t1i, t1q, t2i, t2q, t3i, t3q;
r24 zr0(clk, en, cr24, xi, xq, t0i, t0q);
r12 zr1(clk, en, cr12_d3, t0i, t0q, t1i, t1q);
r8 zr2(clk, en, cr8_d6, t1i, t1q, t2i, t2q);
r2 zr3(clk, en, cr2_d9, t2i, t2q, t3i, t3q);
r1 zr4(clk, en, cr1_d10, t3i, t3q, yi, yq);
endmodule
// Rotate by multiple of 2*pi/8, bias .
// Rotate by multiple of 2*pi/8, bias .
module rotate_8(clk, reset, en, sync,
xi, xq,
yi, yq);
input clk, reset;
input en, sync;
input [15:0] xi, xq;
output [15:0] yi, yq;
//local
wire [1:0] c;
wire cy;
counter #(2,4) cs(clk, reset || (sync&en), en, c, cy);
reg cr;

```

```

reg [1:0] cr;
wire cr8, cr4;
assign {cr8, cr4} = cr;
reg [1:0] ncr;
always @ (c)
case (c)
3'd7: ncr = 2'b00;
3'd0: ncr = 2'b00;
3'd1: ncr = 2'b00;
3'd2: ncr = 2'b10;
3'd3: ncr = 2'b00;
3'd4: ncr = 2'b01;
3'd5: ncr = 2'b00;
3'd6: ncr = 2'b11;
default: ncr = 2'bxx;
endcase
always @ (posedge clk)
if (reset)
cr <= #1 2'b00;
else begin
if (en)
cr <= #1 ncr;
end
wire cr4_d3,
delay1 # (2) d0 (clk, en, cr4, cr4_d3);
wire [15:0] t0i, t0q;
r8 z0 (clk, en, cr8, xi, xq, t0i, t0q);
r4 z1 (clk, en, cr4_d3, t0i, t0q, yi, yq);
endmodule
// Rotate by multiple of 2*pi/4.
// Rotate by multiple of 2*pi/4.
module rotate_4(clk, reset, en, sync,
xi, xq,
yi, yq);
input clk, reset;
input en, sync;
input [15:0] xi, xq;
output [15:0] yi, yq;
//local
wire [1:0] c;

```

## fft24.v

```

wire cr2;
assign cr2 = cr;
reg ncr;

always @ (c)
  case (c)
    2'd3: ncr = 1'b0;
    2'd0: ncr = 1'b0;
    2'd1: ncr = 1'b0;
    2'd2: ncr = 1'b1;
    default: ncr = 2'bxx;
  endcase

always @ (posedge clk)
  if (reset)
    cr <= #1'b0;
  else begin
    if (en)
      cr <= #1 ncr;
  end
  r2 z0(clk, en, cr2, xi, xq, yi, yq);
endmodule

// stage 1 commutator, to feed 3-pt butterfly
// module commutator_s1(clk, reset, en, sync,
//                      x, y0, y1, y2);
// parameter W = 32;
// input clk, reset;
// input en;
// input sync;
// input [(W-1):0] x;
// output [(W-1):0] y0, y1, y2;
// reg [(W-1):0] y0, y1, y2;
// /local
//   wire [(W-1):0] r0, r1, r2;
//   reg bank;
//   wire [2:0] ic;
//   wire [1:0] ie;
//   wire [2:0] oe;
//   wire [1:0] oc;
//   wire icy, ocy;
//   bi_counter #(2,3,3,8) bc0(clk, reset||sync&en, en, {ic,ie}, icy);
//   bi_counter #(3,8,2,3) bc1(clk, reset||sync&en, en, {oe,oc}, ocy);
// always @ (posedge clk)
//   if (reset)

module commutator_s1(clk, reset, en, sync,
                     x, y0, y1);
parameter W = 32;
input clk, reset;
input en;
input sync;
input [(W-1):0] x;
output [(W-1):0] y0, y1;
reg [(W-1):0] y0, y1;
//local
//  wire [(W-1):0] r0, r1, r2;
//  reg bank;
//  wire [2:0] ic;
//  wire [1:0] ie;
//  wire [2:0] oe;
//  wire [1:0] oc;
//  wire icy, ocy;
//  bi_counter #(2,3,3,8) bc0(clk, reset||sync&en, en, {ic,ie}, icy);
//  bi_counter #(3,8,2,3) bc1(clk, reset||sync&en, en, {oe,oc}, ocy);
// always @ (posedge clk)
//   if (reset)

```

fft24.v

```

if (reset)
  bank <= #1 0;
else begin
  if (en && icy)
    bank <= #1 !bank;
end

// stage 3 commutator
// module commutator_s3(clk, reset, en, sync,
// x, y0, y1);
parameter W = 32;
input clk, reset;
input en;
input sync;
input [(W-1):0] x;
input [(W-1):0] y0, y1;
output [(W-1):0] y0, y1;
reg [(W-1):0] y0, y1;
req [(W-1):0] y0, y1;
//local
wire [(W-1):0] r0, r1;
reg bank;
wire [1:0] ic;
wire ie;
wire [1:0] oc;
wire oe;
wire icy, ocy;
bi_counter #(2,4,1,2) bc0(clk, reset||(sync&en), en, (bank,ic), icy);
bi_counter #(2,4,1,2) bc1(clk, reset||(sync&en), en, (oc,oe), ocy);
always @ (posedge clk)
  if (reset)
    bank <= #1 0;

```

```

else begin
  if (en && icy)
    bank <= #1 !bank;
end

wire en0 = en && (ic[1]==0);
wire en1 = en && (ic[1]==1);
wire [2:0] waddr = {bank,ie,ic[0]};
wire [2:0] raddr = {!bank,oc};
ram #(W,3) ra0(clk, waddr, x, en0, raddr, r0);
ram #(W,3) ra1(clk, waddr, x, en1, raddr, r1);

always @ (posedge clk)
  if (en) begin
    y0 <= #1 r0;
    y1 <= #1 r1;
  end
endmodule

// stage 4 commutator
// module commutator_s4(clk, reset, en, sync,
// x, y0, y1);
parameter W = 32;
input clk, reset;
input en;
input sync;
input [(W-1):0] x;
output [(W-1):0] y0, y1;
reg [(W-1):0] y0, y1;
//local
wire [(W-1):0] r0, r1;
wire bank;
wire [1:0] ic;
wire oe;
wire icy, ocy;
bi_counter #(1,2,1,2) bc1(clk, reset||(sync&en), en, (oc,oe), ocy);

wire en0 = en && (ic[1]==0);
wire en1 = en && (ic[1]==1);
wire [1:0] waddr = {bank,ic[0]};
wire [1:0] raddr = {!bank,oc};

```

## fft24.v

```

ram #(W,2) r#0(clk, waddr, x, en0, raddr, r0);
ram #(W,2) r#1(clk, waddr, x, en1, raddr, r1);

always @ (posedge clk)
  if (en) begin
    y0 <= #1 r0;
    y1 <= #1 r1;
  end
endmodule

// Rotate by plus or minus pi/24.
// Uses rational approximation 47/357 of tan(pi/24), accurate to about 20
bits.
// In binary, 101111 / 101100101.
// In CSD, 11000x / 10110001 where x is the -1 digit.
// Latency three cycles.

module r24(clk, en, sign, xi, xq, yi, yq);
  input clk;
  input en, sign;
  input [15:0] xi, xq;
  output [15:0] yi, yq;
endmodule

ip_357_47 ip0(clk, en, sign, xi, xq, yi);
ip_357_47 ip1(clk, en, sign, xi, xq, yi);
// c = sign ? (357/512)*a - (47/512)*b : (357/512)*a + (47/512)*b. Latency
three cycles.

module ip_357_47 (clk, en, sign, a, b, c);
  input clk;
  input en, sign;
  input [15:0] a,b;
  output [15:0] c;
endmodule

reg [15:0] c;

//local
reg [16:0] t1, t3, t4, t5, t6;
reg lsb;
wire sign2;
delay1 #(1) d(clk, en, sign, sign2);

always @ (posedge clk)
  if (en) begin
    t1 <= #1 (a[15],a) + ((2(a[15]),a)[15:1]);
    t3 <= #1 (b[15],b) + ((2(b[15]),b)[15:1]);
    t4 <= #1 (b[15],b) + ((5(b[15]),b)[15:4]);
    t5 <= #1 t1 + ((6(t1[16]),t1)[16:6]);
    t6 <= #1 t3 + ((3(t4[16]),t4)[16:3]);
    (c,lsb) <= #1 sign2 ? t5 - ((2(t6[16]),t6)[16:2]) : t5 +
    ((2(t6[16]),t6[16:2]));
  end
endmodule

always @ (posedge clk)
  if (en) begin
    t1 <= #1 (a[15],a) + ((3(a[15]),a)[15:2]);
    t3 <= #1 sign ? (a[15],a)-((b[15],b)) : (a[15],a)+(b[15],b);
    t4 <= #1 (b[15],b) - ((5(b[15]),b)[15:4]);
    t5 <= #1 t1 + ((6(t1[16]),t1)[16:6]);
    t6 <= #1 signl ? ((3(t3[16]),t3[16:3])) - (((4(t4[16]),t4)[16:4]) :
    //t3>>3 + t4>>8;
    ((3(t3[16]),t3[16:3])) + (((4(t4[16]),t4)[16:4]));
  end
endmodule

```

```

{c,lsb} <= #1 t5 + t6;
endmodule

// Rotate by plus or minus pi/12.
// Uses rational approximation 209/780 of tan(pi/12), accurate to about 20
bits.
// In binary, 11010001 / 1100001100
// Latency three cycles.

module r12(clk, en, sign, xi, xq, yi, yq);
  input clk;
  input en, sign;
  input [15:0] xi, xq;
  output [15:0] yi, yq;
endmodule

ip_780_209 ip0(clk, en, sign, xi, xq, yi);
ip_780_209 ip1(clk, en, sign, xi, xq, yi);
// c = sign ? (780/1024)*a - (209/1024)*b : (780/1024)*a + (209/1024)*b.

always @ (posedge clk)
  if (en) begin
    reg [16:0] t1, t3, t4, t5, t6;
    reg lsb;
    wire sign2;
    delay1 #(1) d(clk, en, sign, sign2);

    always @ (posedge clk)
      if (en) begin
        t1 <= #1 (a[15],a) + ((2(a[15]),a)[15:1]);
        t3 <= #1 (b[15],b) + ((2(b[15]),b)[15:1]);
        t4 <= #1 (b[15],b) + ((5(b[15]),b)[15:4]);
        t5 <= #1 t1 + ((6(t1[16]),t1)[16:6]);
        t6 <= #1 t3 + ((3(t4[16]),t4)[16:3]);
        (c,lsb) <= #1 sign2 ? t5 - ((2(t6[16]),t6)[16:2]) : t5 +
        ((2(t6[16]),t6[16:2]));
      end
    endmodule
  end
endmodule

```

## fft24.v

```

// module r8(clk, en, sign, xi, xq, yi, yq);
//   input clk;
//   input en, sign;
//   input [15:0] xi, xq;
//   output [15:0] yi, yq;
// ip_1393_577 ip0(clk, en, sign, xi, xq, yi);
// ip_1393_577 ip1(clk, en, !sign, xq, xi, yq);
endmodule

// c = sign ? 1393*a - 577*b : 1393*a + 577*b. Latency three cycles.
module ip_1393_577(clk, en, sign, a, b, c);
  input clk;
  input en, sign;
  input [15:0] a,b;
  output [15:0] c;
  reg [15:0] t1, t2, t3, t4, t5, t6;
  reg lsb;
  reg sign;
  always @ (posedge clk)
    if (en)
      sign <= #1 sign;
  endmodule

// local
reg [15:0] t1, t2, t3, t4, t5, t6;
reg sign;
always @ (posedge clk)
  if (en)
    sign <= #1 sign;
  always @ (posedge clk)
    if (en) begin
      t1 <= #1 (a[15],a) + ((3*a[15]),a[15:2]);
      t2 <= #1 (a[15],a) - ((4*a[15]),a[15:3]);
      t3 <= #1 (b[15],b) + ((4*b[15]),b[15:3]);
      t4 <= #1 sign ? (a[15],a) - (b[15],b) : (a[15],a) + (b[15],b);
      t5 <= #1 t1 + (3*t2[16]),t2[16:3];
      t6 <= #1 sign1 ? ((9*t4[16]),t4[16:9]) - t3 : ((9*t4[16]),t4[16:9]) +
      t3;
      {c,lsb} <= #1 t5 + {t6[16],t6[16:1]};
    end
  endmodule

// Rotate by plus or minus pi/4.
module r4(clk, en, sign, xi, xq, yi, yq);
  input clk;
  input en, sign;
  input [15:0] xi, xq;
  output [15:0] yi, yq;
ip_1_1 ip0(clk, en, sign, xi, xq, yi);
ip_1_1 ip1(clk, en, !sign, xq, xi, yq);
endmodule

```

## fft24.v

```

/*
// Bit-reverse to give IFFT time samples in conventional
// order. ("Bit-reverse" is a slight misnomer, given the radix-3
// factor in 24, but the principle is similar to pure radix-2^n.)
//


module bitrev24(clk, reset, en, sync, x, y);
parameter W = 32;
input clk, reset;
input en;
input sync;
input [W-1:0] x;
output [W-1:0] y;

reg [(W-1):0] y;

//local
reg bank;
wire [4:0] vc;
wire icy;
wire rc;
wire oxy;

bi_counter #(12,3,3,8) c0(clk, reset[(sync&en)], en, wc, icy);
bi_counter #(3,6,2,3) c1(clk, reset[(sync&en)], en, rc, oxy);

wire [5:0] waddr = {bank,wc[0],wc[1],wc[2],wc[4:31]};
wire [5:0] raddr = {bank,rc};

always @ (posedge clk)
if (reset)
    bank <= #1 0;
else begin
    if (en && icy)
        bank <= #1 !bank;
end

wire [(W-1):0] ty;
ram #(W,6) r(clk, waddr, x, 1'b1, raddr, ty);

always @ (posedge clk)
if (en)
    y <= #1 ty;

endmodule

```

**interpolate\_256qam.v****interpolate\_256qam.v**

```

// Polyphase interpolation, SRRC pulse shaping for 256-QAM.
// Copyright 2002 rgb media, inc.
// Peter Monta
//
module interpolate_256qam(clk, reset, en, xi, xq, yi, yq);
  input clk, reset;
  output en;
  input [13:0] xi, xq;
  input [13:0] yi, yq;
  // DDS
  wire [19:0] phase_20;
  wire [4:0] addr;
  wire [9:0] phase = phase_20[19:10];
  dds ddc(clk, reset, phase_20, en, addr);
  // SRRC filter coefficients
  wire [4:0] h0, h21;
  wire [5:0] h1, h20;
  wire [6:0] h3, h18, h16, h17;
  wire [7:0] h4, h5, h16, h14, h15;
  wire [8:0] h6, h7, h14, h13;
  wire [9:0] h8, h10, h12;
  coeffs_256gam coeffs, phase,
  interpolate_channel_256gam ireal(clk, reset, en, addr,
  h0, h1, h2, h3, h4, h5, h6, h7, h8, h9, h10,
  h11, h12, h13, h14, h15, h16, h17, h18, h19, h20, h21,
  xi, yi);
  // real and imaginary FIR filters
  interpolate_channel_256gam imag(clk, reset, en, addr,
  h0, h1, h2, h3, h4, h5, h6, h7, h8, h9, h10,
  h11, h12, h13, h14, h15, h16, h17, h18, h19, h20, h21,
  xq, yq);
endmodule
// Interpolator FIR filter
//
module interpolate_channel_256qam(clk, reset, en, addr,

```

