TI Developer Conference
March 7-9, 2007 • Dallas, TX

Minds in Motion

Code Optimization for Next-Generation Mobile Infrastructures

WiMax and UMTS Physical Layer Challenges and Solutions

Jimmy Zhang
Constantin Bajenaru
Texas Instruments

Acknowledgement

- This presentation is based on the work by Mingjian Yan, Slobodan Jovanovic, Aleksandar Purkovic from Brian Johnson's WiMax engineering team
- The UMTS part of this presentation was not possible without the contribution of Peter Dent and TX Chip Rate FL SW team: Pothukuchi Vijay, Gary Dean, Pragat Chaudhari and Greg Simpson

Agenda

- Technical challenges
- Code optimizations in general
- Examples of code optimization in TI WiMax Functional Library
- RSA and its optimized utilization in TI UMTS Functional Library for HSUPA and HSDPA

Technical Challenges for WiMax PHY Layer Implementation

- Highly mathematical operations with complex numbers, such as FFT/IFFT
- Intensive bit manipulations
- Frequent 2-D grid resource allocations and scheduling

Distinct Features of C64x and C64x+ DSP

- Hardware accelerators such as VCP and RSA

(see backup slides for HW architectures of C6416, C6482, C6487, C6488 and C6455)

- New instructions to reduce CPU cycles
  - GMPY, GMYP4 (Galois field multiply)
  - CMPY (complex multiply two pairs)
  - DOP2, DDPOTPH2
  - SHFL, SHFL3

TI Offers a Powerful Compiler

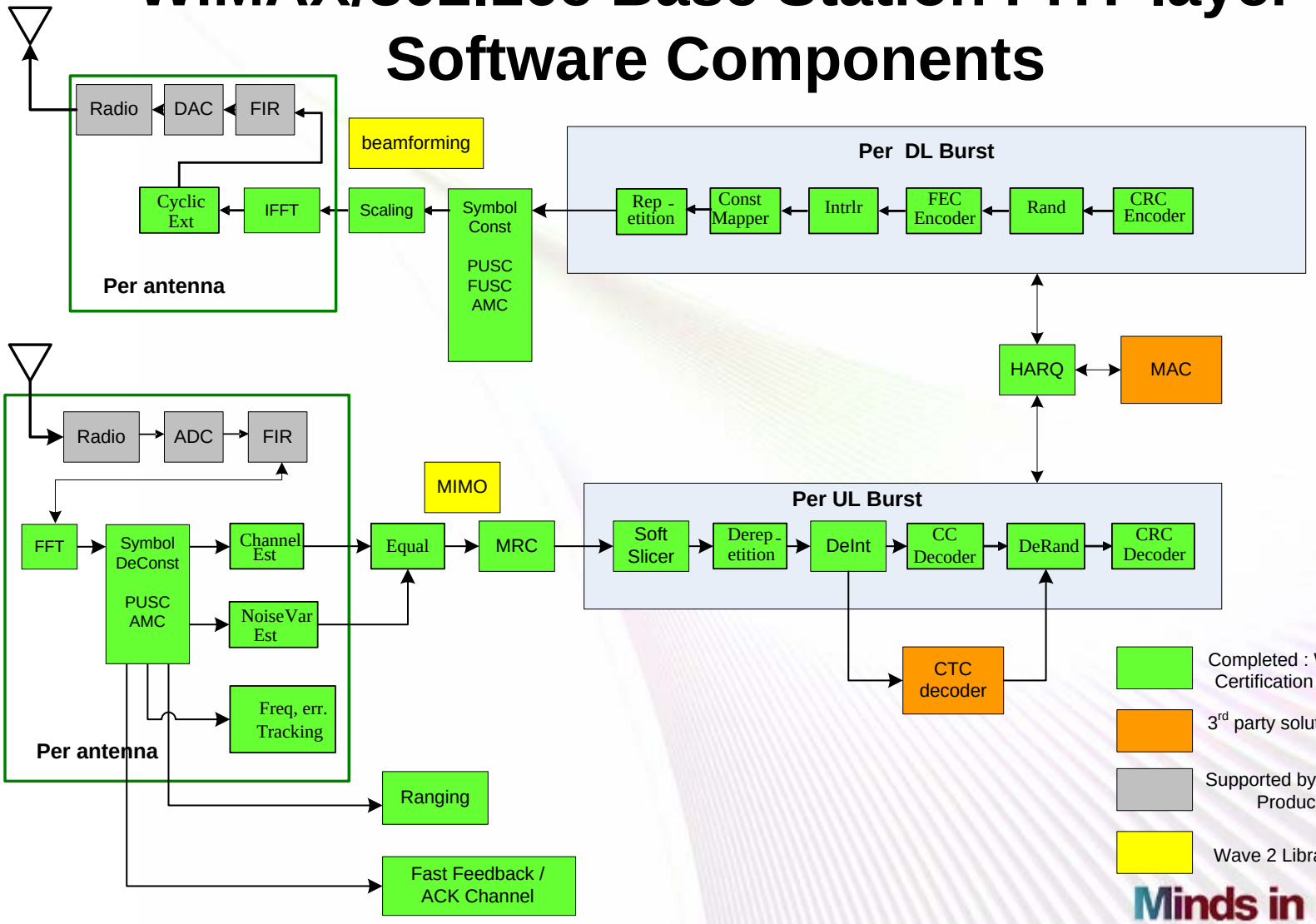
	C/C++	Assembly	Linear Assembly
Instruction Selection	**Automatic**	Manual	Manual
Partitioning	**Automatic**	Manual	Optional
Functional Unit allocation	**Automatic**	Manual	Automatic
Instruction scheduling	**Automatic**	Manual	Automatic
Register allocation	**Automatic**	Manual	Optional
Ease of Use	**Ease**	Hard	In-between
Performance	**Good**	Best	ok

Ref: https://focus.ti.com/seclit/ml/sprp314/sprp314.pdf

Code Optimization in General

- **Design for efficiency, maintenance and reusability**
  - Software layer interfaces and module interfaces
- **Design for optimization**
  - Module profiling
  - System level control code optimization
  - Module level processing code optimization
  - Optimization by balancing memory, MIPS and cache

WiMAX/802.16e Base Station PHY layer Software Components



Minds in Motion

Interleaver and De-Interleaver for Convolutional Coding

Interleaver

Algorithm described in Section 8.4.9.3 of 802.16. This is a two-step block interleaver.

- k -- index of the original data bits
- j -- index of the data bits after the interleaver operations

$$interleavedBits[\ j\ ] = inputBits[\ k\ ]$$

First permutation by Step-1:

$$m = (N/d) * (k \bmod d) + floor(k/d)$$

Second permutation by Step-2:

$$j = s * floor(m/s) + (m+N-floor(d*m/N)) \bmod s$$

- s -- number of bits per constellation point per dimension
- N -- FEC block size
- d – modulo factor (d = 16)

Analyzing the operations

First permutation by Step-1:

$$m = (N/d) * (k \bmod d) + \text{floor}(k/d)$$

No. of rows

Bits in the same column have the same rotation factor

Second permutation by Step-2:

$$j = s * \text{floor}(m/s) + (m+N- \text{floor}(d*m/N)) \bmod s$$

The rotation pattern is applied every s bits

The rotation is applied to the s bits in a circular fashion

Implementing the operations

Given that N is a multiple of d and s for Convolutional Coding in 802.16, we have,

j = s * floor(m/s) + (m + N- (k mod d)) mod s

So the algorithm can be implemented as follows:

1. **Read in 16 bits each time in a row by row fashion**
2. **Write out the 16 bits in a column by column order, but before write, modify the index with the offset as follows.**
3. **Offset calculations**
    1. **If s = 0, offset = 0;**
    2. **If s =1,**
        1. **offset = 0, if (k mod d) mod s = 0**
        2. **offset = 1, if (k/d) mod s = 0**
        3. **offset = -1, if (k/d) mod s = 1**
    3. **If s =2,**
        1. **offset = 0, if (k mod d) mod s = 0**
        2. **offset = 2, if (k/d) mod s = 0, and (k mod d) mod s = 1**
        3. **offset = -1, if (k/d) mod s = 1, and (k mod d) mod s = 1**
        4. **offset = -1 if (k/d) mod s = 2, and (k mod d) mod s = 1**
        5. **offset = 1, if (k/d) mod s = 0, and (k mod d) mod s = 2**
        6. **offset = 1, if (k/d) mod s = 1, and (k mod d) mod s = 2**
        7. **offset = 2, if (k/d) mod s = 2, and (k mod d) mod s = 2**

An example

			InputBits				(s=3, N=96)									
b0	b1	b2	b3	b4	…											j
			InterleavedBits													
b0	b16	b32	b48	b64	b80	b17	b33	b1	b65	b81	b49	b34	b2	b18	…	k

			InputBits Matrix												
b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15
b16	b17	b18	b19	b20	b21	b22	b23	b24	b25	b26	b27	b28	b29	b30	b31
b32	b33	b34	b35	b36	b37	b38	b39	b40	b41	b42	b43	b44	b45	b46	b47
b48	b49	b50	b51	b52	b53	b54	b55	b56	b57	b58	b59	b60	b61	b62	b63
b64	b65	b66	b67	b68	b69	b70	b71	b72	b73	b74	b75	b76	b77	b78	b79
b80	b81	b82	b83	b84	b85	b86	b87	b88	b89	b90	b91	b92	b93	b94	b95
			InterleaveredBits Matrix												
b0	b17	b34	b3	b20	b37	b6	b23	b40	b9	b26	b43	b12	b29	b46	b15
b16	b33	b2	b19	b36	b5	b22	b39	b8	b25	b42	b11	b28	b45	b14	b31
b32	b1	b18	b35	b4	b21	b38	b7	b24	b41	b10	b27	b44	b13	b30	b47
b48	b65	b82	b51	b68	b85	b54	b71	b88	b57	b74	b91	b60	b77	b94	b63
b64	b81	b50	b67	b84	b53	b70	b87	b56	b73	b90	b59	b76	b93	b62	b79
b80	b49	b66	b83	b52	b69	b86	b55	b72	b89	b58	b75	b92	b61	b78	b95

rot = c mod s

			InputBits				(s=3, N=96)									
b0	b1	b2	b3	b4	…											j
			InterleavedBits													
b0	b16	b32	b48	b64	b80	b17	b33	b1	b65	b81	b49	b34	b2	b18	…	k

			InputBits Matrix												
b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15
b16	b17	b18	b19	b20	b21	b22	b23	b24	b25	b26	b27	b28	b29	b30	b31
b32	b33	b34	b35	b36	b37	b38	b39	b40	b41	b42	b43	b44	b45	b46	b47
b48	b49	b50	b51	b52	b53	b54	b55	b56	b57	b58	b59	b60	b61	b62	b63
b64	b65	b66	b67	b68	b69	b70	b71	b72	b73	b74	b75	b76	b77	b78	b79
b80	b81	b82	b83	b84	b85	b86	b87	b88	b89	b90	b91	b92	b93	b94	b95
			InterleaveredBits Matrix												
0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
b0	b17	b34	b3	b20	b37	b6	b23	b40	b9	b26	b43	b12	b29	b46	b15
b16	b33	b2	b19	b36	b5	b22	b39	b8	b25	b42	b11	b28	b45	b14	b31
b32	b1	b18	b35	b4	b21	b38	b7	b24	b41	b10	b27	b44	b13	b30	b47
b48	b65	b82	b51	b68	b85	b54	b71	b88	b57	b74	b91	b60	b77	b94	b63
b64	b81	b50	b67	b84	b53	b70	b87	b56	b73	b90	b59	b76	b93	b62	b79

