Design of a CDMA system simulator and implantation on a TMS320C6201

D. Janu [2], G. Baudoin [1], J.-F. Bercher [1], O. Venard [1]

[1] Signal Processing and Telecommunications department
ESIEE, BP99, 93162, Noisy Le Grand, CEDEX, FRANCE
janud@esiee.fr, baudoing@esiee.fr, bercherj@esiee.fr, venardo@esiee.fr

[2] Technical University of Brno, Czech Republic

ABSTRACT:

This paper presents the different steps and results of the development of a CDMA (Code division Multiple Access) real-time simulator based on the IS95 standard and implemented on a TMS320C6201. Optimization works have focused on the elements that will also be important for European UMTS standard : Viterbi decoder, long code generation, correlators.

1. INTRODUCTION

CDMA techniques are becoming more and more important. Some cellular mobile radiocommunications systems are already using CDMA (IS95 in USA for example), and the new generation of cellular European Mobile telecommunication networks (UMTS : Universal Mobile Telecommunication system) will use CDMA.
The performances of DS-CDMA (Direct Sequence CDMA) systems could be increased by developing new algorithms for the receiver taking into account the multi-user aspects. We have developed a real time CDMA base-band simulator on a TMS320C6201 in order to test these new algorithms and also to evaluate the capacities of the C6201 in a CDMA base station.
This DSP is particularly interesting because it is very powerful and because it can be efficiently programmed in C which reduces the development times compared to assembler programming.
In this work we have developed and implanted on the TMS320C6201 a CDMA system based on the IS95 standard. The forward link emitter has been implanted as well as part of the receiver for the reverse links (base station approach). The receiver is for the moment a classical Rake receiver.
First the different blocks have been written in Matlab™ then in C with fixed point format and transferred on the DSP.
The programs are as flexible as possible in order to be able to quickly transform them for another standard.
This work was performed at ESIEE during the 3-month master project of a student from the technical university of Brno.
The following equipment was used : a PC with software development tools for the TMS320C6201 and the numerical computation software Matalb™.

2. IS95 REVERSE AND FORWARD LINK

It is out of the scope of this paper to describe in details the CDMA principles and IS95 standard. Many pieces of information can be found on the web [1, 2, 3] and in [4, 5, 6].
Different techniques are used for multiple access in mobile cellular radio networks. The classical approaches are : FDMA, TDMA and DS-CDMA (Frequency Division, Time Division and Direct Sequence Code Division Multiple access). These techniques share the radio resources between several users. Each user is allocated :

- in FDMA a portion of the frequency band,
- in TDMA a time slot
- in CDMA a specific code.

In CDMA, each user can occupy the full frequency band continuously in time. The separation between users is obtained thanks to individual orthogonal pseudo noise codes. Each user sends a bit sequence at the bit rate 1/Tb. This sequence is multiplied by the user code sequence at the chip rate 1/Tc which is higher than the symbol rate. At the receiver, the signal is re-multiplied with the user's code. CDMA is a spread spectrum technique, with a processing gain equal to Tb/Tc, resulting in frequency diversity advantages.
In the IS95 standard, the physical layer for the forward and reverse links has the following characteristics.

Forward link

The forward link is the link from the base station to the mobile. Different types of channels can be distinguished : pilot, paging and traffic channels. They are synchronously added and sent together to the mobiles. The pilot channel is permanently sent by the

base station for the synchronization of the mobiles. There are at the maximum 64 different channels by base station. Figure 1 describes the main parts of a forward link traffic channel..

Reverse link

The reverse link is the link from the mobile to the base station. Figure 1 describes a reverse link traffic channel.