```

h0, h1, h2, h3, h4, h5, h6, h7, h8, h9, h10,
h11, h12, h13, h14, h15, h16, h17, h18, h19, h20, h21,
x, y;
input clk, reset;
input en;
input [4:0] addr;
input [4:0] h0, h21;
input [5:0] h1, h20;
input [6:0] h2, h3, h18, h19;
input [7:0] h4, h5, h16, h17;
input [8:0] h6, h7, h14, h15;
input [9:0] h8, h13;
input [11:0] h9, h10, h11, h12;
input [13:0] x;
output [13:0] y;
reg [13:0] y;
//local
wire [13:0] x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,
x11, x12, x13, x14, x15, x16, x17, x18, x19, x20, x21;
// delay taps
delay_tap tap0(clk, en, addr, x, x0);
delay_tap tap1(clk, en, addr, x0, x1);
delay_tap tap2(clk, en, addr, x1, x2);
delay_tap tap3(clk, en, addr, x2, x3);
delay_tap tap4(clk, en, addr, x3, x4);
delay_tap tap5(clk, en, addr, x4, x5);
delay_tap tap6(clk, en, addr, x5, x6);
delay_tap tap7(clk, en, addr, x6, x7);
delay_tap tap8(clk, en, addr, x7, x8);
delay_tap tap9(clk, en, addr, x8, x9);
delay_tap tap10(clk, en, addr, x9, x10);
delay_tap tap11(clk, en, addr, x10, x11);
delay_tap tap12(clk, en, addr, x11, x12);
delay_tap tap13(clk, en, addr, x12, x13);
delay_tap tap14(clk, en, addr, x13, x14);
delay_tap tap15(clk, en, addr, x14, x15);
delay_tap tap16(clk, en, addr, x15, x16);
delay_tap tap17(clk, en, addr, x16, x17);
delay_tap tap18(clk, en, addr, x17, x18);
delay_tap tap19(clk, en, addr, x18, x19);
delay_tap tap20(clk, en, addr, x19, x20);
delay_tap tap21(clk, en, addr, x20, x21);
// multipliers
wire [17:0] p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10,
p11, p12, p13, p14, p15, p16, p17, p18, p19, p20, p21;
rmult #(5) rm0(clk, x0, h0, p0);
rmult #(6) rm1(clk, x1, h1, p1);
rmult #(7) rm2(clk, x2, h2, p2);

```

## interpolate\_256qam.v

```

// adder tree
always @ (posedge clk) begin
    a0 <= #1 p0 + p1;
    a1 <= #1 p2 + p3;
    a2 <= #1 p4 + p5;
    a3 <= #1 p6 + p7;
    a4 <= #1 p8 + p9;
    a5 <= #1 p10 + p11;
    a6 <= #1 p12 + p13;
    a7 <= #1 p14 + p15;
    a8 <= #1 p16 + p17;
    a9 <= #1 p18 + p19;
    a10 <= #1 p20 + p21;

    b0 <= #1 a0 + a1;
    b1 <= #1 a2 + a3;
    b2 <= #1 a4 + a5;
    b3 <= #1 a6 + a7;
    b4 <= #1 a8 + a9;
    b5 <= #1 a10;

    c0 <= #1 b0 + b1;
    c1 <= #1 b2 + b3;
    c2 <= #1 b4 + b5;
    d0 <= #1 c0 + c1;
    d1 <= #1 c2 + 18'd7;
    {yy, lsb5} <= #1 d0 + d1;
end

```

```

// adder tree
always @ (posedge clk) begin
    a0 <= #1 (lsbs<<4'd8) ? yy : (yy+1);
    end
endmodule

// FIR interpolator
// Delay of 24 samples, one tap for the FIR interpolator.
// Recirculates the data when enable is disasserted.
module delay_tap(clk, en, addr, in, out);
    input clk, en;
    input [4:0] addr;
    input [13:0] in;
    output [13:0] out;
    ram_sp #(14,5) r(clk, addr, in, en, out);
endmodule

// DDS (direct digital synthesizer) to generate baud timing
// (fractional phase and enable for next symbol).
// Local
reg [19:0] p0, np0, nincr0, nincr1, p1;
reg [19:0] step;
reg [19:0] step1;
reg [4:0] addr;
reg [4:0] c, cl;
reg cy;
reg en1, en2, en3;
dds_step_from_dsr(clk, c, step);

always @ (posedge clk)
    if (reset) begin
        addr <= #1 0;
        p0 <= #1 0;
        en1 <= #1 0;
        c <= #1 0;
        cl <= #1 0;
        cy <= #1 0;
    end
    else begin
        cy <= #1 (c==5'd22);
        c <= #1 cy ? 0 : c+1;
    end
end

```

## interpolate\_256qam.v

```

    n0 <= #1 ! (p0 < 20'd111754);
    ninc0 <= #1 p0 + 20'd468111;
    nincrl <= #1 p0 - 20'd55877;
    np0 <= #1 n0 ? nincrl : ninc0;
    en1 <= #1 cy ? np0 : p0;
    end
    step1 <= #1 step;
    p1 <= #1 p0;
    phase <= #1 p1 + step1;
    en2 <= #1 en1;
    en <= #1 en2;
    c1 <= #1 c;
    addr <= #1 c1;
    end
    // DDS polyphase steps (within a baud) for channels Q--Z.
    // local
    // DDS step romm(clk, addr, step);
    module dds_step_romm(clk, addr, step);
        input clk;
        input [4:0] addr;
        output [19:0] step;
        reg [19:0] step;
        always @ (addr)
            case (addr)
                5'd0:   step = 20'd0;
                5'd1:   step = 20'd19517;
                5'd2:   step = 20'd39054;
                5'd3:   step = 20'd558551;
                5'd4:   step = 20'd78069;
                5'd5:   step = 20'd97586;
                5'd6:   step = 20'd117103;
                5'd7:   step = 20'd136620;
                5'd8:   step = 20'd156137;
                5'd9:   step = 20'd175654;
                5'd10:  step = 20'd195171;
                5'd11:  step = 20'd214688;
                5'd12:  step = 20'd234206;
                5'd13:  step = 20'd253123;
                5'd14:  step = 20'd273240;
                5'd15:  step = 20'd292757;
                5'd16:  step = 20'd312274;
                5'd17:  step = 20'd3331791;
                5'd18:  step = 20'd353308;
                5'd19:  step = 20'd370825;
                5'd20:  step = 20'd390343;
                5'd21:  step = 20'd409860;
                5'd22:  step = 20'd429377;
                5'd23:  step = 20'd448894;
                default: step = 20'bsy;
    endmodule

```

```

endcase
endmodule
// Generate impulse-response coefficients by table lookp.
// Saves block ROMs by using smaller words for the filter tails.
//
module coeffs_256qam(clk, phase,
                      h0, h1, h2, h3, h4, h5, h6, h7, h8, h9, h10,
                      h11, h12, h13, h14, h15, h16, h17, h18, h19, h20, h21);
    input clk;
    input [9:0] phase;
    output [4:0] h0, h21;
    output [5:0] h1, h20;
    output [6:0] h, h3, h18, h19;
    output [7:0] h4, h5, h16, h17;
    output [8:0] h6, h7, h14, h15;
    output [9:0] h8, h13;
    output [11:0] h9, h0, h11, h12;
    //local
    wire [9:0] phase_d16, phase_d23;
    delay10 #(15) dn(clk, 1'b1, phase_d16);
    delay10 #(6) d1(clk, 1'b1, phase_d16, Phase_e_d23);

    wire [185:0] r;
    rom_256qam rom(clk, phase_d23, r);
    wire [4:0] h0 = r[185:181];
    wire [5:0] h1 = r[180:175];
    wire [6:0] h2 = r[174:168];
    wire [6:0] h3 = r[167:161];
    wire [7:0] h4 = r[160:153];
    wire [7:0] h5 = r[152:145];
    wire [8:0] h6 = r[144:136];
    wire [8:0] h7 = r[136:127];
    wire [9:0] h8 = r[126:117];
    wire [11:0] h9 = r[116:105];
    wire [11:0] h10 = r[104:93];
    wire [11:0] h11 = r[92:81];
    wire [11:0] h12 = r[80:69];
    wire [19:0] h13 = r[68:59];
    wire [18:0] h14 = r[55:50];
    wire [18:0] h15 = r[49:41];
    wire [17:0] h16 = r[40:33];
    wire [17:0] h17 = r[32:25];
    wire [16:0] h18 = r[22:18];
    wire [6:0] h19 = r[17:11];
    wire [5:0] h20 = r[10:5];
    wire [4:0] h21 = r[4:0];
endmodule

```

## interpolate\_256qam.v

```

module rom_256qam(clk, phase, x);
  input clk;
  input [9:0] phase;
  output [165:0] x;

  wire [17:0] w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10;
  assign x = {w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10[17:12]};

  rom_256_0 r0(clk, phase, w0);
  rom_256_1 r1(clk, phase, w1);
  rom_256_2 r2(clk, phase, w2);
  rom_256_3 r3(clk, phase, w3);
  rom_256_4 r4(clk, phase, w4);
  rom_256_5 r5(clk, phase, w5);
  rom_256_6 r6(clk, phase, w6);
  rom_256_7 r7(clk, phase, w7);
  rom_256_8 r8(clk, phase, w8);
  rom_256_9 r9(clk, phase, w9);
  rom_256_10 r10(clk, phase, w10);

endmodule

```

e16.v

165

```

// Swallows the first sixteen data items, then passes
// data unmmodified. Used only to assist in matching
// the GI test vectors; may be removed in final synthesis.
// The following code is generated by the QM2 tool

module a16(clk, reset,
           in, in_sync, in_req, in_ack,
           out, out_sync, out_req, out_ack);
  input clk;
  input reset;
  input [7:0] in;
  input in_sync;
  input in_req;
  output in_ack;
  output [7:0] out;
  output out_sync;
  input out_req;
  output out_ack;

  //Local
  reg [4:0] c;

  assign out = in;
  assign out_sync = in_sync;
  wire gate = (c==5'd16);
  assign in_ack = out_req && gate;
  assign out_ack = in_req && gate;

  always @ (posedge clk)
    begin
      if (~reset)
        c <= #1 0;
      else
        begin
          if (in_ack && in_req)
            c <= #1 (c<5'd1) ? (c+1) : c;
        end
    end
endmodule

```

## fec.v

## fec.v

```

`define async_reset
`define QAM56
`define N_CHANNELS 16 // number of channels - MUST be greater than 3
`define N_CHANNELS 4 // number of bits needed for channel number

module randomize(clk,reset_l,en_init,sync_in,sync_out,din,dout);
  input clk;
  input reset_l;
  input en;
  input init;
  input sync_in;
  output sync_out;
  input [13:0] din;
  output [13:0] dout; // data out (channel n/n+1 pair - 7 bits each)
  output [13:0] dout; // data out (channel n/n+1 pair - 7 bits each)

  reg [6:0] a,b,c;
  reg [N_CHANNELS-1:0] k;

  assign dout = din ^ (init ? 14'h3fff : {a,a});
  assign sync_out = sync_in;

  always @(posedge clk `async_reset)
    if(!reset_l)begin
      k <= #1 0;
      a <= #1 7'h7f;
      b <= #1 7'h7f;
      c <= #1 7'h7f;
    end
    else if(en)begin
      k <= #1 sync_in ? 1 : (k==(`N_CHANNELS/2-1) ? 0 : k + 1);
      if(init)begin
        a <= #1 7'h7f;
        b <= #1 7'h7f;
        c <= #1 7'h7f;
      end
      else if(k==(`N_CHANNELS/2-1)) begin
        a <= #1 b;
        b <= #1 a ^ c;
        c <= #1 {a[3],a[2]^a[6],a[1]^a[5],a[0]^a[4],a[6],a[5],a[4]};
      end
    end
endmodule

`ifndef QAM256
  module trellisgroup(clk,reset_l,en_in,en_out,intervleave_mode,init_rnd,sync_in,sync_out,
    t,din,dout);
    input clk;
    input reset_l;
    input en_in;
    output en_out;
    input [3:0] interleave_mode;
    output init_rnd;
    input sync_in;

```

```

// write side parameters
  reg init_rnd; // flag to identify first word of new fec frame
  reg fec_1; // start of fec frame on first channel
  reg fec_2; // start of last 5 trellis groups with inserted sync
  headers reg [`NB_CHANNELS-2:0] kw; // channel pair number
  reg [2:0] i; // phase for 7 to 8 bit data multiplexers
  reg [7:0] n; // counter to maintain fec framing - counts each set of
  21 trellis groups until fec_2;
  reg [2:0] leapmode; // then counts each 7 bit write cycle until fec_1
  shift reg [2:0] leapmode_next; // identifies each 8 bit word with upcoming 2 bit phase
  reg [23:0] we; // next setting for leapmode
  reg wait; // write enables for all bits of each 8 bit word
  reg [2:0] re; // write enables for read side
  reg [2:0] fec_q; // write enables for read side

  // read side parameters
  reg [`NB_CHANNELS-1:0] ks; // channel number
  reg [2:0] re; // read enables for each 8 bit word
  reg [1:0] qre; // re1:0 delayed one on cycle
  reg fec_q; // flag used to queue read fec sync from write fec_sync
  reg fec_r; // start of fec frame on first channel

  wire sync = sync_in;
  wire en = en_in;
  wire [2:0] rdata_a,rdata_b;
  wire [7:0] date_a0 = i==3'd0 ? { din[13:7], 1'bx } :
    i==3'd1 ? { din[ 7], 1'bx, din[13: 7] } :
    i==3'd2 ? { din[ 7], 1'bx, din[13: 8] } :
    i==3'd3 ? { din[ 8:7], 1'bx, din[13: 9] } :
    i==3'd4 ? { din[ 9:7], 1'bx, din[13:10] } :
    i==3'd5 ? { din[10:7], 1'bx, din[13:11] } :
    i==3'd6 ? { din[11:7], 1'bx, din[13:12] } :
    { din[12:7], 1'bx, din[13] } ;
  wire [7:0] date_a1 = i==3'd6 ? { din[13:7], 1'bx } :
    i==3'd7 ? { din[ 7], 1'bx, din[13: 7] } :
    i==3'd0 ? { din[ 7], 1'bx, din[13: 8] } :
    i==3'd1 ? { din[ 8:7], 1'bx, din[13: 9] } :
    i==3'd2 ? { din[ 9:7], 1'bx, din[13:10] } :
    i==3'd3 ? { din[10:7], 1'bx, din[13:11] } :
    i==3'd4 ? { din[11:7], 1'bx, din[13:12] } :
    { din[12:7], 1'bx, din[13] } ;
  wire [7:0] date_b0 = i==3'd0 ? { din[6:0], 1'bx } :
    i==3'd1 ? { din[ 0], 1'bx, din[6:0] } :
    i==3'd2 ? { din[ 0], 1'bx, din[6:1] } :
    i==3'd3 ? { din[1:0], 1'bx, din[6:2] } :
    i==3'd4 ? { din[2:0], 1'bx, din[6:3] } :
    i==3'd5 ? { din[3:0], 1'bx, din[6:4] } :
    i==3'd6 ? { din[4:0], 1'bx, din[6:5] } :
    { din[5:0], 1'bx, din[ 6] } ;

  module trellisgroup(clk,reset_l,en_in,intervleave_mode,init_rnd,sync_in,sync_out,
    t,din,dout);
    input clk;
    input reset_l;
    input en_in;
    output en_out;
    input [3:0] interleave_mode;
    output init_rnd;
    input sync_in;

```

## fec.v