Further optimize using intrinsic

```
Nrow = (NumCodedBitsPerBlock >> 4); /* N/16 */
  NumInit = 0;

  /* Initialization of 16 indexes of the interleaved
     */
  /* locations for the first 16 bits              */
  IndexBit0 = NumInit; NumInit += Nrow;
  IndexBit1 = NumInit; NumInit += Nrow;
  IndexBit2 = NumInit; NumInit += Nrow;
  IndexBit3 = NumInit; NumInit += Nrow;
  IndexBit4 = NumInit; NumInit += Nrow;
  IndexBit5 = NumInit; NumInit += Nrow;
  IndexBit6 = NumInit; NumInit += Nrow;
  IndexBit7 = NumInit; NumInit += Nrow;
  IndexBit8 = NumInit; NumInit += Nrow;
  IndexBit9 = NumInit; NumInit += Nrow;
  IndexBit10 = NumInit; NumInit += Nrow;
  IndexBit11 = NumInit; NumInit += Nrow;
  IndexBit12 = NumInit; NumInit += Nrow;
  IndexBit13 = NumInit; NumInit += Nrow;
  IndexBit14 = NumInit; NumInit += Nrow;
  IndexBit15 = NumInit;….
```

```
for(k=0; k<NumCodedBitsPerBlock; k+=16)       {
      In0 = _amem8((void * )InputBits);  InputBits+=8;
      In1 = _amem8((void * )InputBits);  InputBits+=8;
      if ( k_div_d_mod_s == 0 ){
        IndexOffset1 = 2;
        IndexOffset2 = 1
       } else if ( k_div_d_mod_s == 1 )
          IndexOffset1 = -1;
    else if ( k_div_d_mod_s == 2 )
          IndexOffset2 = -2;
        j0 = IndexBit0;
        InterleavedBits[j0] = (_loll(In0)) & 0x0FF;  IndexBit0++;
        j1 = IndexBit1 + IndexOffset1;
        InterleavedBits[j1] = (_loll(In0)>>8) & 0x0FF; IndexBit1++;
        j2 = IndexBit2 + IndexOffset2;
        InterleavedBits[j2] = (_loll(In0)>>16) & 0x0FF;   IndexBit2++;
        j3 = IndexBit3;
        InterleavedBits[j3] = (_loll(In0)>>24) & 0x0FF;   IndexBit3++;
        j4 = IndexBit4 + IndexOffset1;
        InterleavedBits[j4] = (_hill(In0)) & 0x0FF; IndexBit4++;
        j5 = IndexBit5 + IndexOffset2;
        InterleavedBits[j5] = (_hill(In0)>>8) & 0x0FF;
```

Minds in Motion

Benchmark

- Cycles = (N/16) * 14 + 60
- Where :
  N = number of input bits

De-interleaver

Performs the inverse operation of the interleaver.

- j -- index of the interleaved data bits
- k -- index of the de-interleaved data bits

deinterleavedBits[k] = interleavedBits[j]

First permutation by Step-1:

m = s * floor(j/s) + (j + floor(d * j/N)) mod s

Second permutation by Step-2:

j = d * m - (N-1)* floor(d*m/N)

- s -- number of bits per constellation point per dimension
- N -- FEC block size
- d – modulo factor (d = 16)

Benchmark

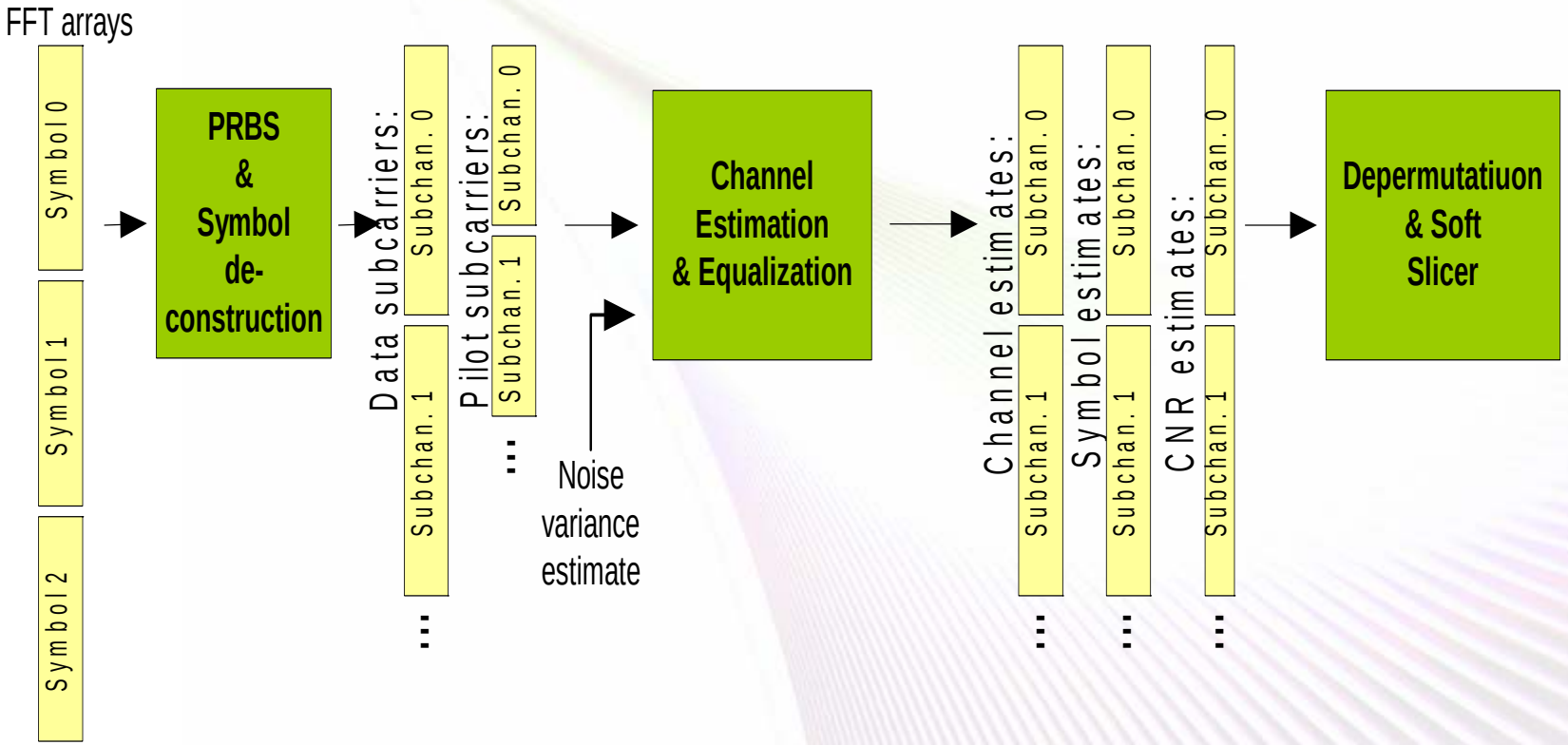
	S		
	1	2	3
Cycles	N + 213	(7 * N)/6 + 230	(7*N)/6 + 440

N = number of input soft bits

- N = number of input soft bit

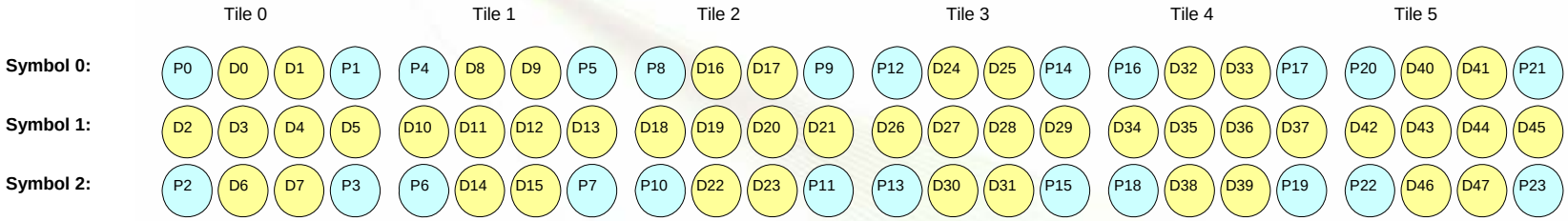
Uplink Channel Estimation and Equalization for PUSC Mode for SISO channels

Channel Estimation and Equalization in the Receiver Path



Data and Pilot Subcarrier Order

One slot:



Little endian:

Data buffer: (48 elements)

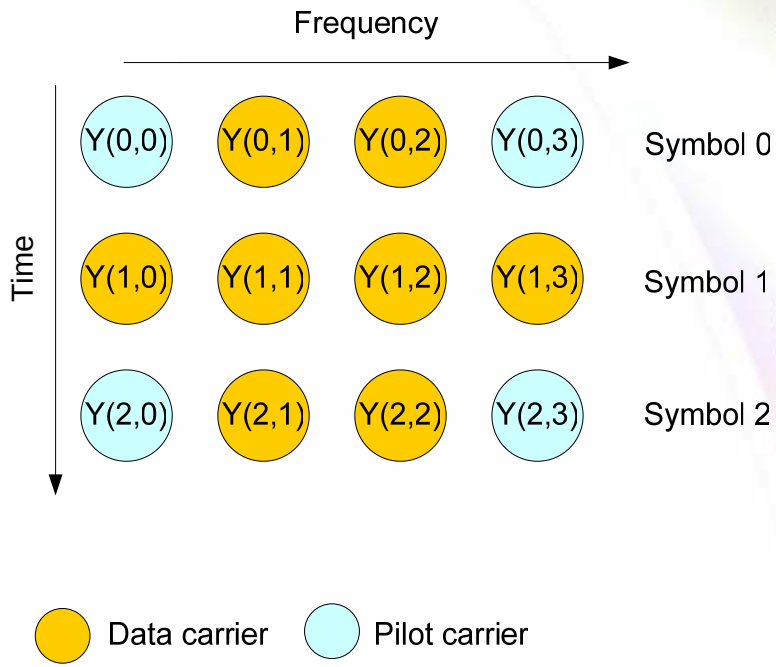
$$Y_D = \left[D0\ D1\ D2\ D3\ D4\ D5\ D6\ D7\ D8\ D9\ D10\ D11\ D12\ D13\ D14\ D15\ \cdots\ D40\ D41\ D42\ D43\ D44\ D45\ D46\ D47 \right]$$

Pilot buffer: (24 elemnts)

$$Y_P = \left[P0\ P1\ P2\ P3\ P4\ P5\ P6\ P7\ \cdots\ P20\ P21\ P22\ P23 \right]$$