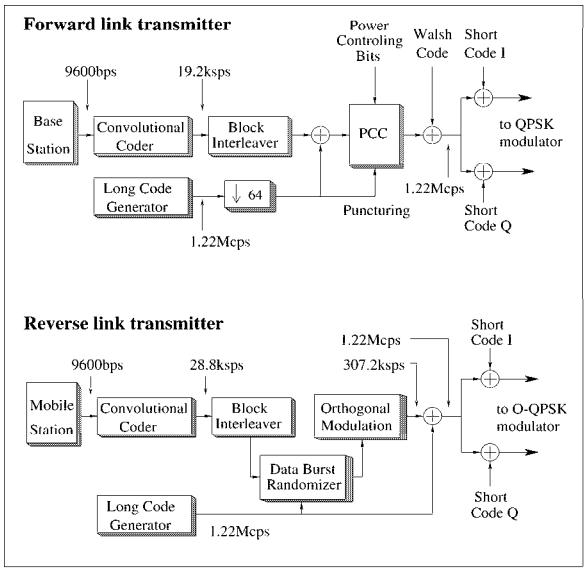

Figure 1: Traffic Data Channels of the IS95 standard

Rake receivers

For an additive white gaussian noise channel, the receiver minimizing the symbol error probability is a simple adapted filter to the user's signature. But radio channels are multi-paths channels. The received signal can be represented by a sum of a few delayed and attenuated replicas of a main component, each replica corresponding to a particular « radio path ». Most of existing CDMA receivers are Rake receivers. They take advantages of the wide spectrum of CDMA emissions to resolve main radio propagation paths and recombine them synchronously to increase the signal to noise ratio. Classically a Rake receiver has 3 or 4 fingers, each finger being associated to a radio path. Figure 2 represents a Rake receiver.

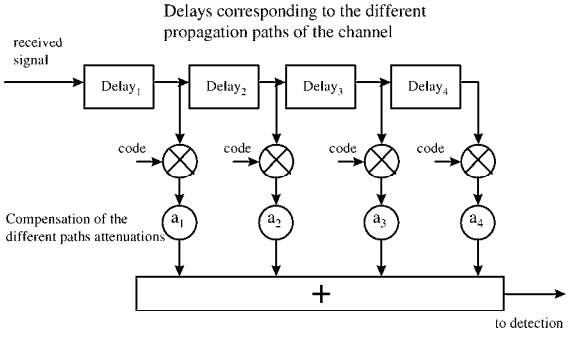

Figure 2: Rake receiver

3. MATLAB SIMULATION OF CDMA IS95 STANDARD

There were two main purposes for using MATLAB. First we used this user friendly tool for understanding and simulating the CDMA inner structure. Second the MATLAB results were used for validation of our TMSC6201 implementation results.

Implementation of Base Station algorithms was our main interest. This includes forward link (downlink) emitter and reverse link (uplink) receiver part. However the reverse link emitter was also successfully simulated. Only traffic data channels were simulated since they are the most complex in the CDMA channel generation.

The following blocks of IS95A standard were simulated for the forward link: Convolutional Coder, Long Code Generator, Block Interleaver, PCC (Power Control) bits puncturing, Walsh Code spreading, I and Q short codes and FIR shaping filter.

In the reverse link similar blocks were employed plus Walsh Code Orthogonal Modulation. Two important blocks of the Base Station receiver were optimized. Viterbi decoder and RAKE Receiver.

Both in MATLAB and C, we worked with input frames corresponding to 20ms intervals. At the initial bit rate 9600bps these input frames result in blocks of 192 bits coming from vocoder.

4. IMPLANTATION ON THE TMS320C6201

The TMS320C6201

The DSP C6201 [7,11,12] is a 32 bits fixed-point DSP. Its typical cycle time is 5ns at 200 MHz. Its CPU contains 2 almost identical blocks of 4 functional units (2 16x16 bit Multipliers and 6 ALUs). Each block of 4 units communicates with 1 general purpose register file of 16 32-bit register each. Six of the units have access to the opposite side's register file via a cross path. There are two 32-bit paths for loading data from memory to the register files and two 32-bit paths for storing register to memory. The ALUs can perform 32-bit or 40-bit fixed point and logical or bit fields manipulation operations.