```

wire [7:0] data_b1 = i==3'd6 ? { din[6:0], 1'bx, din[6:0] } : 
i==3'd7 ? { din[ 0], 1'bx, din[6:1] } : 
i==3'd0 ? { din[ 0], 1'bx, din[6:2] } : 
i==3'd1 ? { din[1:0], 1'bx, din[6:3] } : 
i==3'd2 ? { din[2:0], 1'bx, din[6:4] } : 
i==3'd3 ? { din[3:0], 1'bx, din[6:5] } : 
i==3'd4 ? { din[4:0], 1'bx, din[6:6] } : 
{ din[5:0], 1'bx, din[ 6] } ;

wire [7:0] wdata_a2 = {leapmode[0]?we[2] ? data_a0;
wdata_a2 = {leapmode[0]?we[2] ? data_b0;
wire [7:0] wdata_a1 = {leapmode[2]?we[18] ? data_a0;
wdata_a1 = {leapmode[2]?we[18] ? data_b0;
wire [7:0] wdata_b1 = {leapmode[12]?we[18] ? data_b1;
wdata_b1 = {leapmode[12]?we[18] ? data_a0;
wire [7:0] wdata_a0 = {leapmode[1]?we[10] ? data_a0;
wdata_a0 = {leapmode[1]?we[10] ? data_b0;
wire [7:0] wdata_b0 = {leapmode[11]?we[10] ? data_b1;
wdata_b1 = {leapmode[11]?we[10] ? data_a0;

assign en_out = fec_1 & !sync | !fec_1 & !rvait;
wire [2:0] add2 = re[2] ? kr[3:1] : kw;
wire [2:0] add1 = re[1] ? kr[3:1] : kw;
wire [2:0] add0 = re[0] ? kr[3:1] : kw;

assign en_out = fec_1 & !sync | !fec_1 ? rdata_a[15:8];
wire [7:0] data = kr[0] ? (qre[0] ? rdata_b[7:0] : qre[1] ? rdata_a[15:8];
rdata_a[23:16]);
rdata_b[23:16]);
insertfeheader(clk, reset_1, en, interleave_mode, fec_r,
sync_out, data, dout);
rdata_a[23:16];
rdata_b[23:16];
end

```

```

wire [7:0] data_b1 = i==3'd6 ? { din[6:0], 1'bx, din[6:0] } : 
i==3'd7 ? { din[ 0], 1'bx, din[6:1] } : 
i==3'd0 ? { din[ 0], 1'bx, din[6:2] } : 
i==3'd1 ? { din[1:0], 1'bx, din[6:3] } : 
i==3'd2 ? { din[2:0], 1'bx, din[6:4] } : 
i==3'd3 ? { din[3:0], 1'bx, din[6:5] } : 
i==3'd4 ? { din[4:0], 1'bx, din[6:6] } : 
{ din[5:0], 1'bx, din[ 6] } ;

wire [7:0] wdata_a2 = {leapmode[0]?we[2] ? data_a0;
wdata_a2 = {leapmode[0]?we[2] ? data_b0;
wire [7:0] wdata_a1 = {leapmode[2]?we[18] ? data_a0;
wdata_a1 = {leapmode[2]?we[18] ? data_b0;
wire [7:0] wdata_b1 = {leapmode[12]?we[18] ? data_b1;
wdata_b1 = {leapmode[12]?we[18] ? data_a0;
wire [7:0] wdata_a0 = {leapmode[1]?we[10] ? data_a0;
wdata_a0 = {leapmode[1]?we[10] ? data_b0;
wire [7:0] wdata_b0 = {leapmode[11]?we[10] ? data_b1;
wdata_b1 = {leapmode[11]?we[10] ? data_a0;

assign en_out = fec_1 & !sync | !fec_1 & !rvait;
wire [2:0] add2 = re[2] ? kr[3:1] : kw;
wire [2:0] add1 = re[1] ? kr[3:1] : kw;
wire [2:0] add0 = re[0] ? kr[3:1] : kw;

assign en_out = fec_1 & !sync | !fec_1 ? rdata_a[15:8];
wire [7:0] data = kr[0] ? (qre[0] ? rdata_b[7:0] : qre[1] ? rdata_a[15:8];
rdata_a[23:16]);
rdata_b[23:16]);
insertfeheader(clk, reset_1, en, interleave_mode, fec_r,
sync_out, data, dout);
rdata_a[23:16];
rdata_b[23:16];
end
always @ (posedge clk) `async_reset)
if (reset_1) begin
    fec_1 <= #1 1;
    fec_2 <= #1 0;
    kw <= #1 0;
    i <= #1 0;
    n <= #1 0;
    leapnode <= #1 0;
    leapnode.next <= #1 3'b010;
    we <= #1 24'b0;
    rwait <= #1 1;
    kr <= #1 0;
    re <= #1 3'b100;
    qre <= #1 2'b00;
    fec_q <= #1 0;
    fec_r <= #1 0;
end else begin
    if (en) begin
        kr <= #1 kr + 1;
        qre <= #1 re[0];
        if (kr == (N_CHANNELS-1)) begin
            kr <= #1 0;
            re <= #1 fec_1 & sync & qre[1] | init_rnd & rvait;
            fec_q <= #1 fec_1 & sync & qre[1] | fec_q & !fec_r;
            if (fec_1) begin
                if (sync & qre[1])
                    fec_1 <= #1 0;
                fec_2 <= #1 0;
                kw <= #1 0;
                i <= #1 0;
                n <= #1 0;
            end
        end
    end
end

```

## fec.v

```

leapmode <= #1 0;
leapmode_next <= #1 3'b010;
we <= #1 { 7'b111111, 17'b0 };
rwait <= #1 { ((we[23], we[15], we[7]) & {~qre[1]&~qre[0], qre} );
end else begin
  if(kw==((N_CHANNELS/2)-1)) begin
    if(fec_2)
      leapmode <= #1 3'b111;
    else
      leapmode <= #1 { we[8], we[0], we[16] } & leapmode_next | leapmode &
~{we[18], we[10], we[12]};
    if(leapmode==leapmode_next)
      leapmode_next <= #1 { leapmode[1:0],leapmode[2] };
    kw <= #1 0;
    i <= #1 { (leapmode==3'b111 & i==3'5) ? 0 : (1'(leapmode &
(we[18], we[10], we[12])) ? (i + 1) : (i - 1));
    if((leapmode[0] & we[8] & we[2]) | fec_2)begin // this happens every
21 trellis groups (the repeat cycle)
      n <= #1 n + 1;
    end
    fec_1 <= #1 n==7'd120;
    fec_2 <= #1 n==7'd98 & leapmode_next[1] & we[3] & we[0] & we[21];
  endcase
  case(leapmode)
    3'b000: we <= #1 { we[6:0], we[23:7] };
    3'b001: we <= #1 { we[8:2], we[23:9], 2'b000 };
    3'b010: we <= #1 { we[6:0], we[23:17], 2'b00, we[16:10], we[7] };
    3'b100: we <= #1 { we[6:1], 2'b00, we[0], we[23:18], we[15:7] };
    3'b111: we <= #1 { we[6:1], 2'b00, we[12], we[23:19], 2'b00, we[18], we[15:11], 2'b00 };
    (we[10], we[7:3], 2'b00, we[12], we[23:19], 2'b00, we[18], we[15:11], 2'b00);
    default: we <= #1 24'hxxxxxx;
  endcase
  rwait <= #1 1;
  end else begin
    rwait <= #1 { ((we[23], we[15], we[7]) & {~qre[1]&~qre[0], qre} );
    kw <= #1 kw + 1;
  end // if(kw).
  end // if(fec_2..);
end // if(!reset..);
endmodule

```

```

module insertFchHeader(clk, reset_1, en, interleave_mode, sync_in, sync_out, din, dout);
  input clk;
  input rset_1;
  input en;
  input [3:0] interleave_mode;
  input sync_in;
  output sync_out;
  input [7:0] din;
  output [7:0] dout;
  wire [39:0] FECHDR = { 32'h71E84DD4, interleave_mode, 4'h0 };
  // FEC header for 256 QAM
endmodule

```

```

reg [7:0] d;
reg sync_out, sync2;
reg ['NB_CHANNELS-1:0] k;
reg [2:0] i;
reg [2:0] j;
reg [11:0] n;
reg last5;
assign sync_out = sync2;
assign dout = d;
always @ (posedge clk `async_reset)
  if(!reset_1)begin
    d <= #1 0;
    sync1 <= #1 0;
    sync2 <= #1 0;
    k <= #1 0;
    i <= #1 0;
    j <= #1 0;
    n <= #1 0;
    last5 <= #1 0;
  end else begin
    if(en)begin
      casex((last5,i,j))
        1'b0,3'bxxx,3'b0xx : d <= #1 { din[3],din[4],din[5],
        din[7],din[0],din[11],din[12], din[6];
        1'b1,3'bxxx,3'b1xx : d <= #1 { din[5],din[6],din[7],FECHDR[39],din[2],din[3],din[6],din[7],
        1'bxx,din[2],din[3],din[4], 1'bxx;
        1'b1,3'b000,3'b000 : d <= #1 { din[5],din[6],din[7],FECHDR[39],din[2],din[3],din[6],din[7],
        (din[5],din[6],din[7],FECHDR[39],din[2],din[3],din[6],din[7],FECHDR[38]);
        (din[5],din[6],din[7],FECHDR[37],din[2],din[3],din[6],din[7],FECHDR[36]);
        (din[5],din[6],din[7],FECHDR[35],din[2],din[3],din[6],din[7],FECHDR[34]);
        (din[5],din[6],din[7],FECHDR[35],din[2],din[3],din[6],din[7],FECHDR[34]);
        (din[5],din[6],din[7],FECHDR[35],din[2],din[3],din[6],din[7],FECHDR[33]);
        (din[5],din[6],din[7],FECHDR[35],din[2],din[3],din[6],din[7],FECHDR[32]);
        (din[5],din[6],din[7],FECHDR[29],din[2],din[3],din[6],din[7],FECHDR[28]);
        (din[5],din[6],din[7],FECHDR[29],din[2],din[3],din[6],din[7],FECHDR[27]);
        (din[5],din[6],din[7],FECHDR[25],din[2],din[3],din[6],din[7],FECHDR[26]);
        (din[5],din[6],din[7],FECHDR[25],din[2],din[3],din[6],din[7],FECHDR[24]);
        (din[5],din[6],din[7],FECHDR[23],din[2],din[3],din[6],din[7],FECHDR[22]);
        (din[5],din[6],din[7],FECHDR[21],din[2],din[3],din[6],din[7],FECHDR[20]);
        (din[5],din[6],din[7],FECHDR[19],din[2],din[3],din[6],din[7],FECHDR[18]);
        (din[5],din[6],din[7],FECHDR[17],din[2],din[3],din[6],din[7],FECHDR[16]);
        (din[5],din[6],din[7],FECHDR[15],din[2],din[3],din[6],din[7],FECHDR[14]);
        (din[5],din[6],din[7],FECHDR[13],din[2],din[3],din[6],din[7],FECHDR[12]);
    end
  end
end

```

## fec.v

```

{1'b1,3'b011,3'b010} : d <= #1;
(din[5],din[6],din[7],FECHDR[11],din[2],din[3],din[4],FECHDR[10],j
9),din[2],din[3],din[4],FECHDR[ 8]); d <= #1; (din[5],din[6],din[7],FECHDR[
7],din[2],din[3],din[4],FECHDR[ 6]);
{1'b1,3'b1xx,3'b001} : d <= #1; (din[5],din[6],din[7],FECHDR[5],
5),din[2],din[3],din[4],FECHDR[ 4]); d <= #1; (din[5],din[6],din[7],FECHDR[
3],din[2],din[3],din[4],FECHDR[ 2]);
{1'b1,3'b1xx,3'b010} : d <= #1; (din[5],din[6],din[7],FECHDR[1],
1],din[2],din[3],din[4],FECHDR[ 0]); d <= #1; 8'bxx;
endcase;
default:
d <= #1;

sync2 <= #1; sync1;
if(sync_in)begin
  k <= #1; 0;
  j <= #1; 0;
  i <= #1; 0;
  n <= #1; 0;
  last5 <= #1; 0;
  sync1 <= #1; 1;
end else begin
  if(k==(N_CHANNELS-1))begin
    j <= #1; j[2] ? 0 : j + 1;
    if(j[2])begin
      sync1 <= #1; 1;
      n <= #1; n + 1;
    end
    last5 <= #1; j[2] & n==12;d2070 | last5;
    if(last5 & j[2])
      i <= #1; i + 1;
      k <= #1; 0;
    end else
      sync1 <= #1; 0;
      k <= #1; k + 1;
    end
  end
end
endmodule

```

```

module gam_map (clk,reset_1,en,sync_in,sync_out,din,I,Q);
input clk;
input reset_1;
input en;
input sync_in;
output sync_out;
input [7:0] din;
output [4:0] I;
output [4:0] Q;
reg [7:0] d;
reg sync;
assign I = (d[7:4],1'b1);
assign Q = (d[3:0],1'b1);
assign sync_out = sync;

```

## Appendix -

fec.v

Appendix -

```

8'b0001000110: d <= #1 ('4'h8,4'h4);
b1b00110100: d <= #1 ('4'h9,4'h4);
b1b00111000: d <= #1 ('4'ha,4'h4);
b1b00010100: d <= #1 ('4'hb,4'h4);
b1b01001100: d <= #1 ('4'hc,4'h4);
b1b01100010: d <= #1 ('4'hd,4'h4);
b1b01000100: d <= #1 ('4'he,4'h4);
b1b01100001: d <= #1 ('4'hf,4'h4);
b1b01000011: d <= #1 ('4'hg,4'h4);
b1b01100000: d <= #1 ('4'hh,4'h4);
b1b01110000: d <= #1 ('4'hi,4'h4);
b1b00011100: d <= #1 ('4'hz,4'h4);
b1b00111000: d <= #1 ('4'h2,4'h4);
b1b00010011: d <= #1 ('4'h3,4'h4);
b1b01100011: d <= #1 ('4'h4,4'h4);
b1b00010001: d <= #1 ('4'h5,4'h4);
b1b01100001: d <= #1 ('4'h6,4'h4);
b1b00010000: d <= #1 ('4'h7,4'h4);

b0b00110000: d <= #1 ('4'h1,4'h4);
b0b01100000: d <= #1 ('4'h2,4'h4);
b0b00111000: d <= #1 ('4'h3,4'h4);
b0b01111000: d <= #1 ('4'h4,4'h4);
b0b01001100: d <= #1 ('4'h5,4'h4);
b0b01101100: d <= #1 ('4'h6,4'h4);
b0b01000100: d <= #1 ('4'h7,4'h4);

b0b10110111: d <= #1 ('4'h8,4'h5');
b0b10111011: d <= #1 ('4'ha,4'h5);
b0b10101101: d <= #1 ('4'hb,4'h5);
b0b10111001: d <= #1 ('4'hc,4'h5);
b0b10101100: d <= #1 ('4'hd,4'h5);
b0b10110011: d <= #1 ('4'he,4'h5);
b0b10100111: d <= #1 ('4'hf,4'h5);
b0b10110001: d <= #1 ('4'hg,4'h5);
b0b10100011: d <= #1 ('4'hh,4'h5);
b0b10110000: d <= #1 ('4'hi,4'h5);
b0b10111000: d <= #1 ('4'hz,4'h5);
b0b10011100: d <= #1 ('4'hh,4'h5);
b0b10111000: d <= #1 ('4'hi,4'h5);
b0b10010011: d <= #1 ('4'hh,4'h5);
b0b10110001: d <= #1 ('4'hh,4'h5);
b0b10010001: d <= #1 ('4'hh,4'h5);
b0b10110000: d <= #1 ('4'hh,4'h5);

b1b11100001: d <= #1 ('4'h8,4'h6);
b1b11010001: d <= #1 ('4'ha,4'h6);
b1b11001001: d <= #1 ('4'hb,4'h6);
b1b11000101: d <= #1 ('4'hc,4'h6);
b1b11000010: d <= #1 ('4'hd,4'h6);
b1b11000100: d <= #1 ('4'he,4'h6);
b1b11000010: d <= #1 ('4'hf,4'h6);
b1b11000001: d <= #1 ('4'hg,4'h6);
b1b11000000: d <= #1 ('4'hh,4'h6);
b1b11000110: d <= #1 ('4'hi,4'h6);
b1b11000111: d <= #1 ('4'hz,4'h6);
b1b11000011: d <= #1 ('4'hh,4'h6);
b1b11000101: d <= #1 ('4'hi,4'h6);
b1b11000010: d <= #1 ('4'hh,4'h6);
b1b11000001: d <= #1 ('4'hh,4'h6);
b1b11000000: d <= #1 ('4'hh,4'h6);

b1b11100010: d <= #1 ('4'h8,4'h7);
b1b11010010: d <= #1 ('4'ha,4'h7);
b1b11001010: d <= #1 ('4'hb,4'h7);
b1b11000110: d <= #1 ('4'hc,4'h7);
b1b11000010: d <= #1 ('4'hd,4'h7);
b1b11000100: d <= #1 ('4'he,4'h7);
b1b11000010: d <= #1 ('4'hf,4'h7);
b1b11000001: d <= #1 ('4'hg,4'h7);
b1b11000000: d <= #1 ('4'hh,4'h7);
b1b11000110: d <= #1 ('4'hi,4'h7);
b1b11000111: d <= #1 ('4'hz,4'h7);
b1b11000011: d <= #1 ('4'hh,4'h7);
b1b11000101: d <= #1 ('4'hi,4'h7);
b1b11000010: d <= #1 ('4'hh,4'h7);
b1b11000001: d <= #1 ('4'hh,4'h7);
b1b11000000: d <= #1 ('4'hh,4'h7);

b1b11100011: d <= #1 ('4'h8,4'h8);
b1b11010011: d <= #1 ('4'ha,4'h8);
b1b11001011: d <= #1 ('4'hb,4'h8);
b1b11000111: d <= #1 ('4'hc,4'h8);
b1b11000011: d <= #1 ('4'hd,4'h8);
b1b11000101: d <= #1 ('4'he,4'h8);
b1b11000011: d <= #1 ('4'hf,4'h8);
b1b11000001: d <= #1 ('4'hg,4'h8);
b1b11000000: d <= #1 ('4'hh,4'h8);
b1b11000110: d <= #1 ('4'hi,4'h8);
b1b11000111: d <= #1 ('4'hz,4'h8);
b1b11000011: d <= #1 ('4'hh,4'h8);
b1b11000101: d <= #1 ('4'hi,4'h8);
b1b11000010: d <= #1 ('4'hh,4'h8);
b1b11000001: d <= #1 ('4'hh,4'h8);
b1b11000000: d <= #1 ('4'hh,4'h8);

```

fec.v

## fec.v