Channel Estimation

Tile is a basic building block in the uplink: a tile



Channel is estimated at pilot positions:

$$\hat{H}(0,0) = Y(0,0) / X_P$$

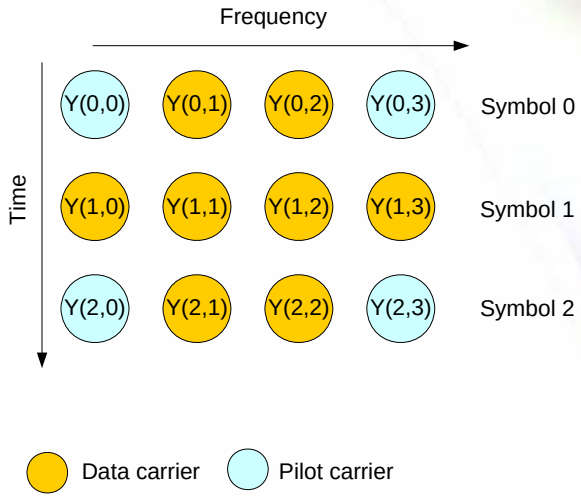
$$\hat{H}(0,3) = Y(0,3) / X_P$$

$$\hat{H}(2,0) = Y(2,0) / X_P$$

$$\hat{H}(2,3) = Y(2,3) / X_P$$

$$\hat{H}'(t,f) = \hat{H}(t,f) \cdot X_P = Y(t,f)$$

Channel Estimation by Interpolation



$$\hat{H}(1,0) = \frac{1}{2}\hat{H}(0,0) + \frac{1}{2}\hat{H}(2,0)$$

$$\hat{H}(1,3) = \frac{1}{2}\hat{H}(0,3) + \frac{1}{2}\hat{H}(2,3)$$

$$\hat{H}(0,1) = \frac{2}{3}\hat{H}(0,0) + \frac{1}{3}\hat{H}(0,3)$$

$$\hat{H}(2,1) = \frac{2}{3}\hat{H}(2,0) + \frac{1}{3}\hat{H}(2,3)$$

$$\hat{H}(0,2) = \frac{1}{3}\hat{H}(0,0) + \frac{2}{3}\hat{H}(0,3)$$

$$\hat{H}(2,2) = \frac{1}{3}\hat{H}(2,0) + \frac{2}{3}\hat{H}(2,3)$$

$$\hat{H}(1,1) = \frac{2}{6}\hat{H}(0,0) + \frac{2}{6}\hat{H}(2,0) + \frac{1}{6}\hat{H}(0,3) + \frac{1}{6}\hat{H}(2,3)$$

$$\hat{H}(1,2) = \frac{1}{6}\hat{H}(0,0) + \frac{1}{6}\hat{H}(2,0) + \frac{2}{6}\hat{H}(0,3) + \frac{2}{6}\hat{H}(2,3)$$

Implementation Estimation Using Intrinsic

- for(i = 0; i < numOfPilotSubCars; i += NUM_PILOTS_PER_TILE) {
  - temp1 = _amem8(&pilotSubCars[i]); temp2 = _amem8(&pilotSubCars[i+2]);
  - pilot0 = _loll(temp1); pilot1 = _hill(temp1);pilot2 = _loll(temp2); pilot3 = _hill(temp2);
  - h2 = _avg2(pilot0,pilot2); h5 = _avg2(pilot1,pilot3);
  - temp1 = _smpy2ll(pilot0, 0x55555555); temp2 = _smpy2ll(pilot1, 0x2AAB2AAB);
  - stempEven = _packh2(_sadd(_hill(temp1),_hill(temp2))+0x8000,_sadd(_loll(temp1),_loll(temp2))+0x8000);
  - temp1 = _smpy2ll(pilot0, 0x2AAB2AAB); temp2 = _smpy2ll(pilot1, 0x55555555);
  - stempOdd = _packh2(_sadd(_hill(temp1),_hill(temp2))+0x8000,_sadd(_loll(temp1),_loll(temp2))+0x8000);
  - _amem8(&chanEstPtr[i*2]) = _itoll(stempOdd,stempEven);
  -
  - temp1 = _smpy2ll(pilot2, 0x55555555); temp2 = _smpy2ll(pilot3, 0x2AAB2AAB);
  - stempEven = _packh2(_sadd(_hill(temp1),_hill(temp2))+0x8000,_sadd(_loll(temp1),_loll(temp2))+0x8000);
  - temp1 = _smpy2ll(pilot2, 0x2AAB2AAB); temp2 = _smpy2ll(pilot3, 0x55555555);
  - stempOdd = _packh2(_sadd(_hill(temp1),_hill(temp2))+0x8000,_sadd(_loll(temp1),_loll(temp2))+0x8000);
  - _amem8(&chanEstPtr[i*2+6]) = _itoll(stempOdd,stempEven);
  -
  - temp0 = _smpy2ll(pilot0, 0x2AAB2AAB); temp1 = _smpy2ll(pilot2, 0x2AAB2AAB);
  - temp2 = _smpy2ll(pilot1, 0x15551555); temp3 = _smpy2ll(pilot3, 0x15551555);
  - tempr = _sadd(_hill(temp0),_hill(temp1)); tempr = _sadd(tempr,_hill(temp2));
  - tempr = _sadd(tempr,_hill(temp3)); tempi = _sadd(_loll(temp0),_loll(temp1));
  - rempi = _sadd(tempi,_loll(temp2)); tempi = _sadd(tempi,_loll(temp3));
  - stempOdd = _packh2(tempr+0x8000, tempi+0x8000);
  - _amem8(&chanEstPtr[i*2+2]) = _itoll(stempOdd,h2);
  -
  - temp0 = _smpy2ll(pilot0, 0x15551555); temp2 = _smpy2ll(pilot1, 0x2AAB2AAB);
  - temp1 = _smpy2ll(pilot2, 0x15551555); temp3 = _smpy2ll(pilot3, 0x2AAB2AAB);
  - tempr = _sadd(_hill(temp0),_hill(temp1)); tempi = _sadd(_loll(temp0),_loll(temp1));
  - tempr = _sadd(tempr,_hill(temp2)); tempi = _sadd(tempi,_loll(temp2));
  - tempr = _sadd(tempr,_hill(temp3)); tempi = _sadd(tempi,_loll(temp3));
  - stempEven = _packh2(tempr+0x8000, tempi+0x8000);
  - _amem8(&chanEstPtr[i*2+4]) = _itoll(h5,stempEven);
- }

Benchmark

Cycles = (N/4) * 10 + 50

Where :

N = number of input bits

Channel Equalization

$$\hat{W}_{t,f} = \frac{\hat{H}^{*}_{t,f}}{\left|\hat{H}_{t,f}\right|^{2}} = \frac{\hat{H}'^{*}_{t,f} \cdot X_P}{\left|\hat{H}'_{t,f}\right|^{2}} \quad \text{for } t = 0,...,2,\ f = 0,...,3$$

$$\hat{X}_{t,f} = \hat{W}_{t,f} \cdot Y_{t,f} = \frac{\hat{H}'^{*}_{t,f} \cdot Y_{t,f} \cdot X_P}{\left|\hat{H}'_{t,f}\right|^{2} + C} \quad \text{for } t = 0,...,2, \quad f = 0,...,3$$

$$\hat{X} = \hat{W} \cdot Y = \frac{\hat{H}'^{*} \cdot Y \cdot X_P}{\hat{H}' \cdot \hat{H}'^{*} + C} = \hat{H}'^{*} \cdot Y \cdot X_P \cdot b \cdot 2^{\text{expn}}$$

Implementation Equalization Using Intrinsic

- for(k = 0; k < numDataSubCars; k+=2){

```
    temph = _amem8(&chanEstPtr[k]);  tempy = _amem8(&dataSubCarsPtr[k]);
    y0 = _loll(tempy); y1 = _hill(tempy); h0 = _loll(temph);h1 = _hill(temph);
    hsqrd = _abs(_dotp2(h0,h0));denom = _sadd(hsqrd, eqDenomCnst);
    normal =  _norm(denom);
    a   = (denom << normal) & 0x7fff0000;
    b = 0x80000000;                    /* dividend = 1 */
    for(i = 15; i > 0; i--)  b = _subc(b,a);
    b = b & 0x7fff;
    expn = (Int32) normal - 15;
    xr = _dotp2(y0, h0);   xi = _dotpn2(_rotl(y0,16), h0);
    xr = _mpylir(b, xr);   xi = _mpylir(b, xi);
    xr = _mpylir(eqOutRms, xr);          xi = _mpylir(eqOutRms, xi);
    x_even = _packh2(_sadd(_sshvl(xr, expn+16), 0x8000), _sadd(_sshvl(xi, expn+16), 0x8000));
    hsqrd = _abs(_dotp2(h1,h1)); denom = _sadd(hsqrd, eqDenomCnst);
    normal =  _norm(denom);
    a   = (denom << normal) & 0x7fff0000;
    b = 0x80000000;                    /* dividend = 1 */
    for(i = 15; i > 0; i--)  b = _subc(b,a);               /* divide */
    b = b & 0x7fff;
    expn = (Int32) normal - 15;
    xr = _dotp2(y1, h1); xi = _dotpn2(_rotl(y1,16), h1);
    xr = _mpylir(b, xr); xi = _mpylir(b, xi);
    xr = _mpylir(eqOutRms, xr); xi = _mpylir(eqOutRms, xi);
    x_odd = _packh2(_sadd(_sshvl(xr, expn+16), 0x8000), _sadd(_sshvl(xi, expn+16), 0x8000));
    _amem8(&symbEstPtr[k]) = _itoll(x_odd,x_even);
    _amem4(&cnrEstPtr[k]) = _packh2(_sadd(cnr_odd, 0x8000), _sadd(cnr_even, 0x8000));
}
```

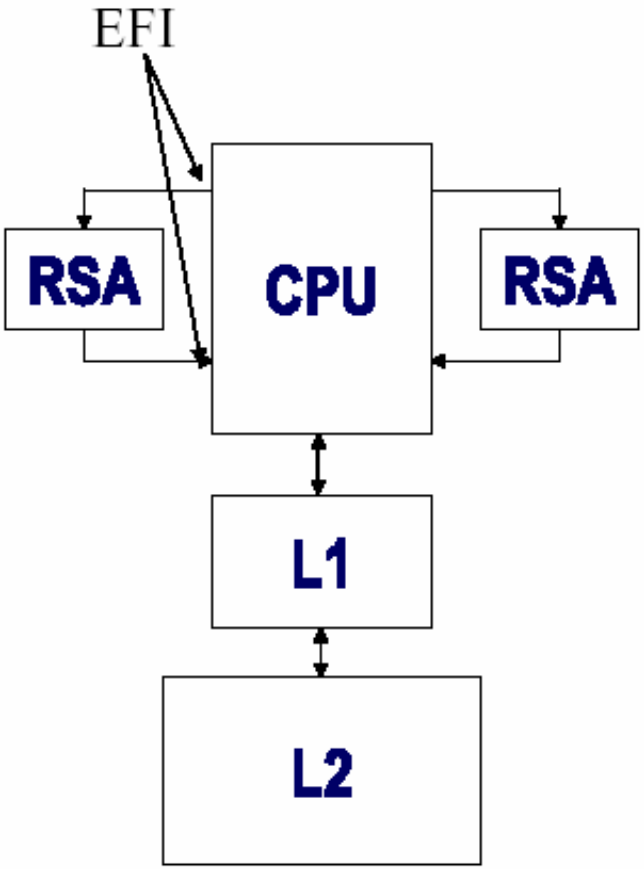
Benchmark

Cycles = *17*N/2+90*

Where:

N = number of input bits

Extended Functional Interface

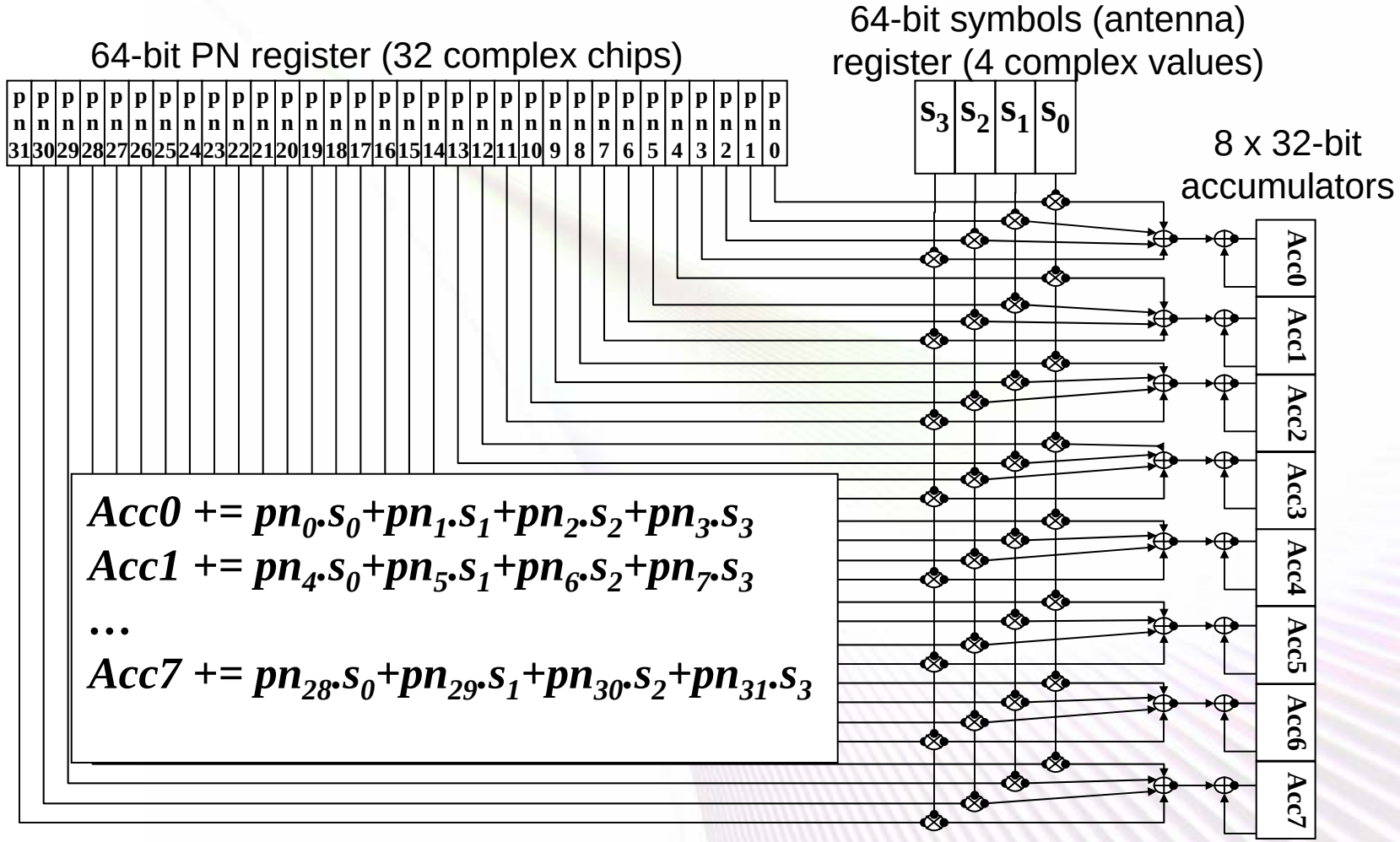


- All EFI instructions execute in a single cycle
- EFI Instructions supported:
  - 64 bit wide data bus
  - EFSW / EFSDW .L src_reg, RSA_CMD
    - Sends single or double word data from CPU to RSA
    - RSA_CMD is a 5 bit opcode field used to convey the RSA instruction
  - EFCMD .M/.S RSA_CMD
    - Sends commands to the RSA e.g. write RSA results into the output FIFO
    - RSA_CMD is a 10 bit opcode field
  - EFRDW / EFRW .S dst_reg
    - Reads data from the RSA output FIFO into a 32/64 bit CPU register
    - For every EFCMD write FIFO operation sequence there must be a corresponding EFRDW/EFRW sequence

RSA Modes

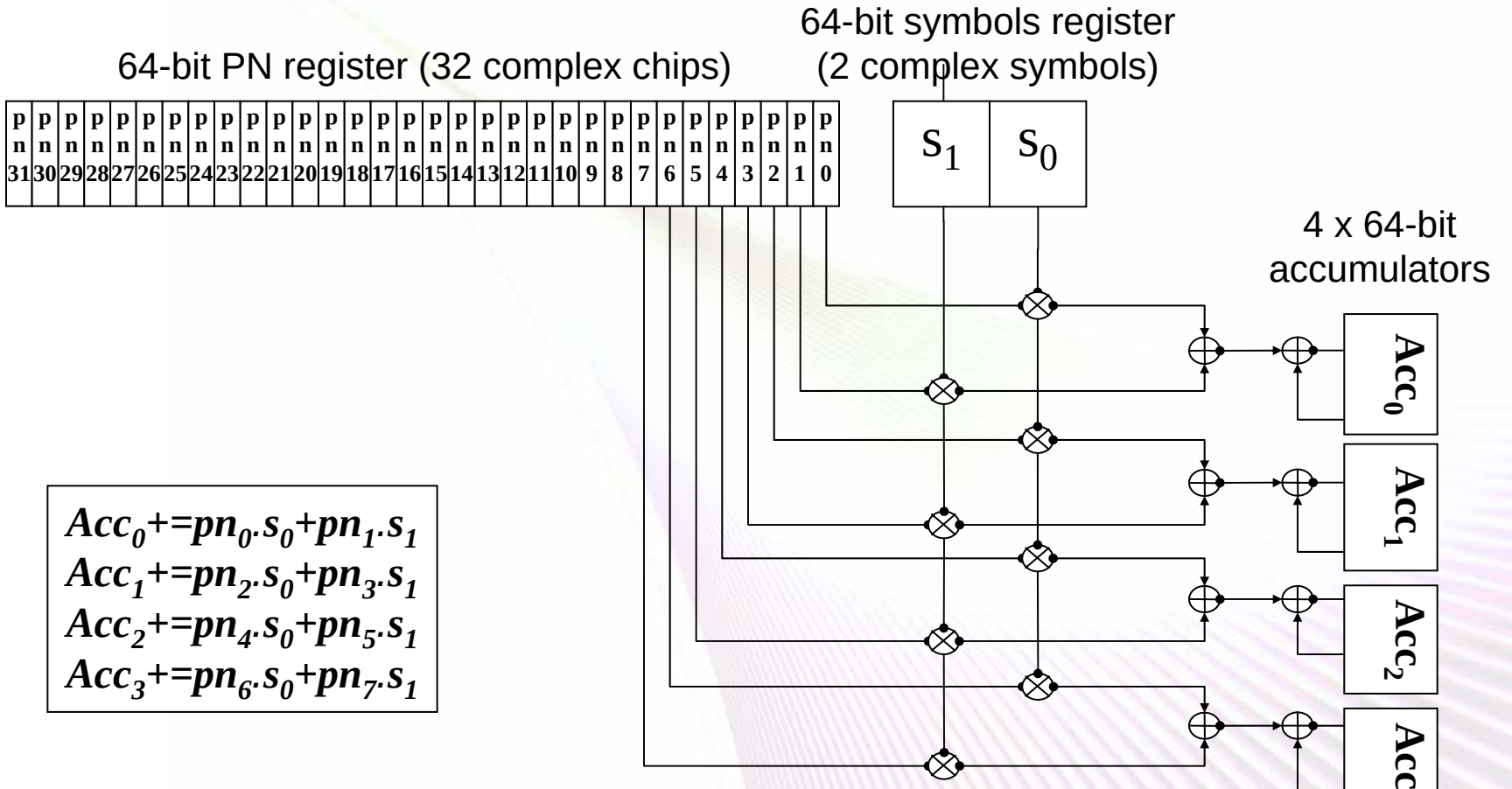
- **Precision**
  - 8-bit mode: 8-bit input and 16-bit real and imag. output data
  - 16-bit mode: 16-bit input and 32-bit real and imag. output data
- **Modes**
  - SPREAD
    - Downlink spreading
    - Finger despreading (Octal Rake)
    - Signatures aggregation (HSUPA)
    - QAM modulation
    - Hadamard Transform
  - SEARCH
    - Random access preamble search
    - Path monitoring
    - Finger despreading

8-bit Spread Mode (Octal Rake)



Minds in Motion

16-bit SPREAD Mode



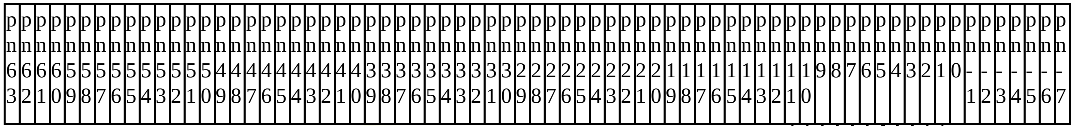
$$Acc_0 += pn_0.s_0 + pn_1.s_1$$
$$Acc_1 += pn_2.s_0 + pn_3.s_1$$
$$Acc_2 += pn_4.s_0 + pn_5.s_1$$
$$Acc_3 += pn_6.s_0 + pn_7.s_1$$

Minds in Motion

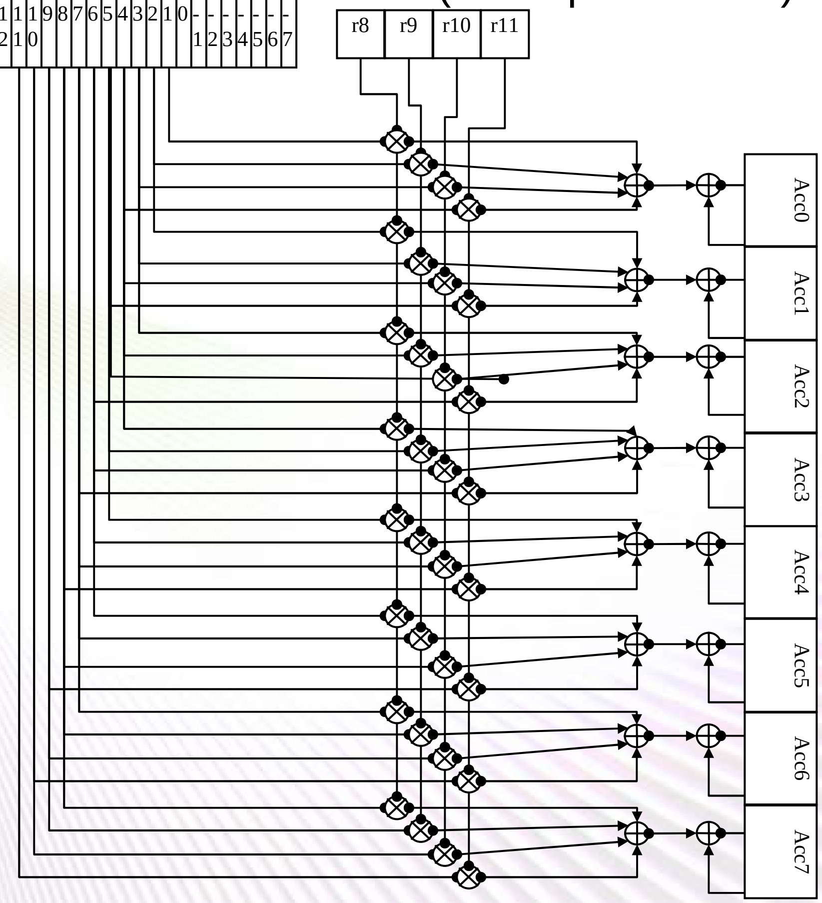
TI | Developer Conference

8-bit Search Mode

64-bit antenna register
(4 complex values)



128-bit PN register (64 complex chips)