The TMS320C6201 is a VLIW (Very Long Word Instruction) DSP. Its architecture called VelociTI allows up to 8 instructions to be executed in parallel on the 8 functional units, leading to a maximum of 1600 Mips. The DSP fetches 8 32-bit instructions at a time. This constitutes a Fetch Packet (FP) of 256 bits (VLIW). The 8 instructions of a Fetch Packet can be executed serially, in parallel or partially serially. All instructions of a FP executing in parallel (8 at the maximum) constitute an Execute Packet (EP). No 2 instructions in the same EP can use the same

resources (each instruction must use a different functional units, and there are constraint on cross-paths, constraints on Load and Store, Constraint on long data ,constraint on register reads and writes).
The on-chip memory is divided into a program memory and a data memory space. The data memory can be byte, half-word (16 bits) or word (32 bits) accessed. The internal data memory contains 64K bytes. The internal program memory is made of 2 K 256-bit words.
The DSP uses a pipeline with 3 steps : Program Fetch (4 cycles), Decode (2 cycles) and Execute (1 to 6 cycles depending on the delay slot of the instruction). A delay slot of n cycles means that the results of the instruction will be available only n cycles later. For example, the branch instructions has 5 delay slots. It may be necessary to introduce NOP (no operation) instructions after an instruction with non zero delay slots in order to wait for its results to be ready.
All instructions can operate conditionally, depending on the value (zero or not) of one of 5 condition registers. This helps preventing the delays introduced by branches.
To use this DSP at its maximum capacity, it is necessary to optimize the parallel use of the 8 functional units and Texas-Instruments offers specific tools for C and/or assembler optimization.

Programming languages and associated software development tools for the TMSC6201

Three coding languages can be used : ANSI C language (files with extension .c), assembly language (files with extension .asm) and linear assembly language (files with extension .sa) [8,9].
The linear assembly is similar to regular C6201 assembly code in that it uses C6201 instructions, but it is not necessary to specify all of the information that is necessary for regular assembly such as : parallel instructions, pipeline latencies, register usage, which functional unit is used. The assembly optimizer can determine this information itself in an optimized way.
The 3 coding approaches can be ranked by decreasing order of development complexity : assembly code, linear assembly code and C-code.
The classical development flow consists in 3 phases:

- phase 1 = develop C code.
- Phase 2 = refine C code
- phase 3 = write linear assembler

According to TI documentation, in comparison with good hand-coded assembler, the C compiler will typically generate 70-80% efficient code. Therefore, with have only used C programming in our work. We chose C programming to shorten the development time and to test the efficiency of the C optimizer to fully use the parallel capacity of the DSP.

The main software development tools (optimizing C compiler, linear assembly optimizer, assembler and linker) can be called in a single step by the shell program cl6x in order to create an executable file.
The executable file can be loaded into the debugger which is the user interface for most of TI's development tools : software simulator, evaluation module (including an evaluation board with a DSP) or the emulator [10]. We used the Release 2.00 of the simulator sim6x. This simulator has a profiling mode which allows to determines the number of cycles used by the different parts of the program.

C - Programming of the DSP C6201 for simulation of the base station IS95 Emitter and Receiver

Only the base station part of the standard was simulated on the DSP, since this DSP, because of its power consumption, is better suited for base stations than for mobiles.
The processing was done frame by frame of 20ms.
Almost through the whole emitter part, we worked with unsigned 32-bits integer words. Six words were necessary to store one input frame of 192 bits (9600 bps). From output of the convolutional coder to Walsh code spreading, one frame of symbols (encoded bits at 19.2 Kbps) was resolved in twelve words. Then 768 (64x12) words were required to store all the chips in one 20ms frame. In the emitter, most of the processing was done at the bit level
There were 2 development phases :

1. Making the C program work and checking it.
2. Refining and better optimizing the C-code.

First step : development of a correct C code
In the first step we compiled, optimized and linked our C programs by the cl6x shell program with the following standard set of options :

```
cl6x -g -o -k -mg filename.c  -z lnk.cmd -l rts6201.lib
-o filename.out.
```