```

input clk;
input reset_1;
input en_in;
output en_out;
input [3:0] interleave_mode;
output init_rd;
input sync_in;
output sync_out;
input [13:0] din;
output [7:0] dout;

// write side parameters
reg init_rd; // flag to identify first word of new fec frame
reg [6:0] fec; // gate for each 7 bit fec header word
reg [1:0] fecphase; // identifies 4 possible fec header alignments
reg [NB_CHANNELS-2:0] kr; // channel pair number
reg [9:0] n; // counter to maintain fec framing - counts 2 x 960
trellis groups
reg [2:0] we;
each) reg [1:0] sync; // write enables for each 7 bit word (2 banks of 4 words
each) reg [1:0] rwait; // write side waiting for read side

// read side parameters
reg [NB_CHANNELS-1:0] kr; // channel number
reg [3:0] re; // read enables for each 8 bit word (2 banks of 5 words
each)
reg [3:0] qre; // re delayed one en cycle
reg [1:0] sync; // flag indicating start of new trellis group
wire en = en_in;
reg [7:0] rd;
wire [27:0] rd0, rd1, rd2, rd3;

wire [2:0] addr0 = re[3] ? kr[3:1] : kw;
wire [2:0] addr1 = re[3] ? kr[3:1] : kw;
assign en_out = init_rd & !sync_in | fec[0] & !rwait;
assign sync_out = sync[1];
assign dout = rd;

wire [13:0] d = fec[1] ? { 7'h75, 7'h75 } :
fec[2] ? { 7'h2c, 7'h2c } :
fec[3] ? { 7'h0d, 7'h0d } :
fec[4] ? { 7'h0c, 7'h0c } :
fec[5] ? { interleave_mode, 3'h0, interleave_mode, 3'h0 } :
fec[6] ? 14'h0 : dina;
// even channels bank 1
ram_ars #(3,7) _tgram_a7(clk, !rwait&we==3'd0, addr0_d[13:7], rd0[27:21]);
ram_ars #(3,7) _tgram_a6(clk, !rwait&we==3'd1, addr0_d[13:7], rd0[20:14]);
ram_ars #(3,7) _tgram_a5(clk, !rwait&we==3'd2, addr0_d[13:7], rd0[13:7]);
ram_ars #(3,7) _tgram_a4(clk, !rwait&we==3'd3, addr0_d[13:7], rd0[ 6: 0]);
// even channels bank 2
ram_ars #(3,7) _tgram_a3(clk, !rwait&we==3'd4, addr1_d[13:7], rd1[27:21]);
ram_ars #(3,7) _tgram_a2(clk, !rwait&we==3'd5, addr1_d[13:7], rd1[20:14]);
ram_ars #(3,7) _tgram_a1(clk, !rwait&we==3'd6, addr1_d[13:7], rd1[ 6: 0]);

```

```

ram_ars #(3,7) _tgram_a0(clk, !rwait&we==3'd7, addrl1_d[13:7], rd1[ 6: 0]);
// odd channels bank 1
ram_ars #(3,7) _tgram_b7(clk, !rwait&we==3'd0, addrl0_d[ 6:0], rd2[27:21]);
ram_ars #(3,7) _tgram_b6(clk, !rwait&we==3'd1, addrl0_d[ 6:0], rd2[20:14]);
ram_ars #(3,7) _tgram_b5(clk, !rwait&we==3'd2, addrl0_d[ 6:0], rd2[13:7]);
ram_ars #(3,7) _tgram_b4(clk, !rwait&we==3'd3, addrl0_d[ 6:0], rd2[ 6: 0]);
// odd channels bank 2
ram_ars #(3,7) _tgram_b3(clk, !rwait&we==3'd4, addrl1_d[ 6:0], rd3[27:21]);
ram_ars #(3,7) _tgram_b2(clk, !rwait&we==3'd5, addrl1_d[ 6:0], rd3[20:14]);
ram_ars #(3,7) _tgram_b1(clk, !rwait&we==3'd6, addrl1_d[ 6:0], rd3[13:7]);
ram_ars #(3,7) _tgram_b0(clk, !rwait&we==3'd7, addrl1_d[ 6:0], rd3[ 6: 0]);

always @ (posedge clk `async_reset)
begin
if(!reset_1)begin
    sync <= #1 0;
    kr <= #1 0;
    re <= #1 0;
    qre <= #1 0;
    kw <= #1 0;
    we <= #1 3'b0;
    fec <= #1 7'b00000001;
    fephase <= #1 2'b11;
    n <= #1 0;
    init_end <= #1 1;
    rwait <= #1 1;
    rd <= #1 0;
end else begin
    if(kr== ( N_CHANNELS-1))begin
        kr <= #1 0;
        sync <= #1 (sync[0], kr==0 & re[2:0]==0);
        kr <= #1 kr + 1;
        qre <= #1 re;
    end
    re <= #1 re[2] ? (~re[3], 3'b0) : (re[3], re[2:0]&3'b1);
    end
    rd <= #1 kr[0] ? (qre==4'd0 ? { 1'b0, rd0[22], rd0[21], rd0[17],
1'b0, rd0[ 8], rd0[ 7], rd0[ 3] } :
qre==4'd1 ? { 1'b0, rd0[24], rd0[23], rd0[18],
1'b0, rd0[10], rd0[ 9], rd0[ 4] } :
1'b0, rd0[12], rd0[11], rd0[ 5] } :
qre==4'd3 ? { 1'b0, rd0[27], rd0[26], rd0[25],
1'b0, rd0[14], rd0[13], rd0[ 6] } :
qre==4'd4 ? { 1'b0, rd0[16], rd0[15], 1'bx,
1'b0, rd0[ 2], rd0[ 1], 1'bx } ;
rd1[ 8], rd1[ 7], rd1[ 3] ) :
1'b0, rd1[10], rd1[ 9], rd1[ 4] ) );
    qre==4'd9 ? { 1'b0, rd1[24], rd1[23], rd1[18],
1'b0, rd1[12], rd1[11], rd1[ 5] } ;
    qre==4'd10 ? { 1'b0, rd1[26], rd1[25], rd1[19],
1'b0, rd1[ 9], rd1[13], rd1[ 6] } ;
    qre==4'd11 ? { 1'b0, rd1[14], rd1[27], rd1[20],
1'b0, rd1[ 0], rd1[12], rd1[ 1] } ;
    qre==4'd12 ? { 1'b0, rd1[16], rd1[15], 1'bx,
1'b0, rd1[ 2], rd1[ 1], 1'bx } ;
8'bx ) ;

```

## fec.v

```

fec[ 8], rd2[ 7], rd2[ 3] ): {qre==4'd0 ? { 1'b0, rd2[22], rd2[21], rd2[17], 1'b0,
1'b0, rd2[10], rd2[ 9], rd2[ 4] } ;
1'b0, rd2[12], rd2[11], rd2[ 5] ): {qre==4'd2 ? { 1'b0, rd2[26], rd2[25], rd2[19],
1'b0, rd2[ 0], rd2[13], rd2[ 6] } ;
1'b0, rd2[ 2], rd2[ 1], rd2[17], qre==4'd4 ? { 1'b0, rd2[16], rd2[15], 1'bx,
rd3[ 8], rd3[ 7], rd3[ 3] ): {qre==4'd8 ? { 1'b0, rd3[22], rd3[21], rd3[17], 1'b0,
1'b0, rd3[10], rd3[ 9], rd3[ 4] }: {qre==4'd9 ? { 1'b0, rd3[24], rd3[23], rd3[18],
1'b0, rd3[12], rd3[11], rd3[ 5] }: {qre==4'd10 ? { 1'b0, rd3[26], rd3[25], rd3[19],
1'b0, rd3[ 0], rd3[13], rd3[ 6] }: {qre==4'd11 ? { 1'b0, rd3[14], rd3[12], rd3[20],
1'b0, rd3[ 2], rd3[ 1], rd3[15], 1'bx,
end
int_rnd <= #1 fec[1] | init_rnd & !(fec[0] | sync_in | rwait);
if(kw==('N_CHANNELS/2)-1)begin
we <= #1 we + 1;
fec[0] <= #1 fec[6] | fec[0] & !(n==10'd959 & we==(fecphase,1'b1));
fec[1] <= #1 we[1:0]==10'd959 & we==(fecphase,1'b1);
fec[6:2] <= #1 fec[5:1];
if(fec[6])
fecphase <= #1 fecphase - 1;
if(fec[6])
n <= #1 0;
else if(we==(fecphase,1'b1))
n <= #1 n + 1;
rwait <= #1 we[1:0]==2'b11 | init_rnd & fec[0] & !sync_in;
kw <= #1 0;
end
end begin
rwait <= #1 we == {qre[3],2'b0} | init_rnd & fec[0] & !sync_in;
end // if(kw..)
end // if(!reset..)
endmodule

module qm_map (clk,reset_l,en,sync_in,sync_out,din,I,Q);
input clk;
input reset_l;
input en;
input sync_in;
output sync_out;
input [7:0] din;
output [4:0] I;
output [4:0] Q;
reg [5:0] d;
reg sync;
assign I = (d[5:3],2'b10);

```

```

assign Q = (d[2:0],2'b10);
assign sync_out = sync;
always @ (posedge clk `async_reset)
if(!reset_l)begin
d <= #1 0;
sync <= #1 0;
end
else if(en)begin
sync <= #1 sync_in;
case(din)
8'b01100111: d <= #1 (3'h4,3'h3);
8'b01100110: d <= #1 (3'h5,3'h3);
8'b00100111: d <= #1 (3'h6,3'h3);
8'b00100110: d <= #1 (3'h7,3'h3);
8'b01000101: d <= #1 (3'h0,3'h3);
8'b01000100: d <= #1 (3'h1,3'h3);
8'b01100101: d <= #1 (3'h2,3'h3);
8'b01100100: d <= #1 (3'h3,3'h3);
8'b01100011: d <= #1 (3'h4,3'h2);
8'b01100010: d <= #1 (3'h5,3'h2);
8'b00100010: d <= #1 (3'h6,3'h2);
8'b00100000: d <= #1 (3'h7,3'h2);
8'b01000011: d <= #1 (3'h0,3'h2);
8'b01000010: d <= #1 (3'h1,3'h2);
8'b01100001: d <= #1 (3'h2,3'h2);
8'b01100000: d <= #1 (3'h3,3'h2);
8'b01100101: d <= #1 (3'h4,3'h1);
8'b01100100: d <= #1 (3'h5,3'h1);
8'b00100101: d <= #1 (3'h6,3'h1);
8'b00100100: d <= #1 (3'h7,3'h1);
8'b01000011: d <= #1 (3'h0,3'h1);
8'b01000010: d <= #1 (3'h1,3'h1);
8'b01100011: d <= #1 (3'h2,3'h1);
8'b01100010: d <= #1 (3'h3,3'h1);
8'b00100011: d <= #1 (3'h4,3'h0);
8'b00100010: d <= #1 (3'h5,3'h0);
8'b00000000: d <= #1 (3'h7,3'h0);
8'b00000001: d <= #1 (3'h0,3'h0);
8'b00000000: d <= #1 (3'h1,3'h0);
8'b00000001: d <= #1 (3'h2,3'h0);
8'b00000011: d <= #1 (3'h3,3'h0);
8'b00000010: d <= #1 (3'h4,3'h7);
8'b0000110001: d <= #1 (3'h5,3'h7);
8'b00000001: d <= #1 (3'h6,3'h7);
8'b00000000: d <= #1 (3'h7,3'h7);
8'b00000001: d <= #1 (3'h0,3'h7);
8'b00000000: d <= #1 (3'h1,3'h7);
8'b00000001: d <= #1 (3'h2,3'h7);
8'b000010101: d <= #1 (3'h3,3'h7);
8'b000010100: d <= #1 (3'h4,3'h6);
8'b000010100: d <= #1 (3'h5,3'h6);
8'b00000010: d <= #1 (3'h6,3'h6);

```

fec.v

```

if(!reset_1)begin
    dout <= #1 0;
    sync <= #1 0;
    j <= #1 0;
    k <= #1 0;
end else begin
    if(en)begin
        dout <= #1 d;
        sync <= #1 sync_in | k==0;
        if(sync_in)begin
            k <= #1 3'd0;
            j <= #1 1;
        end else begin
            j <= #1 0;
            k <= #1 j==3'd4 ? 0 : (j + 1);
        end
    end
endmodule

module diff_pre_coder(clk,en,c,din,dout);
    input clk;
    input en;
    input [NB_CHANNELS-1:0] c;
    input [1:0] din;
    output [1:0] dout;
    wire [1:0] d;
    wire t = din[0] & (d[0] ^ d[1]);
    assign dout = din & d ^ {1'b0,din[1]} ^ {t,t};
    ram_ar #(4,2) _dpc(clk,en,c,dout,d);
endmodule

module bcc(clk,en,c,p3,p4,din,dout);
    input clk;
    input en;
    input [NB_CHANNELS-1:0] c;
    input p3;
    input p4;
    input din;
    output dout;
    reg g1d;
    wire [4:0] d;
    wire dx = p4 ? d[0] : din;
    assign dout = p3 ? (din ^ d[3] ^ d[1]) : (dx ^ d[4] ^ d[3] ^ d[2] ^ d[1]);
    ram_ar #(4,5) _bcc(clk,en,[p4,c,(din,d[4:1]),d]);
endmodule

module ram_ar(clk,we,addr,di,do);
parameter N4=4,N1=1;

```

fec.v

```

input clk;
input we;
input [M-1:0] addr;
input [N-1:0] di;
output [N-1:0] do;
reg [N-1:0] mem[0:(1<<M)-1];
assign do = mem[addr];

always @ (posedge clk)
  if (we) begin
    mem[addr] <= #1 di;
  end
endmodule

module ram_ars(clk,we,addr,di,do);
parameter M=4,N=1;
input clk;
input we;
input [M-1:0] addr;
input [N-1:0] di;
output [N-1:0] do;

reg [N-1:0] do;
wire [N-1:0] d;
ram #(N,M) r(clk,addr,di,we,addr,d);
always @ (posedge clk)
  do <= #1 d;
endmodule

```

Appendix -

lib.v

**freq\_shift.v**