$Acc0 \mathrel{+}= pn_1.r_8 + pn_2.r_9 + pn_3.r_{10} + pn_4.r_{11}$

$Acc1 \mathrel{+}= pn_2.r_8 + pn_3.r_9 + pn_4.r_{10} + pn_5.r_{11}$

…

$Acc7 \mathrel{+}= pn_8.r_8 + pn_9.r_9 + pn_{10}.r_{10} + pn_{11}.r_{11}$

Minds in Motion

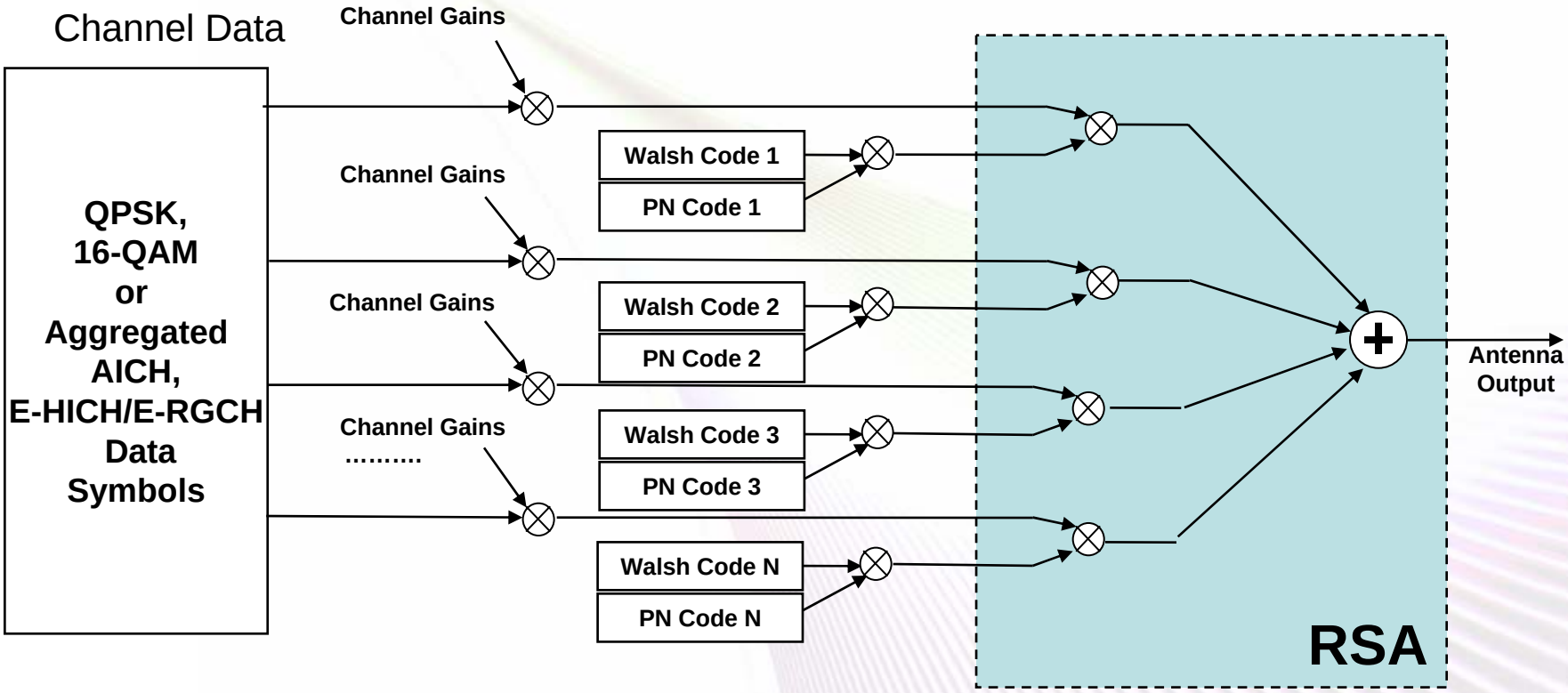
RSA Peak Performance

- Spread and Aggregate – 4 chips for 8 channels in 5 cycles (SPREAD 16-bit mode)
- Finger Despreading - 4-chip 8 aligned fingers in 2 cycles (SPREAD 8-bit mode)
- Path Monitor – 8 64-chip correlations in 18 cycles (SEARCH 8-bit mode)
- Read latency from RSA is not included

Tooling

- Efficient development environment
  - Freely mix RSA and TMS320C64x operation together
  - Fully optimizing compiler extension
  - CCS IDE integration
- All RSA instructions supported in Compiler
  - C level intrinsics
  - Serial Assembly operations
- Code Composer Studio Simulation Environment
  - RSA model is supported

UMTS DL Chip Rate Processing



UMTS Downlink Channels Processing

- Every “subslot” of 128 chips
- Apply gain to symbols
- Generate Walsh Codes and PN codes
  - Apply Walsh Codes to PN codes and interleave
- Spread symbols by using RSA

RSA Spreader

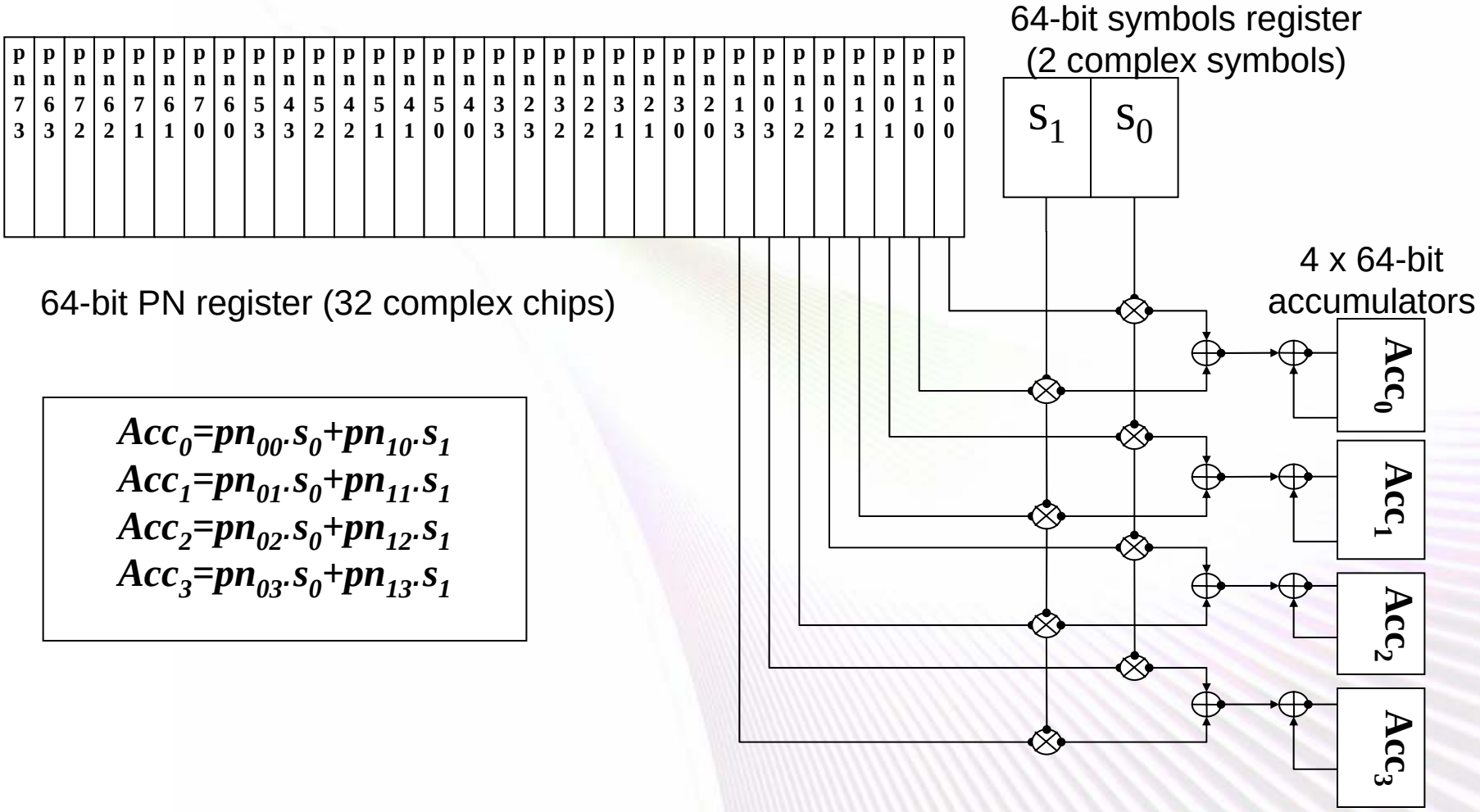
- Every 4 chips
  - Pull appropriate symbols from a symbol table based on spreading factor
  - Modulate symbols, apply gains to symbols, or use the aggregated symbols (AICH, E-HICH/E-RGCH) and place them into a ‘symbol spreading buffer’
  - Spread the ‘symbol spreading buffer’ with the codes from the ‘PN table’ by using the RSA
- Repeat up to 128 chips

RSATX Code Sample

```
_pnloadlo_1(*ptr_pn++;); // send 64 PN bits
_spread_clear_1(*ptr_syms++); // spread
_spread_1(*ptr_syms++);
_spread_1(*ptr_syms++);
_spread_1(*ptr_syms++);
```

$$Acc_0 = pn_{00}.s_0 + pn_{10}.s_1 + pn_{20}.s_2 + pn_{30}.s_3 + pn_{40}.s_4 + pn_{50}.s_5 + pn_{60}.s_6 + pn_{70}.s_7$$
$$Acc_1 = pn_{01}.s_0 + pn_{11}.s_1 + pn_{21}.s_2 + pn_{31}.s_3 + pn_{41}.s_4 + pn_{51}.s_5 + pn_{61}.s_6 + pn_{71}.s_7$$
$$Acc_2 = pn_{02}.s_0 + pn_{12}.s_1 + pn_{22}.s_2 + pn_{32}.s_3 + pn_{42}.s_4 + pn_{52}.s_5 + pn_{62}.s_6 + pn_{72}.s_7$$
$$Acc_3 = pn_{03}.s_0 + pn_{13}.s_1 + pn_{23}.s_2 + pn_{33}.s_3 + pn_{43}.s_4 + pn_{53}.s_5 + pn_{63}.s_6 + pn_{73}.s_7$$