These C-compiler options correspond to the maximum degree of optimization compatible with symbolic debugging and profiling.
We used the internal memory for program and data to optimize the processing speed.
During this step, we compared DSP and Matlab results. The DSP memory pages can be saved in COF (Common Object Format) Files with the debugger. Two Matlab functions were written for reading and writing COFF files from Matlab. So, it was easy to compare DSP and Matlab results as waveforms in one Matlab figure window and to load into memory initial values prepared in Matlab.

Second step : refining of the C code to increase speed
After the results were validated with those we got in Matlab, we started the C code refinement process.

Only such algorithms that can yield a significant improvement from the global point of view were optimized.
The objective of the optimization was to decrease the number of cycles regardless of the program size.
We can distinguish 2 kinds of optimizations : generic optimizations non specific to the processor and optimization specific to the architecture of the C6201.
Among generic optimizations, we find : algorithm optimization, cost-based register allocation, alias disambiguation, branch optimization and control flow simplification, data flow optimizations, expression simplifications, inline expansion of some routines, induction variable optimization, loop-invariant code motion, loop rotation, register variables, register tracking / targeting.
There are 3 basic optimizations specific to this DSP: increase the parallelism of instructions (up to EP of 8 instructions), fill delay-slots with useful instructions instead of NOP, use 1 word-access instead of 2 half-word accesses.
Most of these optimizations can be automatically realized by the optimizing C compiler depending on the chosen level of optimization. The chosen level is specified by the -on option of cl6x, where n is an integer between 0 and 3 (3 = maximum optimization). In order to efficiently schedule instructions in parallel, the compiler must determine their dependencies, because only independent instructions can be parallel. The option -mt allows the compiler to make hypothesis to eliminate memory dependencies. The option -pm enables program level optimization. It is not possible to use the option for the maximum optimization (-o3) and to keep symbolic debugging and profiling. So the option -o3 was reserved for final optimization in the second step.
In the refining optimization step, different techniques can be applied to refine the C code in order to help the C optimizing compiler to be more efficient:

1. Using intrinsics,
2. Using word (32-bit) access for 16-bit short ,
3. Changing the structure of the program by
   - ⇒ Modifying the order of the processing blocks,
   - ⇒ Decrementing loop counters instead of incrementing,
   - ⇒ Loop unrolling,
   - ⇒ Software pipelining the instructions manually

- **Intrisincs**: the c6x compiler provides special functions that map directly to inlined c6201 instructions. DSP Instructions that are difficult to implement in C code are supported as intrinsics. Intrinsics are specified by a leading underscore and called like functions with arguments. For example, the intrinsic _add2(src1, src2) adds the upper and lower halves of src1 to the upper and lower halves of src2 and return the result.
- **Using Word access for short data**: the data busses are on 32 bits but very often we work with 16-bit data. It is efficient to replace 2 memory half-word accesses by a single word access, the word containing two 16 bits data. In the same way, the C6201 has specific instructions with corresponding intrinsics (_add2 for example) that work on two 16-bit data stored in the upper and lower part of a 32-bit register. We have used this method for FIR shaping filtering of short data.
- **Software pipelining :** is a technique used to schedule the instructions in a loop so that multiple iterations of the loop execute in parallel. In order to illustrate this concept, suppose that the loop contains 3 steps A, B, C and that it must be iterated 5 times (trip count = 5). If the 3 steps are dependent, at least 15 = 5 x 3 cycles are necessary to run the loop. But it is possible to decrease this number of cycles by pipelining successive iterations. The following figure 3 shows that the loop can be executed with a loop kernel where 3 instructions, from different iterations, execute in parallel (C1//B2//A3 for example). So the loop is finished after 7 cycles instead of 15 without software pipelining.