```

// Shift frequency by pi/24 to obtain odd stacking.
// Copyright 2002 rgb media, inc.
// Peter Monta
//



module freq_shift_24 (clk, reset,
xi, xqf,
yi, yqf;
input clk, reset_l;
input [13:0] xi, xq;
output [13:0] yi, yq;
//local
reg [3:0] addr;
modulate_even m0(clk,addr,ai,aq,ci,cq);
modulate_odd m1(clk,addr,bi,bq,di,dq);
always @ (posedge clk or negedge reset_l)
if (reset_l)
addr <= #1 0;
else
addr <= #1 addr + 1;
endmodule

module modulate_even(clk,addr,ai,aq,bi,bq);
input clk;
input [3:0] addr;
input [13:0] ai,aq;
output [13:0] bi,bq;
rom_pi32_even_i rom0(clk,addr,wi);
rom_pi32_even_q rom1(clk,addr,wq);
complex_mult c0(clk,1'b1,ai,aq1,wi,wq,bi,bq);
always @ (posedge clk) begin
ai1 <= #1 ai;
aq1 <= #1 aq;
end
endmodule

module rom_pi32_even_i (clk,addr,ai,aq,bi,bq);
input clk;
input [1:0] addr;
output [13:0] wi;
reg [13:0] w;
always @ (posedge clk)
case (addr)
endmodule

module rom_pi32_even_q (clk,addr,wi);
input clk;
input [1:0] addr;
output [13:0] wq;
reg [13:0] wq;
always @ (posedge clk)
begin
wi1 <= #1 wi;
end
endmodule

```

```

0: w = 14'd8191;
1: w = 14'd8094;
2: w = 14'd7567;
3: w = 14'd6811;
4: w = 14'd5722;
5: w = 14'd4551;
6: w = 14'd3155;
7: w = 14'd1598;
8: w = 14'd1'00;
9: w = 14'd14'86;
10: w = 14'd12249;
11: w = 14'd11833;
12: w = 14'd10592;
13: w = 14'd9573;
14: w = 14'd8817;
15: w = 14'd8350;
endcase
endmodule

module rom_pi32_even_q(clk,addr,wi);
input clk;
input [1:0] addr;
output [13:0] w;
reg [13:0] w;
always @ (posedge clk)
case (addr)
0: w <= #1 14'd0;
1: w <= #1 14'd1786;
2: w <= #1 14'd1249;
3: w <= #1 14'd11833;
4: w <= #1 14'd10592;
5: w <= #1 14'd9573;
6: w <= #1 14'd8817;
7: w <= #1 14'd8350;
8: w <= #1 14'd8193;
9: w <= #1 14'd8550;
10: w <= #1 14'd8817;
11: w <= #1 14'd5573;
12: w <= #1 14'd0592;
13: w <= #1 14'd11833;
14: w <= #1 14'd3249;
15: w <= #1 14'd4786;
endcase
endmodule

module modulate_odd(clk,addr,ai,aq,bi,bq);
input clk;
input [3:0] addr;
input [13:0] ai,aq;
output [13:0] bi,bq;
rom_pi32_odd_i rom0(clk,addr,wi);
rom_pi32_odd_q rom1(clk,addr,wq);
complex_mult c0(clk,1'b1,ai,aq1,wi,wq,bi,bq);
always @ (posedge clk) begin
ai1 <= #1 ai;
aq1 <= #1 aq;
end
endmodule

module rom_pi32_odd_i (clk,addr,ai,aq,bi,bq);
input clk;
input [1:0] addr;
output [13:0] wi;
reg [13:0] w;
always @ (posedge clk)
begin
wi1 <= #1 wi;
end
endmodule

```

## lib.v

```

addr <= #1 aq;
end

endmodule

module rom_pi32_0dd_i(clk, addr, w);
input clk;
input [1:0] addr;
output [13:0] w;
reg [13:0] v;
always @ (posedge clk)
  case (addr)
    0: w = 14'd8152;
    1: w = 14'd7838;
    2: w = 14'd7224;
    3: w = 14'd6332;
    4: w = 14'd5196;
    5: w = 14'd3361;
    6: w = 14'd2378;
    7: w = 14'd803;
    8: w = 14'd1581;
    9: w = 14'd1406;
    10: w = 14'd12523;
    11: w = 14'd11188;
    12: w = 14'd005;
    13: w = 14'd1160;
    14: w = 14'd8546;
    15: w = 14'd232;
  endcase
endmodule

module rom_pi32_0dd_q(clk, addr, w);
input clk;
input [1:0] addr;
output [13:0] w;
reg [13:0] v;
always @ (posedge clk)
  case (addr)
    0: v <= #1 14'd15581;
    1: v <= #1 14'd4006;
    2: v <= #1 14'd12523;
    3: v <= #1 14'd11188;
    4: v <= #1 14'd0032;
    5: v <= #1 14'd91160;
    6: v <= #1 14'd8546;
    7: v <= #1 14'd8232;
    8: v <= #1 14'd8232;
    9: v <= #1 14'd8546;
    10: v <= #1 14'd9160;
    11: v <= #1 14'd10052;
    12: v <= #1 14'd1188;
    13: v <= #1 14'd12523;
    14: v <= #1 14'd14006;
    15: v <= #1 14'd15581;
  endcase
endmodule

```

**lib.v**

lib.v

```

SRL16E f11(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
SRL16E f12(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
SRL16E f13(t[13],a[0],a[1],a[2],a[3],ce,clk,in[13]);
SRL16E f14(t[14],a[0],a[1],a[2],a[3],ce,clk,in[14]);
SRL16E f15(t[15],a[0],a[1],a[2],a[3],ce,clk,in[15]);

// Shift register, depth 8, width 10
//
module sr8_10(clk, ce, in, out);
  input clk, ce;
  input [9:0] in;
  output [9:0] out;
//local
  wire [3:0] a = 7; //depth 8
  wire [9:0] t;

  assign out = t;
/
  module sr8_14(clk, ce, in, out);
    input clk, ce;
    input [13:0] in;
    output [11:0] out;
    //
    // local
    wire [3:0] a = 7; //depth 8
    wire [13:0] t;
    assign out = t;
    SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
    SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
    SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
    SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
    SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
    SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
    SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
    SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
    SRL16E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
    SRL16E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
  endmodule
  //
  // Shift register, depth 8, width 16
  //
  module sr8_16(clk, ce, in, out);
    input clk, ce;
    input [15:0] in;
    output [15:0] out;
  endmodule
  //
  // local
  wire [3:0] a = 7; //depth 8
  wire [15:0] t;
  assign out = t;
  module sr6_14(clk, ce, in, out);
    input clk, ce;
    input [11:0] in;
    output [9:0] out;
    //
    // local
    wire [3:0] a = 15; //depth 16
    wire [13:0] t;
    assign out = t;
    SRL16E f1(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
    SRL16E f2(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
    SRL16E f3(t[13],a[0],a[1],a[2],a[3],ce,clk,in[13]);
    SRL16E f4(t[14],a[0],a[1],a[2],a[3],ce,clk,in[14]);
    SRL16E f5(t[15],a[0],a[1],a[2],a[3],ce,clk,in[15]);
    SRL16E f6(t[16],a[0],a[1],a[2],a[3],ce,clk,in[16]);
    SRL16E f7(t[17],a[0],a[1],a[2],a[3],ce,clk,in[17]);
    SRL16E f8(t[18],a[0],a[1],a[2],a[3],ce,clk,in[18]);
    SRL16E f9(t[19],a[0],a[1],a[2],a[3],ce,clk,in[19]);
    SRL16E f10(t[20],a[0],a[1],a[2],a[3],ce,clk,in[20]);
  endmodule
  //
  // Shift register, depth 16, width 14
  //
  module sr16_14(clk, ce, in, out);
    input clk, ce;
    input [13:0] in;
    output [11:0] out;
    //
    // local
    wire [3:0] a = 15; //depth 16
    wire [13:0] t;
    assign out = t;
    SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
    SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
    SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
    SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
    SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
    SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
    SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
    SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
    SRL16E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
    SRL16E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
    SRL16E f10(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
  endmodule

```

## lib.v

```

SR116E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
SR116E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
SR116E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
SR116E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
SR116E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
SR116E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
SR116E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
SR116E f10(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
SR116E f11(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
SR116E f12(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
SR116E f13(t[13],a[0],a[1],a[2],a[3],ce,clk,in[13]);
endmodule

// Shift register, depth 16, width 16
// Local
wire [3:0] a = 15; //depth 16
wire [15:0] t;
assign out = t;
endmodule

module sr16_16(clk, ce, in, out);
input clk, ce;
input [15:0] in;
output [15:0] out;
endmodule

// SR16E F0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
SR116E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
SR116E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
SR116E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
SR116E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
SR116E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
SR116E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
SR116E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
SR116E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
SR116E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
SR116E f10(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
SR116E f11(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
SR116E f12(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
SR116E f13(t[13],a[0],a[1],a[2],a[3],ce,clk,in[13]);
SR116E f14(t[14],a[0],a[1],a[2],a[3],ce,clk,in[14]);
SR116E f15(t[15],a[0],a[1],a[2],a[3],ce,clk,in[15]);
endmodule

// SR16E F16(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
SR116E f16(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
SR116E f17(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
SR116E f18(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
SR116E f19(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
SR116E f20(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
SR116E f21(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
SR116E f22(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
SR116E f23(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
SR116E f24(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
SR116E f25(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
SR116E f26(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
SR116E f27(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
SR116E f28(t[13],a[0],a[1],a[2],a[3],ce,clk,in[13]);
SR116E f29(t[14],a[0],a[1],a[2],a[3],ce,clk,in[14]);
SR116E f30(t[15],a[0],a[1],a[2],a[3],ce,clk,in[15]);
endmodule

// Registered 2-input mux with enable
parameter W = 8;
input clk, en;
input sel;
input [W-1:0] x0, x1;
output [W-1:0] y;
reg [(W-1):0] y;
always @ (posedge clk)
  if (en)
    y <= #1 sel ? x1 : x0;
endmodule

// Registered 6-input mux with enable. The select bits
// are nonbinary; sel[2] specifies x[0|12] or x[3|45],
// then sel[1:0] selects within these groups of three.
// Local
parameter W = 8;
input clk, en;
input [2:0] sel;
input [(W-1):0] x0, x1, x2, x3, x4, x5;
output [(W-1):0] y;
reg [(W-1):0] y;

module r mux6(clk, en, sel, x0, x1, x2, x3, x4, x5, y);
begin
  assign y = (sel == 2) ? x0 : (sel == 1) ? x1 : (sel == 0) ? x2 : (sel == 3) ? x3 : (sel == 4) ? x4 : (sel == 5) ? x5 : x6;
end

module sr12_16(clk, ce, in, out);
input clk, ce;
input [15:0] in;
output [15:0] out;
endmodule

// Shift register, depth 12, width 16
// Local

```

## lib.v

```

reg [(W-1):0] ny;
always @ (sel or x0 or x1 or x2 or x3 or x4 or x5)
  case (sel)
    3'd0: ny = x0;
    3'd1: ny = x1;
    3'd2: ny = x2;
    3'd3: ny = 1'bx;
    3'd4: ny = x3;
    3'd5: ny = x4;
    3'd6: ny = x5;
  endcase
endmodule

always @ (posedge clk)
  if (len)
    y <= #1 ny;

// Register with clock enable
// module rreg(clk, ce, in, out);
// parameter W=;
//   input clk, ce;
//   input [W-1:0] in;
//   output [W-1:0] out;
// endmodule

reg [W-1:0] out;
always @ (posedge clk)
  if (ce)
    out <= #1 in;
endmodule

// SRL-based delay, 5-bit width
module delay5(clk, ce, in, out)
  parameter D = 1;
  input clk, ce;
  input [4:0] in;
  output [4:0] out;
//local
  wire [3:0] a = D - 1;
  wire [4:0] t;
endmodule

SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);

rreg #(5) rr(clk, ce, t, out);
endmodule

```

```

rreg #(5) rr(clk, ce, t, out);
endmodule

// SRL-based delay, 8-bit width
module delay8(clk, ce, in, out);
  parameter D = 1;
  input clk, ce;
  input [7:0] in;
  output [7:0] out;
//local
  wire [3:0] a = D - 1;
  wire [7:0] t;
endmodule

SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);

rreg #(8) rr(clk, ce, t, out);
endmodule

// SRL-based delay, 10-bit width
module delay10(clk, ce, in, out);
  parameter D = 1;
  input clk, ce;
  input [9:0] in;
  output [9:0] out;
//local
  wire [3:0] a = D - 1;
  wire [9:0] t;
endmodule

SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
SRL16E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
SRL16E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);

rreg #(10) rr(clk, ce, t, out);
endmodule

```

lib.v

```

// SRL-based delay, 13-bit width
// Parameter D = 1;
module delay13(clk, ce, in, out);
    input clk, ce;
    input [12:0] in;
    output [12:0] out;
    //local
    wire [13:0] a = D - 1;
    wire [12:0] t;
    SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
    SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
    SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
    SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
    SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
    SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
    SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
    SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
    SRL16E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
    SRL16E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
    SRL16E f10(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
    SRL16E f11(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
    SRL16E f12(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
    rreg # (13) rr(clk, ce, t, out);
    endmodule
    // SRL-based delay, 14-bit width
    // Parameter D = 1;
module delay14(clk, ce, in, out);
    input clk, ce;
    input [13:0] in;
    output [13:0] out;
    //local
    wire [13:0] a = D - 1;
    wire [13:0] t;
    SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
    SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
    SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
    SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
    SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
    SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
    SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
    SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
    SRL16E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
    SRL16E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
    SRL16E f10(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
    SRL16E f11(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
    SRL16E f12(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
    rreg # (14) rr(clk, ce, t, out);
    endmodule
    // SRL-based delay, 15-bit width
    // Parameter D = 1;
module delay15(clk, ce, in, out);
    input clk, ce;
    input [14:0] in;
    output [14:0] out;
    //local
    wire [14:0] a = D - 1;
    wire [14:0] t;
    SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
    SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
    SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
    SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
    SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
    SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
    SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
    SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
    SRL16E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
    SRL16E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
    SRL16E f10(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
    SRL16E f11(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
    SRL16E f12(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
    rreg # (15) rr(clk, ce, t, out);
    endmodule
    // SRL-based delay, 16-bit width
    // Parameter D = 1;
module delay16(clk, ce, in, out);
    input clk, ce;
    input [15:0] in;
    output [15:0] out;
    //local
    wire [15:0] a = D - 1;
    wire [15:0] t;
    SRL16E f0(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
    SRL16E f1(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
    SRL16E f2(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
    SRL16E f3(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
    SRL16E f4(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
    SRL16E f5(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
    SRL16E f6(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);
    SRL16E f7(t[7],a[0],a[1],a[2],a[3],ce,clk,in[7]);
    SRL16E f8(t[8],a[0],a[1],a[2],a[3],ce,clk,in[8]);
    SRL16E f9(t[9],a[0],a[1],a[2],a[3],ce,clk,in[9]);
    SRL16E f10(t[10],a[0],a[1],a[2],a[3],ce,clk,in[10]);
    SRL16E f11(t[11],a[0],a[1],a[2],a[3],ce,clk,in[11]);
    SRL16E f12(t[12],a[0],a[1],a[2],a[3],ce,clk,in[12]);
    rreg # (16) rr(clk, ce, t, out);
    endmodule

```

## lib.v