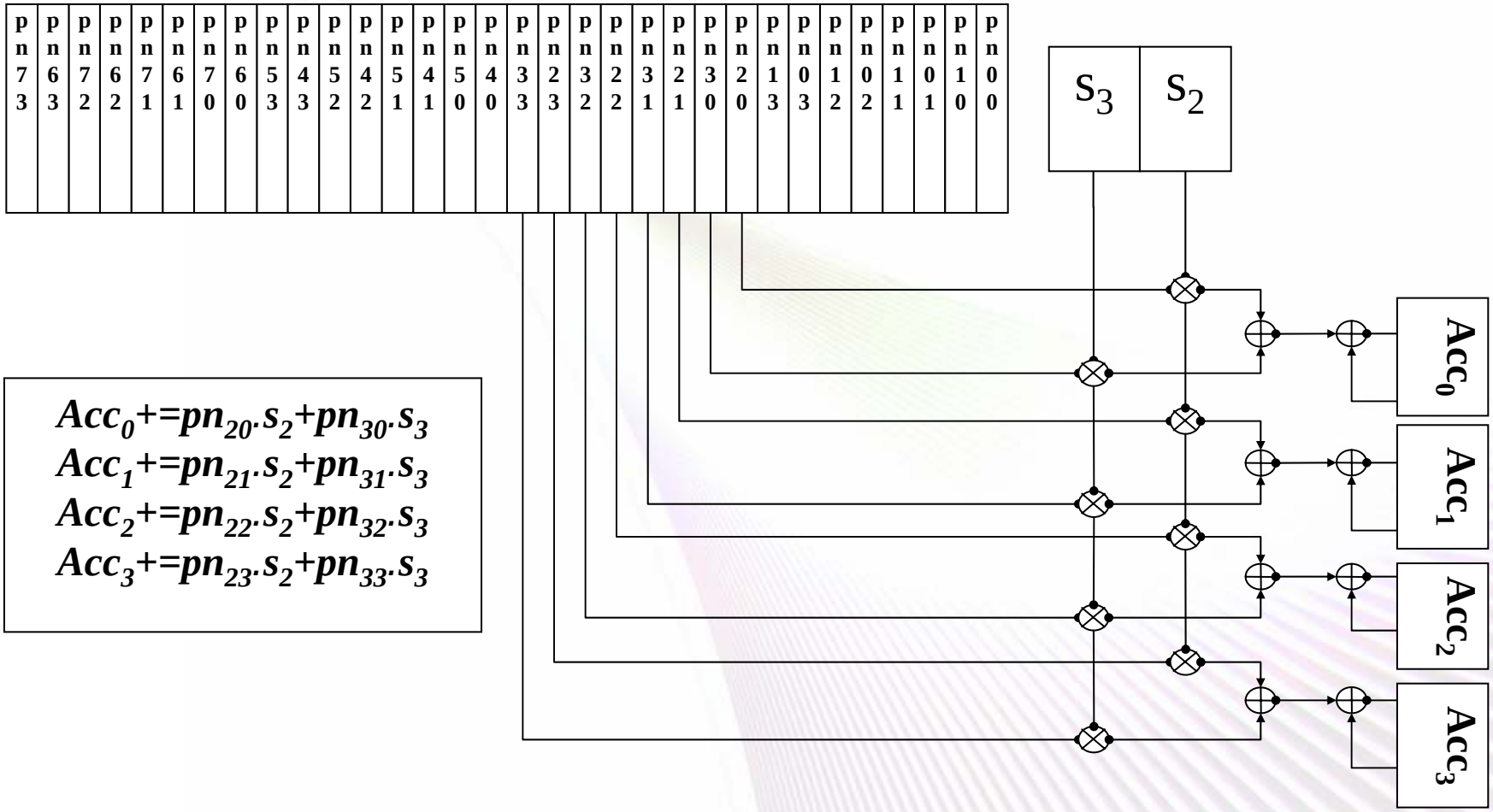
_spread_clear_1()



$$Acc_0 = pn_{00} \cdot s_0 + pn_{10} \cdot s_1$$
$$Acc_1 = pn_{01} \cdot s_0 + pn_{11} \cdot s_1$$
$$Acc_2 = pn_{02} \cdot s_0 + pn_{12} \cdot s_1$$
$$Acc_3 = pn_{03} \cdot s_0 + pn_{13} \cdot s_1$$

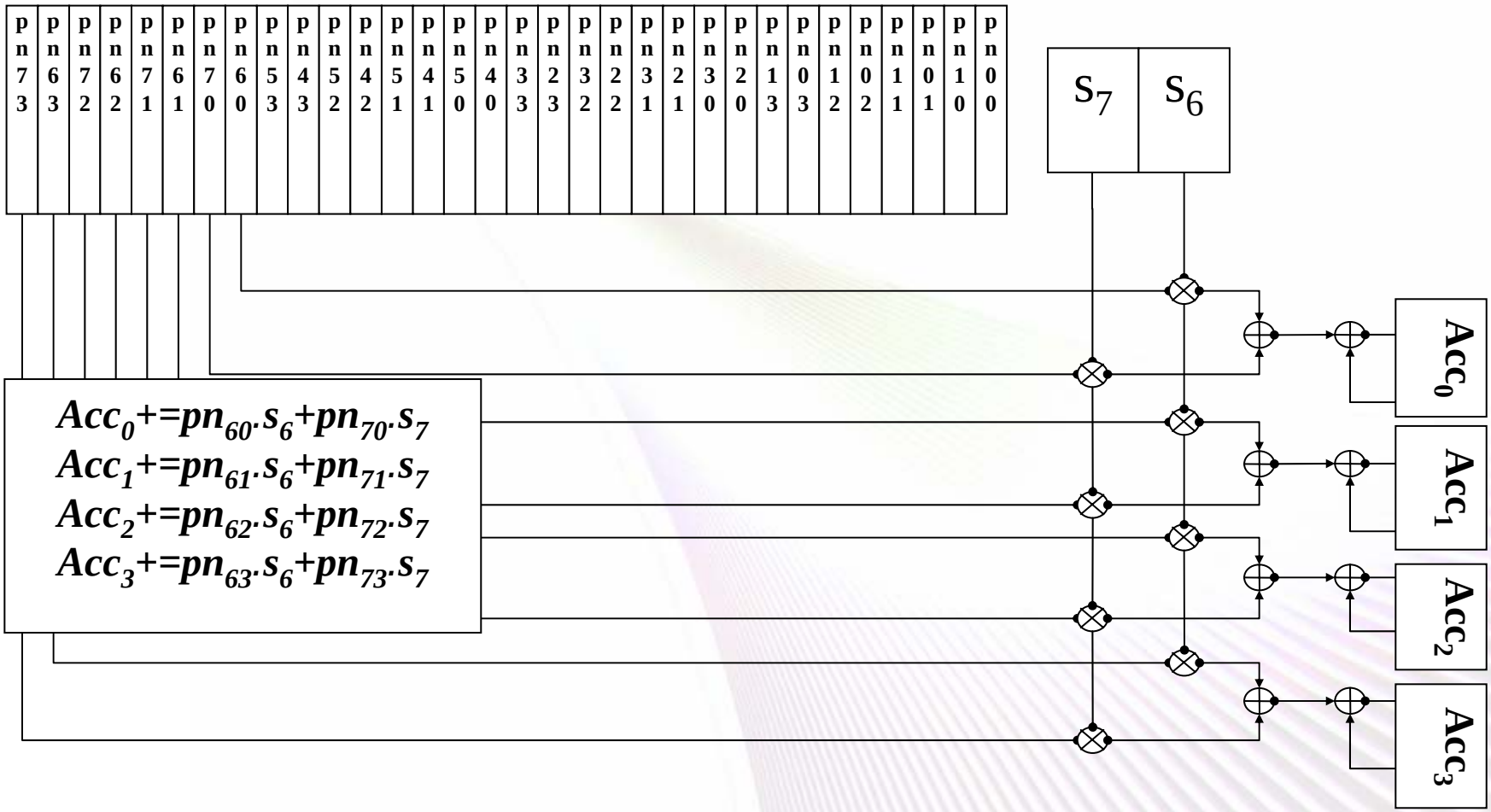
Minds in Motion

_spread_1()



$$Acc_0 += pn_{20}.s_2 + pn_{30}.s_3$$
$$Acc_1 += pn_{21}.s_2 + pn_{31}.s_3$$
$$Acc_2 += pn_{22}.s_2 + pn_{32}.s_3$$
$$Acc_3 += pn_{23}.s_2 + pn_{33}.s_3$$

_spread_1()


pn73 pn63 pn72 pn62 pn71 pn61 pn70 pn60 pn53 pn43 pn52 pn42 pn51 pn41 pn50 pn40 pn33 pn23 pn32 pn22 pn31 pn21 pn30 pn20 pn13 pn03 pn12 pn02 pn11 pn01 pn10 pn00
S7 S6
Acc0+=pn60.s6+pn70.s7
Acc1+=pn61.s6+pn71.s7
Acc2+=pn62.s6+pn72.s7
Acc3+=pn63.s6+pn73.s7
Acc0 Acc1 Acc2 Acc3

PN Codes Generator

- Generate up to 48 scrambling codes per cell
- Memory/MIPS trade off
  - CPU intensive (using _xormpy or _gmpy intrinsics)
    - Insignificant RAM usage
    - 8 MIPS/scrambling code
  - Look-up + CPU
    - 25-kB Internal RAM (L2)
    - 20 MIPS/cell
  - Look-up only (DMA)
    - 500-kB/cell external RAM
    - No CPU Load, DMA setup only

PN Codes Generator

- Rewrite 25.213 5.2.2 equations:

  Sdl,n(i) = Zn(i) + j Zn(i+217), i=0,1,…,38399.

  zn(i) = x(i+n) XOR y(i) ,

  i = 0, 1, …, 38399, 217+1,…, 217+38399,

  n = 0, 1, …, 8192.3-1

- Compute at startup and store
  - x(i), i=0,…,38399 + 8192·3 -1 (8kB)
  - x(i), i=2^{17},…, 2^{17} + 38399 + 8192·3 -1 (8kB)
  - y(i), i=0,…,38399 (5kB)
  - y(i), i=2^{17},…, 2^{17} + 38399 (5kB)

- For each cell compute

  Re{ $S_{dl,n}(i)$ } = x(i + n) XOR y(i) ,

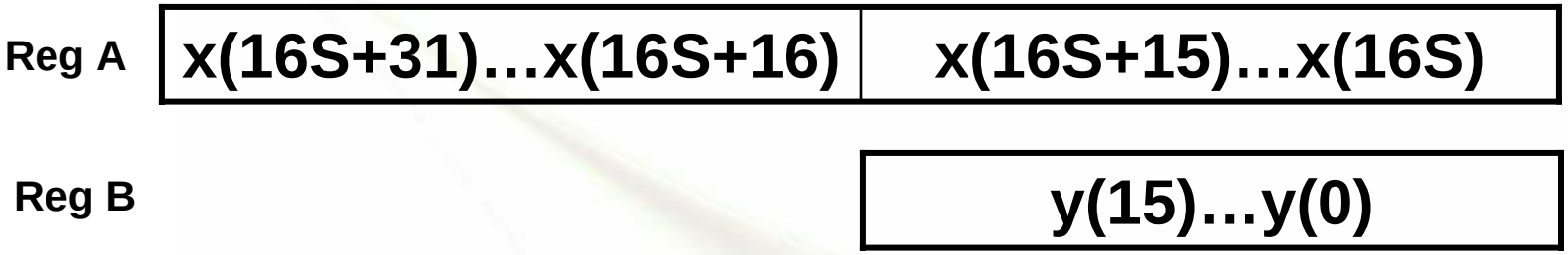
  Im{ $S_{dl,n}(i)$ } = x(i + n + 2^{17}) XOR y(i + 2^{17}) ,

  n = 16·S,…, 16·S + 15

  i = 0 ,…, 38399

  S – primary scrambling code number (0,…,511)

PN Codes Generator



$Re\{ S_{dl,16S}(i) \} = A \text{ XOR } B,\ i = 0,\ldots,15$

$A >>= 1$

$Re\{ S_{dl,16S+1}(i) \} = A \text{ XOR } B,\ i = 0,\ldots,15$

$A >>= 1$

…

$Re\{ S_{dl,16S+15}(i) \} = A \text{ XOR } B,\ i = 0,\ldots,15$

$A >>= 1$

$high(A) = x(16S + 32) \ldots x(16S + 47)$

$B = y(16)\ldots y(31)$

PN Codes Interleaving

- For each Channel
  - Use the ‘codes info table’ to retrieve channelization and scrambling codes and multiply them to form a PN code
  - Interleave the PN codes for every 8 users as required by the RSA spreader
  - Organize a 4 chips based ‘PN table’ as shown below