Cycle number						
1	A1					Prolog
2	B1	A2				
3	**C1**	**B2**	**A3**			Kernel
4		**C2**	**B3**	**A4**		
5			**C3**	**B4**	**A5**	
6				C4	B5	Epilog
7					C5	

Figure 3 : software pipelined loop

Software pipelining is automatically done by the C optimizer from optimization level 2. But it can be helped manually. Coming back to the example, more efficiency could be obtained for a loop with more steps (up to EP of 8 parallel instructions). When the body of a loop is too small to really benefit of software pipeling, it is possible to partially unroll the loop manually. For example instead of repeating N time a set of K=3 instructions, it is possible to repeat N/2 times a set of 6 = 2x3 instructions in order to be able to have 6 instructions in parallel in the kernel.

- **Loop unrolling :** means repeating the body of the loop many times and decrease the loop counter. It increase the number of instructions available to execute in parallel and helps software pipelining.

- **Decrementing loop counters :** only loops with decrementing counters can be software pipelined. The optimizer (-o2 or -o3) tries to convert incrementing counters in downcounting ones. But it does not always succeeded and it can be helped manually. Testing that a counter is at zero is efficient on the DSP because all instructions (including loop B) can be conditionally executed depending on the value zero or non zero of a conditions register (tat can store a loop counter).

An example of optimization process

We have chosen the Long Code Generator algorithm as an example of optimization because it was one of the more time consuming. Figure 4 represents the Long Code Generator with consists of a PN (Pseudo-Noise) sequence generation and of a Long Code Mask applied to the PN Sequence. The PN sequence is created by a 42-bit long Logical Feedback Shift Register. A 42-bit logical mask (public Electronic Serial Number or private Mobile Identification Number) is applied on the PN sequence. This Long Code Generator is used to generate the scrambling sequence in the forward link.

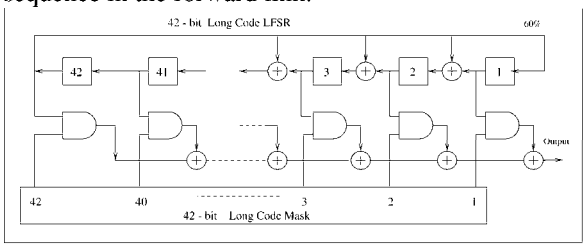

Figure 4: Long Code Generator

We cut the two 42-bit registers in two parts and stored them in 2 integer words, one with the 10 MSB and the other one with the 32 LSB. Both updates and 'and' operations are executed individually and then connected (figure 5).

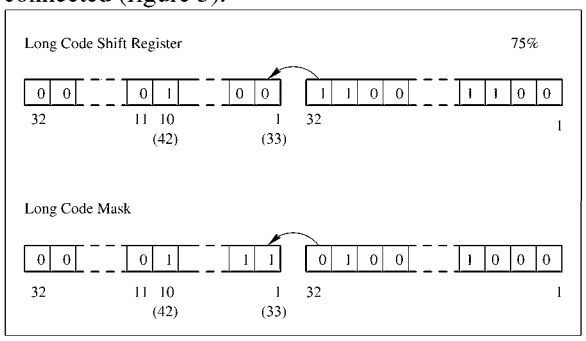

Figure 5: Long Code stored in two words

Long Code generator requires 24576 (192x64x2) shift register updates per 20ms interval The output is calculated after every 64 shift register updates, since the forward link uses long code decimated by 64. We tried to optimize it as much as possible.

Algorithm optimizations :

We have optimized the computation of the output chip as a modulo 2 inner product of the results of 'and' operations between the mask and the current shift register. We called this calculation inner_modulo2. The output chip is equal to 1 (respectively 0) if the number of XOR inputs equal to 1 is odd (resp. even). The 42 XOR inputs are stored in a variable called result.

In a first method, we counted the number of '1' in 'result' by simply adding each bit of 'result' one after the other to a 'number_of_ones' register initialized at 0. This was done by adding the LSB of 'result' to 'number_of_ones' and then shifting 'result' 1-bit to the left. This was iterated 42 times. This method was called 'shifting'.