```

SRL16 f5#(5) a[0],a[1],a[2],a[3],ce,clk,in[5];
SRL16 f6#(6) a[0],a[1],a[2],a[3],ce,clk,in[6];
SRL16 f7#(7) a[0],a[1],a[2],a[3],ce,clk,in[7];
SRL16 f8#(8) a[0],a[1],a[2],a[3],ce,clk,in[8];
SRL16 f9#(9) a[0],a[1],a[2],a[3],ce,clk,in[9];
SRL16 f10#(10) a[0],a[1],a[2],a[3],ce,clk,in[10];
SRL16 f11#(11) a[0],a[1],a[2],a[3],ce,clk,in[11];
SRL16 f12#(12) a[0],a[1],a[2],a[3],ce,clk,in[12];
SRL16 f13#(13) a[0],a[1],a[2],a[3],ce,clk,in[13];
SRL16 f14#(14) a[0],a[1],a[2],a[3],ce,clk,in[14];

rreg #(15) rr(clk, ce, t, out);
endmodule
// SRL-based delay, 1-bit width
// SRL16 f1#(1) a = D - 1;
// module delay1(clk, ce, in, out);
// parameter D = 1;
// input clk, ce;
// input in;
// output out;
//local wire [3:0] a = D - 1;
//wire t;
// SRL16 f0#(t,a[0],a[1],a[2],a[3],ce,clk,in);
rreg #(1) rr(clk, ce, t, out);
endmodule
// SRL-based delay, 7-bit width
// SRL16 f0#(7) a = D - 1;
// module delay7(clk, ce, in, out);
// parameter D = 1;
// input clk, ce;
// input in;
// output out[6:0];
//local wire [3:0] a = D - 1;
//wire t;
// SRL16 f0#(t[0],a[0],a[1],a[2],a[3],ce,clk,in[0]);
// SRL16 f1#(t[1],a[0],a[1],a[2],a[3],ce,clk,in[1]);
// SRL16 f2#(t[2],a[0],a[1],a[2],a[3],ce,clk,in[2]);
// SRL16 f3#(t[3],a[0],a[1],a[2],a[3],ce,clk,in[3]);
// SRL16 f4#(t[4],a[0],a[1],a[2],a[3],ce,clk,in[4]);
// SRL16 f5#(t[5],a[0],a[1],a[2],a[3],ce,clk,in[5]);
// SRL16 f6#(t[6],a[0],a[1],a[2],a[3],ce,clk,in[6]);

rreg #(7) rr(clk, ce, t, out);
endmodule

```

```

// Tri-radix counter. (W,N) are the counters' widths and modulus.
// module tri_counter(clk, reset, en, out, cy);
//   parameter W2 = 3;
//   parameter N2 = 8;
//   parameter W1 = 3;
//   parameter N1 = 8;
//   parameter W0 = 4;
//   parameter N0 = 12;
//   input clk, reset;
//   input en;
//   output [W0+W1+N2-1:0] out;
//   output cy;
// endmodule

// Local
// assign cy = cy0 && cy1;
// assign out = {x2,x1,x0};
// counter #(W0,N0) c0(clk, reset, en, x0, cy0);
// counter #(N1,N1) c1(clk, reset, en && cy0, x1, cy1);
// counter #(W2,N2) c2(clk, reset, en && cy0 && cy1, x2, cy2);
// endmodule

// Bi-radix counter. (W1,N1) is msword's width and modulus, (W2,N2)
// is for lsword. Overflows from the lsword serve to increment msword.
// module bi_counter(clk, reset, en, out, cy);
//   parameter W1 = 3;
//   parameter N1 = 8;
//   parameter W0 = 4;
//   parameter N0 = 12;
//   input clk, reset;
//   input en;
//   output [W0+W1-1:0] out;
//   output cy;
// endmodule

// Local
// assign cy = cy0 && cy1;
// assign out = {x1,x0};
// counter #(W0,N0) c0(clk, reset, en, x0, cy0);
// counter #(W1,N1) c1(clk, reset, en && cy0, x1, cy1);
// endmodule

```

lib.v

```

endmodule

// Counter with enable and width/modulus, carry output
// 

module counter(clk, reset, en, out, cy);
Parameter W = 3;
Parameter N = 8;
input clk, reset;
input en;
output [W-1:0] out;
output cy;

reg [W-1:0] out;
reg cy;

always @ (posedge clk)
if (reset) begin
out <= #1 0;
cy <= #1 0;
end else
if (en) begin
out <= #1 cy ? 0 : (out+1);
cy <= #1 (out==(N-2));
end
endmodule

// small synchronous FIFO
// 

module fifo_d(clk, reset,
in, in_req, in_ack,
out, out_req, out_ack);
parameter W = 8;
input clk, reset;
input [W-1:0] in;
input in_req;
output in_ack;
output [W-1:0] out;
input out_req;
output out_ack;
parameter WDATA = 8;
parameter WRDR = 12;
input clk, reset;
input [WDATA-1:0] in;
input in_req;
endmodule

// Memory-based FIFO
// 

module fifo_ram(clk, reset,
in, in_req, in_ack,
out, out_req, out_ack);
parameter WDATA = 8;
parameter WRDR = 12;
input clk, reset;
input [WDATA-1:0] in;
input in_req;
endmodule

```

lib.v

```

output in_ack;
output [(WDATA-1):0] out;
input out_req;
output out_ack;

reg in_ack;

//local
reg [(WADDR-1):0] waddr;
reg [(WADDR-1):0] raddr;

wire [(WADDR-1):0] nraaddr;

wire we = in_req && in_ack;
wire [(WADDR-1):0] w1 = waddr + 1;
wire [(WADDR-1):0] w2 = waddr + 2;
wire [(WADDR-1):0] r1 = raddr + 1;

assign out_ack = (waddr!=raddr);

assign nraaddr = (out_req && out_ack) ? r1 : raddr;

ram_rw #(WDATA,WADDR) r(clk, waddr, in, we, nraaddr, out, 1'b1);

always @ (posedge clk)
if (reset) begin
    waddr <= #1 0;
    raddr <= #1 0;
    in_ack <= #1 0;
end else begin
    in_ack <= #1 ! (w2==raddr) && !(w1==raddr);
    if (!in_req && in_ack)
        waddr <= #1 w1;
end
endmodule

```

interleaver.v

**interleaver.v**

```

// J.83 interleaver. See ITU-T J.83 section B.5.2.
// Copyright 2002 rgb media, inc.
// Peter Monta

module interleaver_sram(clk, reset,
in, in_sync_4_128, in_4avail, in_ack,
out, out_sync_4_128, out_4avail, out_ack);
input clk;
input reset;
input [27:0] in;
input [27:0] out;
input [4:0] mode;
input [6:0] in;
input [6:0] sync_16_128;
input [6:0] in_req, in_ack;
output [6:0] out;
output [6:0] sync_16_128;
input [6:0] out_req;
output out_ack;
wire [27:0] ram_in;
wire ram_in_sync_4_128;
wire ram_in_4avail;
wire ram_in_ack;
wire [27:0] ram_out;
wire ram_out_sync_4_128;
wire ram_out_ack;
wire ram_out_4avail;
wire ram_out_ack;
wire ram_out_4avail;

interleaver_sram iram(clk, reset,
ram_in, ram_in_sync_4_128, ram_in_4avail, ram_in_ack,
ram_out, ram_out_sync_4_128, ram_out_4avail, ram_out_ack);

interleaver_write iwr(clk, reset,
in, in_sync_16_128, in_req, in_ack,
ram_in, ram_in_sync_4_128, ram_in_4avail, ram_in_ack);

interleaver_read ird(clk, reset,
ram_out, ram_out_sync_4_128, ram_out_4avail, ram_out_ack,
out, out_sync_16_128, out_req, out_ack);

endmodule

// Small J.83 interleaver (mode 9 only; 8x16 symbols), suitable for on-FPGA

```

```

// SRAM.

module interleaver_sram(clk, reset,
in, in_sync_4_128, in_4avail, in_ack,
out, out_sync_4_128, out_4avail, out_ack);
input clk;
input reset;
input [27:0] in;
input [27:0] out;
input [4:0] mode;
input [6:0] in;
input [6:0] sync_16_128;
input [6:0] in_req, in_ack;
output [6:0] out;
output [6:0] sync_16_128;
input [6:0] out_req;
output out_ack;
wire [3:0] diff = waddr[10:7] - rc[10:7];
wire [3:0] wp1 = waddr[10:7] + 1;
wire [3:0] rp1 = rc[10:7] + 1;
wire en = 1'b1;
ram_rw #(28,13) ram(clk, {waddr,wp}, in, we, {raddr, rp}, out, en);
reg [3:0] state;
parameter
IDLE = 4'd0,
WR = 4'd1,
WR0 = 4'd2,
WR1 = 4'd3,
WR2 = 4'd4,
WR3 = 4'd5,
RD = 4'd6,
RD0 = 4'd7,
RD1 = 4'd8,
RD2 = 4'd9,
RD3 = 4'd10;

// always # (posedge clk)
// if (in_ack)

```

Appendix -

interleaver.v

```

// $display($time,"interleaver in [%x,%x,%x,%x] in[6:0],in[13:7];in[20:14],in[27:21],out[1:14],out[13:7],out[6:0],waddr,%x waddr,%x rc
%x",in[20:14],in[27:21],out[1:14],out[13:7],out[6:0],waddr,waddr,rc);

// always @ (posedge clk)
//   begin
//     if (rreset) begin
//       waddr <= #1'0;
//       raddr <= #1'0;
//       vap <= #1'0;
//       rap <= #1'0;
//       rc <= #1'0;
//       wfslag <= #1'0;
//       prio <= #1'0;
//       out_sync_4_128 <= #1'0;
//       in_ack <= #1'0;
//       out_ack <= #1'0;
//       state <= #1'IDLE;
//     end
//     else begin
//       wfslag <= #1' (rfif <= '4'd7);
//       rfslag <= #1' (rc[10:7] != waddr[10:7]) && (rp1 != waddr[10:7]);
//     end
//   end
// case (state)
//   IDLE: if (prior) begin
//     if (ln_4avail && wfslag)
//       state <= #1' WR;
//     else if (out_avail && rfslag)
//       state <= #1' RD;
//   end
//   else begin
//     if (out_avail && rfslag)
//       state <= #1' RD;
//     else if (in_avail && wfslag)
//       state <= #1' WR;
//   end
// end

// VR: begin
//   state <= #1' WRO;
//   we <= #1' 1;
//   in_ack <= #1' 1;
//   vap <= #1' 2'd0;
// end
// VR0: if (in_sync_4_128) begin
//   state <= #1' TDE;
//   we <= #1' 0;
//   in_ack <= #1' 0;
//   waddr <= #1' (wp1,7'd0);
// end
// else begin
//   state <= #1' WPL;
//   wap <= #1' 2'd1;
// end
// VR1: if (in_sync_4_128) begin
//   state <= #1' TDE;
//   we <= #1' 0;
//   in_ack <= #1' 0;
//   waddr <= #1' (wp1,7'd0);
// end
// else begin

```

```

state <= #1'WB2;
waddr <= #1'2'd2;
end

WR2: state <= #1 IDLE;
we <= #1 0;
in_ack <= #1 0;
waddr <= #1 'wp1,7'd01;
end else begin
    state <= #1 WR3;
    waddr <= #1 2'd3;
end

WR3: begin
    state <= #1 IDLE;
    we <= #1 0;
    in_ack <= #1 0;
    if (in_sync_4_128)
        waddr <= #1 'wp1,7'd01;
    else
        waddr <= #1 waddr + 1;
end

RD: begin
    state <= #1 RD0;
    out_ack <= #1 1;
    out_sync_4_128 <= #1 (rc[6:0]==7'd0) ;
    rap <= #1 2'd0;
end

RD0: begin
    state <= #1 RD1;
    rap <= #1 2'd1;
    out_sync_4_128 <= #1 0;
end

RD1: begin
    state <= #1 RD2;
    rap <= #1 2'd2;
end

RD2: begin
    state <= #1 RD3;
    rap <= #1 2'd3;
    out_sync_4_128 <= #1 (rc[6:0]==7'd127);
    //                               ? 11'b011100000001 :
    //                               ? 11'b111100000001;
    delta <= #1 (raddr[2:0]==3'd1);
    out_sync_4_128 <= #1 0;
end

RD3: begin
    state <= #1 IDLE;
    rap <= #1 2'd0;
    out_ack <= #1 0;
    raddr <= #1 raddr + delta;
    rc <= #1 rc + 1;
    out_sync_4_128 <= #1 0;
end

default: state <= #1 IDLE;
endcase
endmodule
//
```

## interleaver.v

```

// write bursts of 16 symbols to RAM.

module interleaver_write(clk, reset,
in, in_sync, in_req, in_ack,
ram, ram_sync, ram_4avail, ram_ack);
input clk, reset;
input [6:0] in;
input in_sync;
input in_req;
output in_ack;
output [27:0] ram;
output ram_sync;
output ram_4avail;
input ram_ack;
reg in_ack;
reg ram_4avail;

local
reg [11:0] s;
reg [3:0] waddr, raddr;
wire we1, we2, we3;

// always @ (posedge clk)
// if (ram_ack)
// $display($time,"waddr %d raddr %d sending (%x %x %x
%x)", waddr, raddr, ram[27:21], ram[20:14], ram[6:0]);
// always @ (posedge clk)
// if (in_req && in_ack)
// $display($time,"iwrite input: s %d, in_sync %d, in
%x, s,in_sync,in);
ram #(7,4) r0(clk, waddr, in, we0, raddr, ram[27:21]);
ram #(7,4) r1(clk, waddr, in, we1, raddr, ram[20:14]);
ram #(7,4) r2(clk, waddr, in, we2, raddr, ram[13:7]);
ram #(7,4) r3(clk, waddr, in, we3, raddr, ram[6:0]);
ram #(1,4) rsync(clk, waddr, in_sync, we3, raddr, ram_sync);

assign we0 = (s==2'd0);
assign we1 = (s==2'd1);
assign we2 = (s==2'd2);
assign we3 = (s==2'd3);

wire [3:0] wp1 = waddr + 1;
wire [3:0] wp2 = waddr + 2;

always @ (posedge clk)
if (reset) begin
s <= #1 0;
waddr <= #1 0;
raddr <= #1 0;
ram_4avail <= #1 0;
in_ack <= #1 0;
end

```

```

end else begin
in_ack <= #1 (wp1==raddr) && (wp2!=raddr);
if (in_req && in_ack) begin
s <= #1 in_sync ? 0 : (s+1);
if ((s==2'd3) || in_sync)
waddr <= #1 waddr + 1;
end
ram_4avail <= #1 ((waddr-raddr) >= 4'd6);
raddr <= #1 raddr + 1;
end
endmodule

// Read bursts of 16 symbols from RAM.
//
module interleaver_read(clk, reset,
ram, ram_sync, ram_4avail, ram_ack,
out, out_sync, out_req, out_sack);
input clk, reset;
input [27:0] ram;
input ram_sync;
output ram_4avail;
input ram_ack;
output [6:0] out;
output out_sync;
input out_req;
output out_ack;
reg ram_4avail;
reg out_ack;

//local
reg [1:0] s;
reg [3:0] waddr, raddr;
wire [27:0] r;
wire r_sync;
wire we = ram_ack;
reg ram_4avail;
reg out_ack;

// always @ (posedge clk)
// if (ram_ack)
// $display($time,"raddr %d raddr %d receiving (%x %x %x
%x)", waddr, raddr, ram[27:21], ram[20:14], ram[6:0]);
assign out = (s==0) ? r[27:21] :

```

interleaver.v

```

{s==1} ? r[20:14] :
{s==2} ? r[13:7] ;
r[6:0];

assign out_sync = r_sync && (s==1);

always @ (posedge clk)
if (reset) begin
    s <= #1 0;
    waddr <= #1 0;
    raddr <= #1 0;
    ram_avail <= #1 0;
    out_ack <= #1 0;
end else begin
    ram_avail <= #1 (diff <= 4'd10);
    if (ram_ack)
        waddr <= #1 waddr + 1;
    out_ack <= #1 (raddr==waddr) && (rp1!=waddr) && (rp2!=waddr);
    if (out_req && out_ack) begin
        s <= #1 s + 1;
        if (s==2'd3)
            raddr <= #1 raddr + 1;
    end
end
endmodule

```

Appendix -

## rs\_encoder.v

**rs\_encoder.v**