User 0, …, 7	Chip 0, …, 3
User 8, …, 15	Chip 0, …, 3
…	Chip 0, …, 3
User 0, …, 7	Chip 4, …, 7
…	…
User 0, …, 7	Chip 124, …, 127
User 8, …, 15	Chip 124, …, 127
…	Chip 124, …, 127

PN Codes Interleaving

Chn 3	Chn 2	Chn 3	Chn 2	Chn 3	Chn 2	Chn 3	Chn 2	Chn 1	Chn 0	Chn 1	Chn 0	Chn 1	Chn 0	Chn 1	Chn 0
I3Q3	I3Q3	I2Q2	I2Q2	I1Q1	I1Q1	I0Q0	I0Q0	I3Q3	I3Q3	I2Q2	I2Q2	I1Q1	I1Q1	I0Q0	I0Q0

Bit31 Bit0

Chn 7	Chn 6	Chn 7	Chn 6	Chn 7	Chn 6	Chn 7	Chn 6	Chn 5	Chn 4	Chn 5	Chn 4	Chn 5	Chn 4	Chn 5	Chn 4
I3Q3	I3Q3	I2Q2	I2Q2	I1Q1	I1Q1	I0Q0	I0Q0	I3Q3	I3Q3	I2Q2	I2Q2	I1Q1	I1Q1	I0Q0	I0Q0

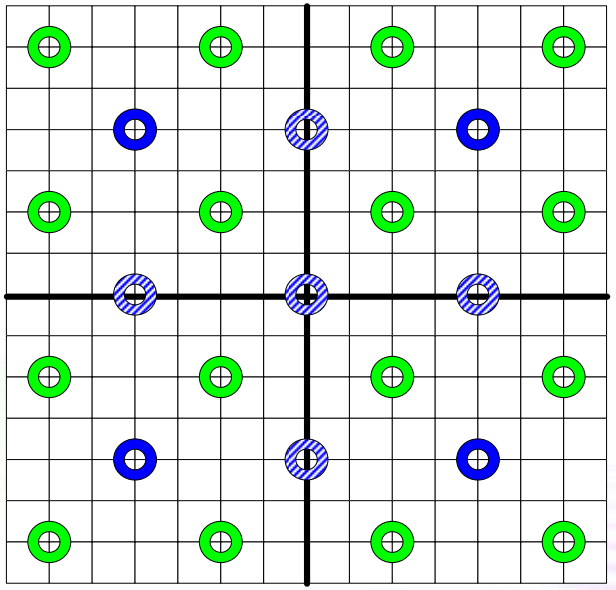
Bit64 Bit32

- 128 bits x 512 entries OVSF table
- 128 IQ bits X 48 entries PN codes
- Apply OVSF codes to PN codes
  - ovsfCode ^ ovsfCodeMask ^ scrCode
- Interleave
  - Use _shfl, _dpack2 intrinsics
  - Process 8 codes at a time

Modulation

- Modulation Types
  - QPSK
    - Each Symbol has I, Q, dtxI and dtxQ bits
  - 16 QAM
    - Each Symbol has I1, I2 ,Q1, Q2 bits
- Implementations
  - Can use the _cmpyr intrinsic, but requires data pre-processing
  - Conditional instructions (QAM):

```
gIsI = gIsQ = gain >> 1;
if ( ((sym) & 0x4) ) gIsQ += gain;
if ( ((sym) & 0x1) ) gIsI += gain;
if ( ((sym) & 0x8) ) gIsQ = -gIsQ;
if ( ((sym) & 0x2) ) gIsI = -gIsI;
symMod = _pack2( gIsI , gIsQ );
```



Basic Modulation Schemes
16 points on -7...7I:-7...7Q Grid

16QAM

QPSK (with DTX)
Data 0/1 => 1/-1
Dtx 0/1 -> Tx/Mute

AICH

- 1 channel with 16 signatures
  - Each channel +/-1
  - Aggregate to -16…+16
  - Split channel in 2

- Use _add4, _sub4 intrinsics
  - Initialize aggregated signature buffers:

```
sig4a = sig4b = sig4c = sig4d = 0;
```

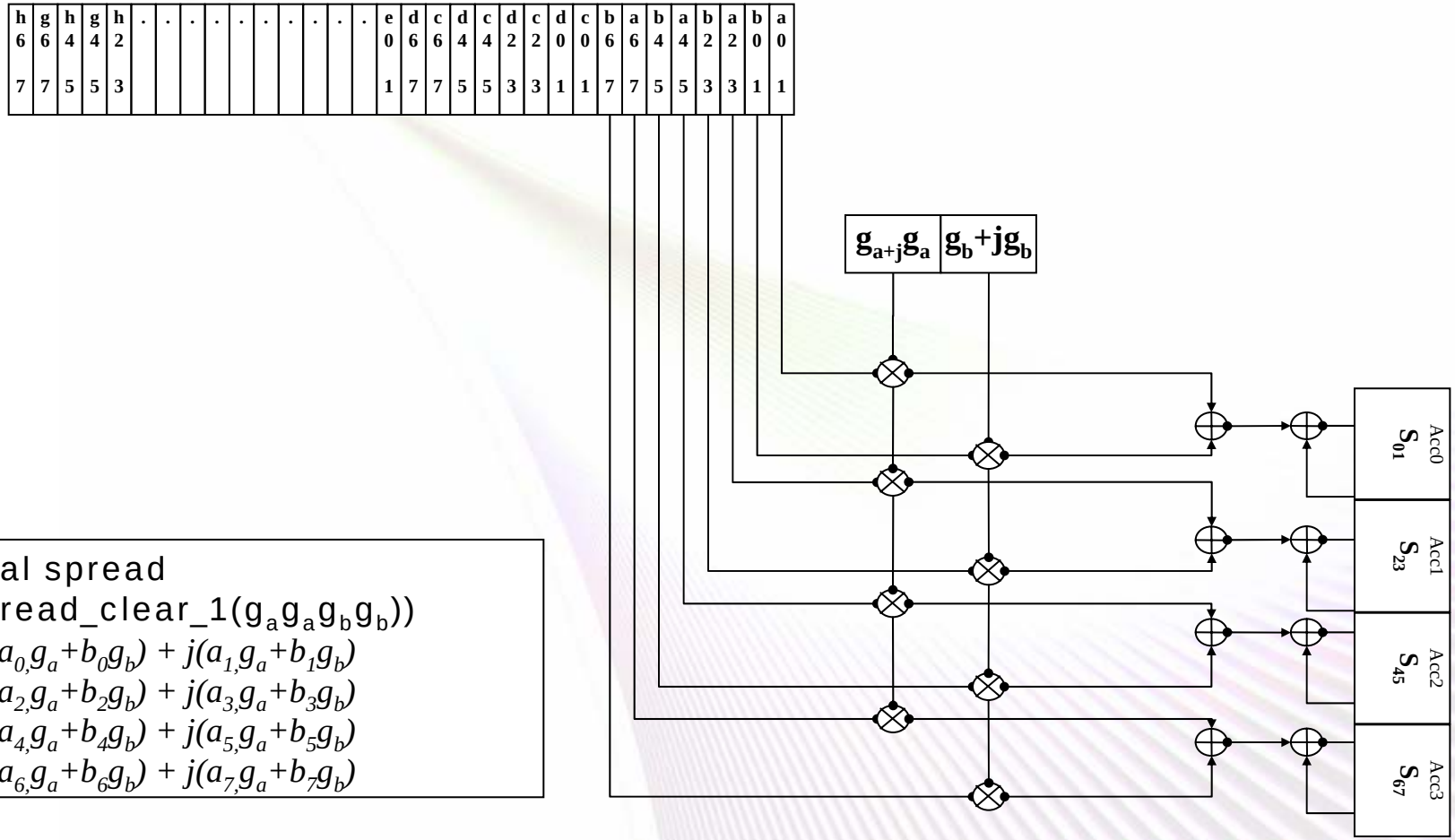
  - For each signature:

```
newSig8a = signature[i][0..7]
newSig8b = signature[i][8..15]
if ( aiInd  == ACK )
  {   /* Add this signature */
      sig4a = _add4( sig4a, _loll( newSig8a ) );
      sig4b = _add4( sig4b, _hill( newSig8a ) );
      sig4c = _add4( sig4c, _loll( newSig8b ) );
      sig4d = _add4( sig4d, _hill( newSig8b ) );
  }
if ( aiInd == NACK )
  {   /* Subtract this signature */
      sig4a = _sub4( sig4a, _loll( newSig8a ) );
      sig4b = _sub4( sig4b, _hill( newSig8a ) );
      sig4c = _sub4( sig4c, _loll( newSig8b ) );
      sig4d = _sub4( sig4d, _hill( newSig8b ) );
  }
```

E-HICH/E-RGCH Aggregation

- 40 signatures have to be aggregated over the same PC and OVSF code.
- Pre-aggregate 8 signatures to reduce spreader load
- Use RSA in spread mode
- Interleave signatures as required by RSA
  - Interleaved 64 bit sequence:
    - PN = $h_7h_6g_7g_6h_5\ldots.\ d_1d_0c_1c_0b_7b_6a_7a_6b_5b_4a_5a_4b_3b_2a_3a_2b_1b_0a_1a_0$
  - Diversity will use a different packing reflecting STTD
- Provide gains to RSA as symbols to be spread by the signature:
  - _spread(_itoll (_pack (g_a ,g_a), _pack (g_b ,g_b)))
  - The _pack (g_a ,g_a) rotates the input gain by 45 degrees
  - This will revert the RSA 45 degrees rotation
- Second RSA can be used for diversity or to aggregate symbols 64-127 while the first one aggregates bits 0-63