```

// (128,122) GF(2^7) extended Reed-Solomon encoder; one symbol
// per clock. See ITU-T J.83 section B.5.1.
// Copyright 2002 rgb media, inc.
// Peter Monta
// 

module rs_encoder(clk, reset,
in, in_sync_16, in_req, in_ack,
out, out_sync_16_128, out_req, out_ack);
input clk;
input reset;
input [6:0] in;
input in_sync_16;
input in_req;
output in_ack;
output [6:0] out;
output out_sync_16_128;
input out_req;
output out_ack;
reg [6:0] out;
reg out_sync_16_128;
//local
wire [6:0] p0, p1, p2, p3, p4, p5;
reg [6:0] c;
reg [2:0] mstate;
reg c121;
reg c4;
reg first;
wire flush = mstate[1] || mstate[2];
wire en = flush ? out_req : (in_req && in_ack);
assign in_ack = out_req && !flush;
assign out_ack = in_req || !flush;
wire [6:0] f;
assign f = in & (first ? 0 : p0);
wire [6:0] g_a15 = {
f[3]^f[5]^f[6],
f[2]^f[4]^f[5]^f[6],
f[1]^f[3]^f[4]^f[5],
f[0]^f[2]^f[3]^f[4],
f[1]^f[6] };
wire [6:0] g_a19 = {
f[0]^f[1]^f[4],
f[0]^f[3]^f[6],
f[2]^f[5],
f[1]^f[4],
f[1]^f[3]^f[4],
f[0]^f[2]^f[3]^f[6],
f[1]^f[2]^f[5] };
wire [6:0] g_a16 = {
f[4]^f[3]^f[4],
f[0]^f[2]^f[3]^f[6],
f[1]^f[2]^f[5],
f[1]^f[4],
f[0]^f[1]^f[4],
f[0]^f[1]^f[4]^f[6],
f[0]^f[3]^f[5]^f[6],
f[2]^f[4]^f[5] };
wire [6:0] g_a52 = {
f[1]^f[2]^f[3]^f[4]^f[6],
f[0]^f[1]^f[2]^f[5]^f[6],
f[0]^f[1]^f[2]^f[4]^f[5],
f[0]^f[1]^f[3]^f[4]^f[6],
f[0]^f[1]^f[4]^f[5]^f[6],
f[2]^f[3]^f[4]^f[5]^f[6] };
wire [6:0] x = first ? 7'b0 : p5;
wire [6:0] g_a6 = {
x[0]^x[4],
x[3]^x[6],
x[2]^x[5]^x[6],
x[1]^x[4]^x[5],
x[3],
x[2]^x[6],
x[1]^x[5] };
wire [6:0] np0, np1, np2, np3, np4, np5;
assign np0 = first ? g_a52 : mstate[0] ? p1 ^ g_a52 : p1;
assign np1 = first ? g_a116 : mstate[0] ? p2 ^ g_a116 : p2;
assign np2 = first ? g_a115 : mstate[0] ? p3 ^ g_a115 : p3;
assign np3 = first ? g_a61 : mstate[0] ? p4 ^ g_a61 : p4;
assign np4 = first ? g_a15 : mstate[0] ? p5 ^ g_a15 : p5;
assign np5 = g_a6 ^ (mstate[0] ? in : p0);
delay7 #(15) sr0(c1k, en, np0, p0);

```

## rs\_encoder.v

```

delay7 #(15) sr1{clk, en, np1, p1};
delay7 #(15) sr2{clk, en, np2, p2};
delay7 #(15) sr3{clk, en, np3, p3};
delay7 #(15) sr4{clk, en, np4, p4};
delay7 #(15) sr5{clk, en, np5, p5};

parameter
  SYS=3'b001,
  RS1=3'b010,
  RS2=3'b100;

always @* mstate or in or p0 or p5)
  case (mstate)
    SYS: out = in;
    RS1: out = p0;
    RS2: out = p5;
    default: out = 7'bxx;
  endcase

  reg [3:0] p;
  reg p15;

  always @ (posedge clk)
  ///
  //  if (en)
  //    $display($time,"rs_encoder: mstate %b flush %d in %x out %x cl21 %d
  c4 #d first #d c #d p #d p15 #d",mstate,flush,in,out,cl21,c4,first,c,p,p15);

  always @ (posedge clk)
  if (reset) begin
    mstate <= #1 SYS;
    cl21 <= #1 0;
    c4 <= #1 0;
    c <= #1 0;
    p <= #1 0;
    p15 <= #1 0;
    out_sync_16_128 <= #1 0;
    first <= #1 1;
  end else begin
    if (en) begin
      p <= #1 (in_sync_16& (!Flush)) ? 0 : (p + 1);
      p15 <= #1 (p==4'd14) ? 0 : (p == 4'd14);
      out_sync_16_128 <= #1 (p==4'd14) && (mstate==RS2);
      if (p15) begin
        first <= #1 (mstate==RS2);
        cl21 <= #1 c==7'd20;
        c4 <= #1 c==7'd3;
      end
    end
  end
endmodule

```

transpose.v

**transpose.v**

```

// Transpose from packets to channel-groups. Each group of 16 incoming 18-
byte TS packets goes out as 188 groups of 16 bytes in channel-sequential order.
// Copyright 2002 rgb media, inc.
// Peter Monta
// 

module transpose(clk, reset,
in, in_sync_16_188, in_req, in_ack,
out, out_sync_16_188, out_req, out_ack);
input clk;
input reset;
input [7:0] in;
input in_sync_16_188;
input in_req;
output in_ack;

output [7:0] out;
output out_sync_16_188;
input out_req;
output out_ack;

//local
wire [11:0] addr1, addr2;
wire addr1_cy, addr2_cy;
reg p1, p2;
reg bank;
reg out_sync_16_188;
wire we;
wire en, en2;

// ram_rw #(8,13) ram(clk, (bank,addr1), in, we,
// (!bank,addr2[3:0],addr2[11:4]), out, en);
// fixme: the above does not infer a block ram
ram_rw #(8,13) ram(clk, (bank,addr1), in, we,
(!bank,addr2[3:0],addr2[11:4]), out, 1'b1);

assign in_ack = p1;
assign we = in_req && in_ack;

bi_counter #(4,16,8,188) bco(clk, reset,(in_sync_16_188&we), we, addr1,
addr1_cy);
bi_counter #(8,188,4,16) bac(clk, reset, p2&en2, addr2, addr2_cy);

pipe_control_1 pc(clk, reset, p2, en2, out_ack, out_req, en);

always @ (posedge clk)
if (reset) begin
p1 <= #1 1;
p2 <= #1 0;
end

```

width\_7\_to\_14.v

**Width\_7\_to\_14.v**

```

// Change data width from 7 bits to 14 bits. The first input word
// appears on the msword of the output, and an input word with
// sync asserted will result in the next input word appearing
// on the output's msword.
// Copyright 2002 rgb media, inc.
// Peter Monta

module width_7_to_14(input clk, reset,
                      input [6:0] in,
                      input in_sync, in_req, in_ack,
                      output [6:0] out,
                      output out_sync, out_req, out_ack);
    input v;
    input [6:0] x;
    reg nv, in_ack, out_ack, en;
    reg [6:0] out;
    assign out = (x,in);
    assign out_sync = in_sync;
    always @ (in_sync or in_req or out_req or v)
        case ({in_req,out_req,v})
            3'b000: {inv,in_ack,out_ack,en} = 4'b0000;
            3'b001: {inv,in_ack,out_ack,en} = 4'b1000;
            3'b010: {inv,in_ack,out_ack,en} = 4'b0000;
            3'b011: {inv,in_ack,out_ack,en} = 4'b1000;
            3'b100: {inv,in_ack,out_ack,en} = ({in_sync,3'b101});
            3'b101: {inv,in_ack,out_ack,en} = 4'b1000;
            3'b110: {inv,in_ack,out_ack,en} = {!in_sync,3'b101};
            3'b111: {inv,in_ack,out_ack,en} = 4'b0110;
        endcase

        always @ (posedge clk)
            if (reset) begin
                v <= #1 0;
            end
            else begin
                v <= #1 nv;
                if (en)

```

```

        end
        x <= #1 in;
    endmodule

```

## width\_8\_to\_7.v

**width\_8\_to\_7.v**

```

// Change data width from 8 bits to 7 bits (RS symbols).
// Copyright 2002 rgb media, inc.
// Peter Monta
// 

module width_8_to_7(clk, reset,
in, in_sync_16, in_req, in_ack,
out, out_sync_16, out_req, out_ack);
input clk;
input reset;
input [7:0] in;
input in_sync_16;
input in_req;
output in_ack;
output [6:0] out;
output out_sync_16;
input out_req;
output out_ack;
reg out_sync_16;
//local
wire [6:0] s;
wire en;
wire flush;
reg [3:0] c;
reg [2:0] shift;
delay7 #(15) sr(clk, en, in[6:0], s);
barrel_shift_14_7 bs((s,in[7:1]), shift, out);

assign flush = (shift==3'd7);
assign en = flush ? out_req : (in_req && in_ack);
assign in_ack = out_req && !flush;
assign out_ack = in_req || flush;

// always @ (posedge clk)
//   if (en)
//     $display($time, "width8: out %x c %d out_sync %d shift %d flush %d
in_sync %d", out, c, out_sync_16, flush, in_sync_16);

always @ (posedge clk)
if (reset) begin
  c <= #1 0;
  shift <= #1 0;
  out_sync_16 <= #1 0;
end

```

**CLAIMS**

What is claimed is:

1. A multi-channel modulator for modulating a plurality of digital data streams onto a single multi-RF output, characterized by:

encoding means for encoding each of the digital data streams into a set of symbol streams;

inverse FFT (IFFT) processing means for simultaneously converting the plurality of symbol streams into a single digital multi-channel IF stream having the multiple symbol streams modulated onto a set of uniformly spaced carrier frequencies in an intermediate frequency band;

digital-to-analog conversion means for converting the single digital multi-channel IF stream into an analog multi-channel IF stream; and

up-conversion means to frequency-shift the analog multi-channel IF stream to a target frequency band on said single multi-RF output.

2. A multi-channel modulator according to claim 1, characterized in that:

the digital data streams are QAM encoded according to ITU J.83 Annex B.

3. A multi-channel modulator according to claim 2, characterized in that:

the digital data streams are 256-QAM encoded.

4. A multi-channel modulator according to claim 2, characterized in that:

the digital data streams are 64-QAM encoded.

5. A multi-channel modulator according to claim 1, further characterized in that:

pre-IFFT baseband filtering means for shaping the symbol streams.

6. A multi-channel modulator according to claim 1, further characterized in that:

post-IFFT anti-imaging filtering means for filtering the digital multi-channel IF stream to achieve channel separation.

RGB-102PCT

7. A multi-channel modulator according to claim 1, further characterized in that:

post-IFFT combined filtering means for performing the combined equivalent of baseband and anti-imaging filtering.

8. A multi-channel QAM modulator according to claim 1, further characterized in that:

interpolation means for compensating for a difference between a QAM symbol rate and a channel spacing.

9. A multi-channel modulator for modulating a plurality of digital data streams onto a single multi-RF output, characterized by:

encoding means for encoding the digital data streams into a like plurality of symbol streams at a symbol rate;

inverse frequency transform processing means having each symbol stream applied to a specific complex frequency input thereof, said transform processing means producing a time-domain signal representative of the plurality of symbol streams modulated onto a set of uniformly spaced carrier frequencies in an intermediate frequency (IF) band;

post-transform means, producing a filtered time-domain signal, for performing the combined equivalent of baseband filtering, anti-imaging filtering and rate interpolation to compensate for a difference between the symbol rate and a channel spacing;

digital-to-analog conversion means for converting the filtered time-domain signal from digital to analog form; and

up-converter means for frequency shifting the analog time-domain signal into a target frequency band on a multi-RF output.

10. A multi-channel modulator according to claim 9, characterized in that:

the inverse transform processing means perform an inverse FFT (IFFT) function.

11. A multi-channel QAM according to claim 9, further characterized in that:

digital quadrature correction means for pre-correcting for non-ideal behavior of the up-converter means.

RGB-102PCT

12. A multi-channel QAM modulator according to claim 9, further characterized in that:  
digital offset compensation means for pre-compensating for DC offsets in the digital-to-analog converter means and up-converter means.
13. A method for multi-channel QAM modulation of a plurality of digital data streams onto a single multi-RF output, comprising:  
providing a plurality of digital data input streams;  
encoding each of the digital data streams into a set of QAM-encoded streams;  
processing the QAM-encoded streams via an inverse FFT (IFFT) to modulate the plurality of QAM-encoded streams into a single digital multi-channel IF stream encoding the multiple QAM encoded streams onto a set of uniformly spaced carrier frequencies in an intermediate frequency band;  
converting the digital multi-channel IF stream to analog form; and  
frequency-shifting the analog multi-channel IF stream to a target frequency band on said single multi-RF output.
14. A method according to claim 13, further comprising:  
encoding the digital data streams according to ITU J.83 Annex B.
15. A method according to claim 14, wherein:  
the digital data streams are encoded according to 256-QAM.
16. A method according to claim 14, wherein:  
the digital data streams are encoded according to 64-QAM.
17. A method according to claim 13, further comprising:  
post-IFFT filtering the digital multi-channel IF stream in a combined baseband and anti-imaging filter.
18. A method according to claim 13, further comprising:

RGB-102PCT

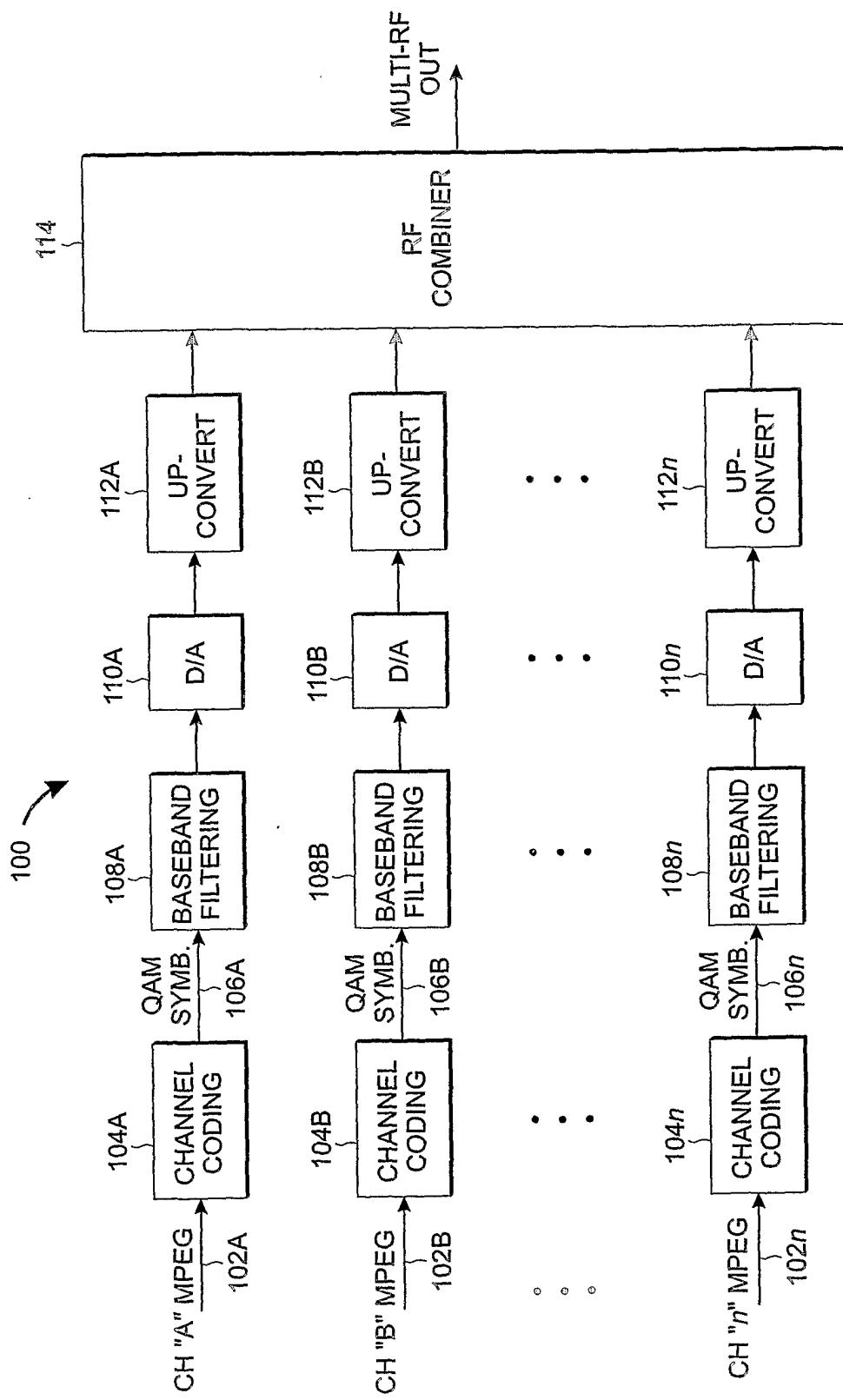
interpolating the digital multi-channel IF stream to compensate for a difference between a QAM symbol rate and a channel spacing.

19. A method according to claim 13, further comprising:

providing digital compensation for non-ideal behavior of the frequency-shifting process.

20. A method according to claim 13, further comprising:

providing digital offset compensation for DC offsets in the digital-to-analog conversion and frequency shifting processes.

**Fig. 1**PRIOR ART

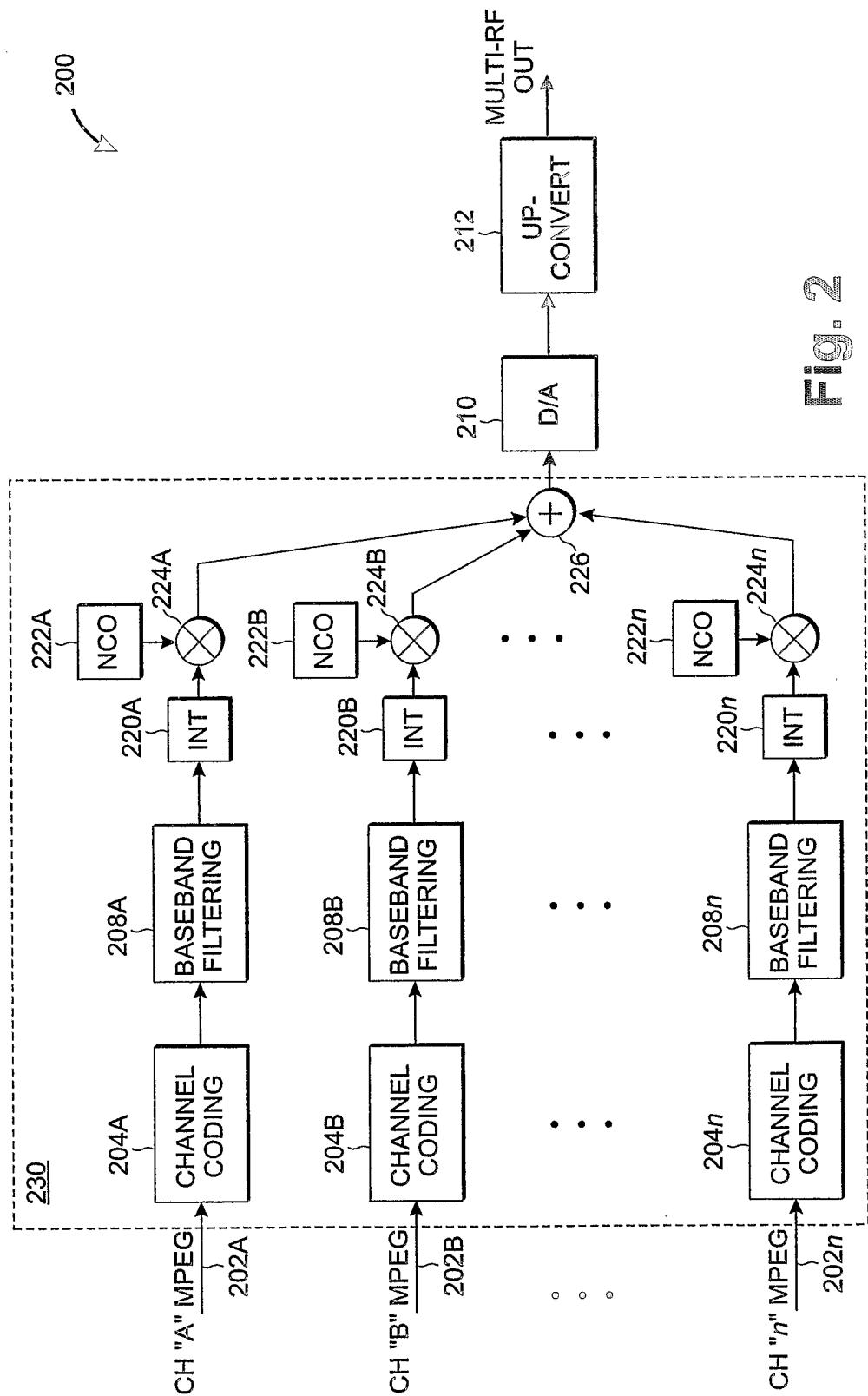


Fig. 2

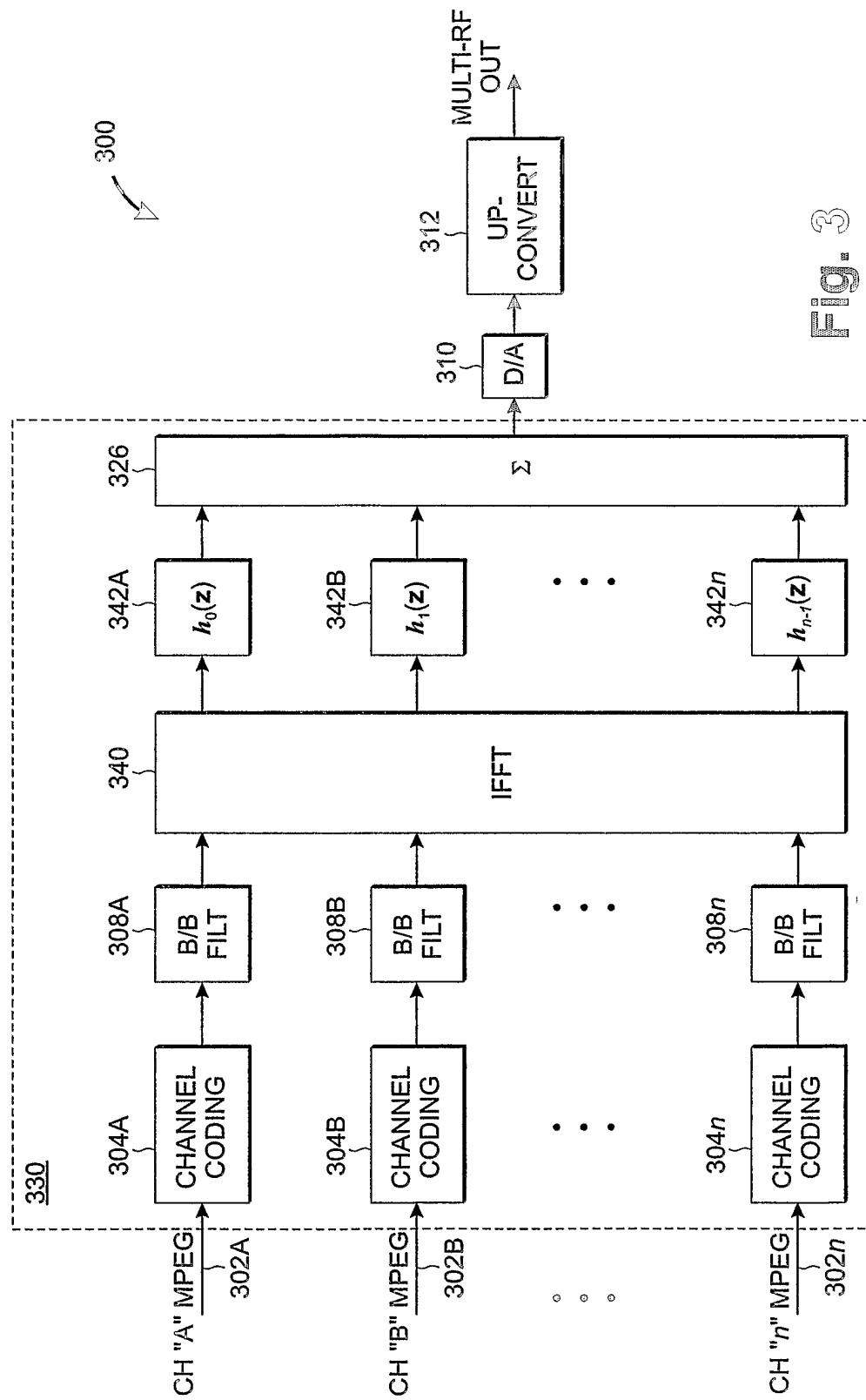
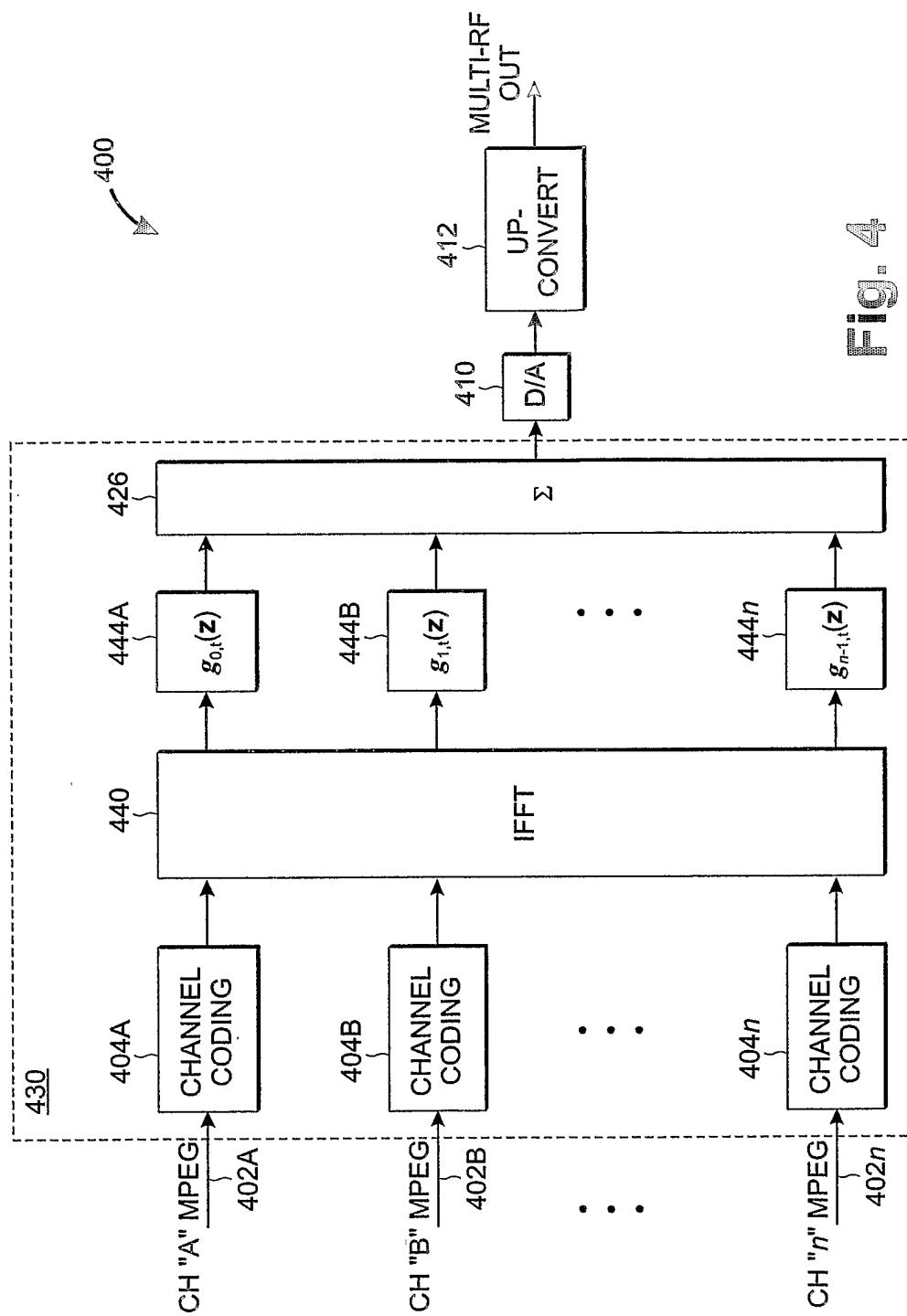


Fig. 3



四  
七

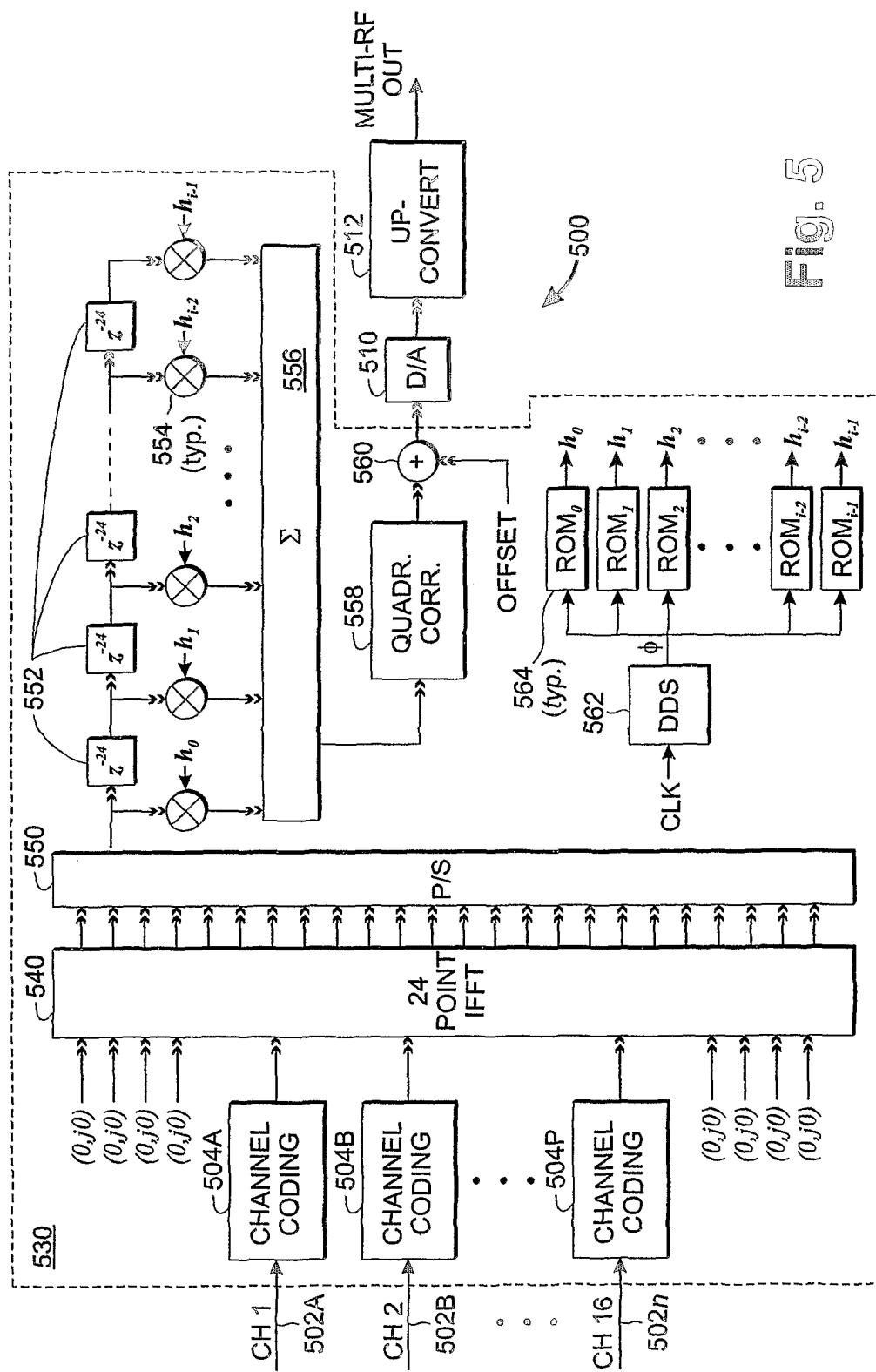


Fig. 5