E-HICH/E-RGCH Aggregation

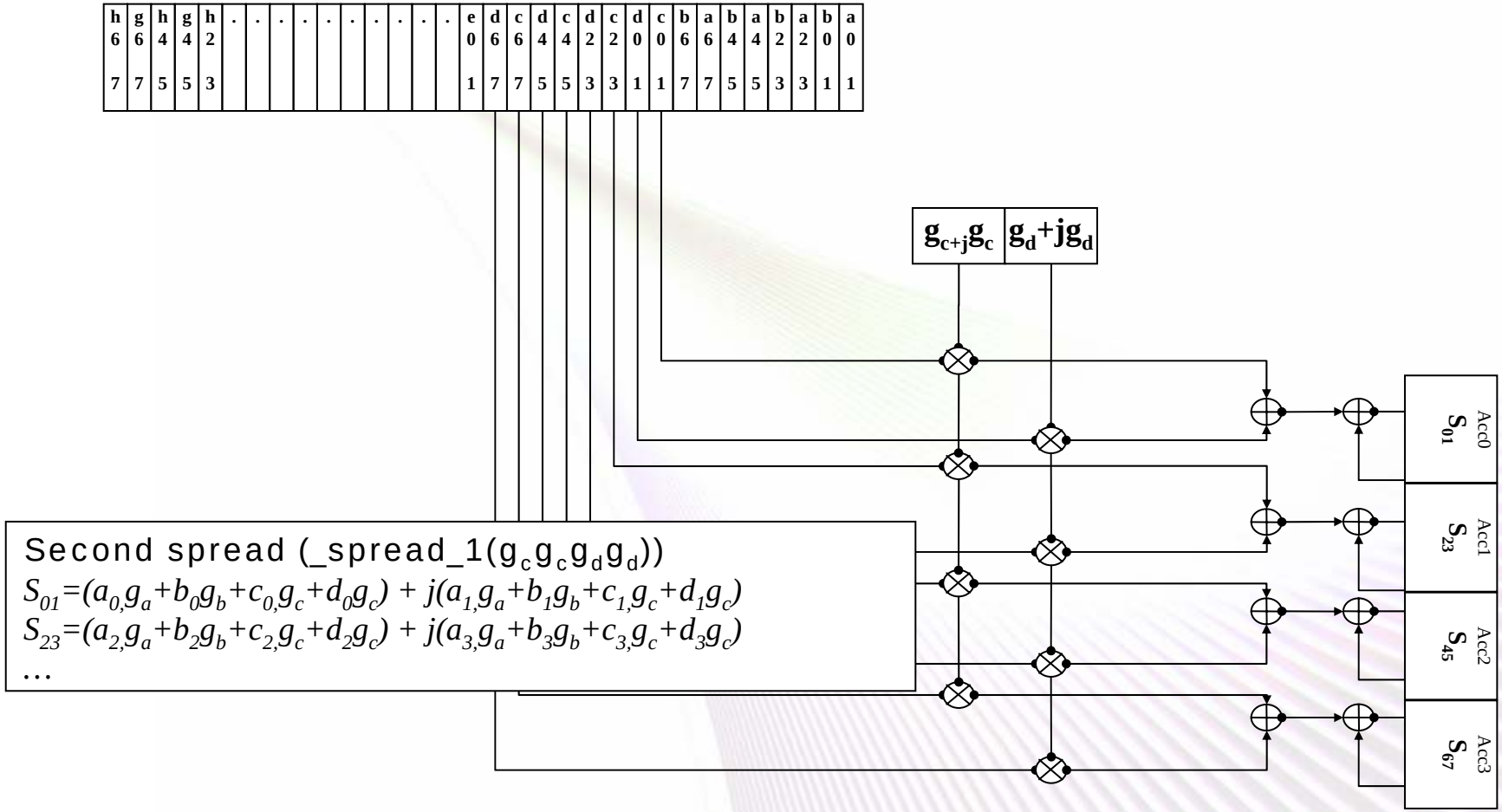


Initial spread
(_spread_clear_1($g_a g_a g_b g_b$))

$S_{01}=(a_0 g_a+b_0 g_b) + j(a_1 g_a+b_1 g_b)$
$S_{23}=(a_2 g_a+b_2 g_b) + j(a_3 g_a+b_3 g_b)$
$S_{34}=(a_4 g_a+b_4 g_b) + j(a_5 g_a+b_5 g_b)$
$S_{45}=(a_6 g_a+b_6 g_b) + j(a_7 g_a+b_7 g_b)$

Minds in Motion

E-HICH/E-RGCH Aggregation



Minds in Motion

E-HICH/E-RGCH Aggregation

- Code sample for aggregating 8 signatures and 8 bits

```
/* load interleaved signatures */
 _pnloadlo_1( PN );
/* use spread to aggregate signatures */
_spread_clear_1 ( _itoll ( _pack (g_a ,g_a), _pack (g_b ,g_b)))
_spread_1(_itoll ( _pack (g_c ,g_c), _pack (g_d ,g_d)))
_spread_1(_itoll ( _pack (g_e ,g_e), _pack (g_f ,g_f)))
_spread_1(_itoll ( _pack (g_g ,g_g), _pack (g_h ,g_h)))
/* send commands to read RSA results */
_shadow_copy_m_1();
  _read_sacc32_cmd_0_m0_1();
 …
  _read_sacc32_cmd_3_m3_1();
bp = _read_sacc32_rd_s0_1(); /* aggregated bits 0, 1 */
chBuffs[aggGrpIdx][0] = _pack2 ( _hill(bp),_loll(bp) )
bp = _read_sacc32_rd_s1_1(); /* aggregated bits 2, 3 */
…
bp = _read_sacc32_rd_s3_1(); /* aggregated bits 6, 7 */
chBuffs[aggGrpIdx][6] = _pack2 ( _hill(bp),_loll(bp) )
```

RSA Reading Latency

EFCMD.S1/.M1 RSA_CMD	;1st CPU cycle
NOP	;2nd EFI cycle
NOP	;inactive phase
NOP	;inactive phase
NOP	;inactive phase
NOP	;inactive phase
NOP	;inactive phase
NOP	;inactive phase
NOP	;Accumulators be stable for read.
NOP	;accumulators can now change
NOP	;accumulators can now change
NOP	;accumulators can now change
NOP	;accumulators can now change
NOP	;accumulators can now change
NOP	;write into EFI gasket register
NOP	;write into EFI FIFO

- 15 cycles latency
- The inactive cycles can be used for
  - Symbol spreading commands for the next iteration (or finger despreading commands)
  - Processing the results from the previous iteration
    - results scaling
    - phase/frequency correction for FD
    - antenna data / symbols read /write

RSA spreader loop

- Spread 8*N users

```
;Read PN codes and symbols for users 1 to 8
PN = MEM_PnRead
Sym = MEM_SymRead
Sym = MEM_SymRead
Sym = MEM_SymRead
Sym = MEM_SymRead

; Repeat for k = 1 to N - 1
    ;Send to RSA PN codes and spread users 8*k - 7 to 8*k
    ;Read PN codes and symbols for users 8*k + 1 to 8*k + 8
    RSA_PNLoad (PN),  PN  = MEM_PnRead
    RSA_Spread (Sym), Sym = MEM_SymRead
    RSA_Spread (Sym),       Sym = MEM_SymRead
    RSA_Spread (Sym),       Sym = MEM_SymRead
    RSA_Spread (Sym),       Sym = MEM_SymRead

;Send to RSA PN codes and spread users 8*N - 7 to 8*N
RSA_PNLoad (PN)
RSA_Spread (Sym)
RSA_Spread (Sym)
RSA_Spread (Sym)
RSA_Spread (Sym)
```

- ~80% RSA usage

Not pipelined 4-chip spreader - 8 users

```
PN = MEM_PnRead       ; Read Users 1 – 8 PN codes
Sym = MEM_SymRead     ; Read Users 1, 2 symbols
…
Sym = MEM_SymRead     ; Read Users 7, 8 symbols
RSA_PNLoad (PN)       ; Users 1 – 8 PN codes
RSA_SpreadClear (Sym); Spread Users 1, 2
RSA_Spread (Sym)      ; Spread Users 3, 4
RSA_Spread (Sym)      ; …
RSA_Spread (Sym)      ; …

;Collect results
RSA_ShadowCopy
RSA_Result0
RSA_Result1           ; RSA read delay pipe 1
RSA_Result2           ; …2
RSA_Result3           ; …3
NOP   12              ; …4-16

; Read RSA data and write to antenna buffer
Reg0 = RSA_Read
Reg1 = RSA_Read, MEM_AntWrite (Reg0)
Reg2 = RSA_Read, MEM_AntWrite (Reg1)
Reg3 = RSA_Read, MEM_AntWrite (Reg2)
MEM_AntWrite (Reg3)
```

- **12% RSA usage**

Not pipelined 4-chip spreader -16 users

```
PN = MEM_PnRead           ; Read Users 1 – 8 PN codes
Sym = MEM_SymRead         ; Read Users 1, 2 symbols
Sym = MEM_SymRead         ; Read Users 3, 4 symbols
Sym = MEM_SymRead         ;…
Sym = MEM_SymRead         ;…

;Send to RSA PN codes and spread users 1 - 8
;Read PN codes and symbols for users 9 - 16
RSA_PNLoad (PN),          PN  = MEM_PnRead
RSA_SpreadClear (Sym),    Sym = MEM_SymRead
RSA_Spread (Sym),         Sym = MEM_SymRead
RSA_Spread (Sym),         Sym = MEM_SymRead
RSA_Spread (Sym),         Sym = MEM_SymReadRSA_ShadowCopy

;Send to RSA PN codes and spread users 9 - 16
RSA_PNLoad (PN)
RSA_SpreadClear (Sym)
RSA_Spread (Sym)
RSA_Spread (Sym)
RSA_Spread (Sym)

;Collect results
RSA_ShadowCopy
RSA_Result0
RSA_Result1    ; RSA read delay pipe 1
…
```

- 21% RSA usage

Two cells pipelined spreader 16 users, 4 chips

- Pipeline the two cells

Read PN codes and symbols for users 1 – 8, cell portion 1 ***; 5 cycles***

Send to RSA PN codes and spread users 1 – 8, cell portion 1 and
Read PN codes and symbols for users 9 – 16, cell portion 1 ***; 5 cycles***

Send to RSA PN codes and spread users 9 – 16, cell portion 1 and
Read PN codes and symbols for users 1 – 8, cell portion 2 ***; 5 cycles***

Collect results cell portion 1 ***; 5 cycles***

Send to RSA PN codes and spread users 1 – 8, cell portion 2 and
Read PN codes and symbols for users 9 – 16, cell portion 2 ***; 5 cycles***

Send to RSA PN codes and spread users 9 – 16, cell portion 2 ***; 5 cycles***
NOP 2 ***; RSA read delay pipe 14 – 15***

Read RSA data and write to antenna buffer cell portion 1 and
Collect results cell portion 2 ***; 6 cycles***

NOP 12 ***; RSA read delay pipe 5 - 16***
Read RSA data and write to antenna buffer cell portion 2 ***; 5 cycles***

- 40% RSA usage

Pipelined 2 cells, 2 x 4 chips spreader

- Pipeline the two 4-chip spreaders
- Pipeline the two cells
- 46% RSA usage for 16 users
- 59% RSA usage for 24 users

MIPS Comparison

	Number of CPU cycles for N_{CELLS} cell portions, N_{CHIPS} and N_{USERS} number of chanels per cell portion	N_{CYCLES} Example (128 chips, 8 cell portions)		
		channels/cell portion **(total max)**	N_{CYCLES}	MIPS
Chip based pipelined (initial spreadsheet)	**$[10 + (5xN_{USERS}/8 + 18)xN_{CHIPS}/4] \times N_{CELLS}$**	**1-8 (64)**	**5,968**	**179**
		9-16 (128)	**7,248**	**217**
		17-24 (192)	**8,528**	**255**
Not pipelined	**$[(28 + 5 \times N_{USERS} / 8) \times N_{CHIPS} / 4] \times N_{CELLS}$**	**1-8 (64)**	**8,448**	**253**
		9-16 (128)	**9,728**	**291**
		17-24 (192)	**11,008**	**330**
Cells and chips pipelined				
8 users	**$16 + 17 \times N_{CELLS} \times N_{CHIPS} / 4$**	**1-8 (64)**	**4,368**	**131**
16 users	**$21 + 17 \times N_{CELLS} \times N_{CHIPS} / 4$**	**9-16 (128)**	**4,373**	**131**
24 users	**$26 + 20 \times N_{CELLS} \times N_{CHIPS} / 4$**	**17-24 (192)**	**5,146**	**154**
N_{USERS} (>=24)	**$(11 + 5xN_{USERS}/8) + [20 + 5x(N_{USERS}/8-3)] \times N_{CELLS} \times N_{CHIPS}/4$**			

Minds in Motion

Using RSA for Finger Despreading

- 4 Fingers for 8 different Users with the same spreading factor (SF)
  - offsets in Chips 0, ½ , 1 , 1½
- RSA1 will do offset 0,1
- RSA2 will do offset ½ , 1½
- The idea is to load 8 users PN for 4 chips for all the different SF
- For SF = 2 we need to zero out each 2 chips at a time.
- The PN code consists of the Walsh code and the scrambling code combined

TI Developer Conference
March 7-9, 2007 • Dallas, TX

Click to add title

Click to add sub-title

Click to add contact info

Minds in Motion