The second method is called 'table'. The figure 8 gives its pseudo-code. Basically the same principle is used, but instead of working bit by bit we worked with packets of 4 bits at a time and we pre-calculated all the possible numbers of '1' in the binary representation of integers between 0 and 15. We prepared an array - 'table' of length 16. Each value in the array represents a number of '1'. For example the seventh value says how many '1' are in the binary representation of number 6 (we start from zero). The 4 LSB of 'result' are used to address the table, then the read value is added to 'number_of_ones' and 'result' is 4-bit left shifted. This is iterated 11 times. The output of the Long Code Generator is modulo2 of the number of ones. If it is even number, '0' is sent out, if it is odd '1' is sent out.

Refining of the C code

Mostly we used logical and no multiplication was employed, so none of 2 multiplying units of the processor was involved which limits to 6 the possible number of parallel instructions.

Three C refinement techniques were successfully applied : decrementing counters, modifying the order of blocks of code, partial loop unrolling.

Decrementing counters :We used two counters in the main loop body. One counts up to 64 and is actually the decimation counter. The other counter checks whether a word is filled with 32 output chips (one output chip occupies 1 bit position in a word). These counters and the main loop counter were upcounting. In our Long Code only the main loop counter was changed to decrementing by the C - compiler. We changed the two remaining ones by rewriting the C code and obtained good improvement (Table 1).

We also tried *different block orders*. Originally in the main loop, the updating of the shift register state was the first block, the checking of the decimation counter was the second. We found an improvement after

reordering these blocks and putting the checking of the counter before the register updates.
Loop unrolling : we have tested different unrolling factors repeating 1 to 8 times the body of the loop. Table 1 gives the obtained results for a loop of 2600 iterations only (instead of 24576 for the full loop).

Unrolling factor	number of cycles	Note
1	33637	No unrolling
2	26361	'2' unrolling
4	22475	'4' unrolling
8	20533	'8' unrolling

Table 1: unrolling the long code generator loop body

Table 2 shows the optimization history of the Long Code Generator.

Cycles	Pipelining	Number of Branch instructions	Counters 64	Counters 32	Update register method
6 520	YES	5	+	+	"shifting"
5 704	YES	3	+	+	"shifting"
5 500	YES	5	+	+	"shifting"
5 084	YES	5	-	+	"table"
3 069	YES	3	-	-	"table"

Note : The numbers of cycles are valid for 384 long code register updates, which is equal to 1/64 th of input data frame, the number 3 069 correspondsto the final result of 223 202 cycles.
The number of loop iterations was shortened for only debugging purposes.

Table2: Long Code Generator optimization history

Results obtained for the forward link emitter
Table3. shows an overview of algorithms for the forward link, their numbers of cycles required for 20ms interval and whether pipelining was achieved or not. The algorithms with lower numbers of cycles were not optimized.

	Number of cycles	Pipelining
Convolutional coder	4 522	NO
Block interleaver	2 298	YES
PCC data puncturing	483	NO
Long Code generator	223 202	YES
Walsh code	983	NO
1 Short code	147 474	YES
TOTAL for emitter	**526 436**	

Table3: Number of cycles for forward link emitter

5. WORK PLANNED

We have not given the results obtained for the receiver, because the optimization are not fully finished for the rake receiver and the Viterbi decoder. Both of them have been simulated with Matlab and a first C code have been developed.

In our future work we will finish this optimization and then adapt the code for UMTS W-CDMA standard.
We will also test new receiver algorithms for multi-user detection.

6. CONCLUSION

A base station CDMA forward link emitter was simulated using Matlab and implemented on the TMS320C6201 using C programming. The algorithms requiring the largest part of the processing time (long code generator for example) were optimized. Some of the receiver blocks are still to be optimized.
4 millions cycles of the processor are available for each 20 ms frame (with 5ns cycle time). The total number of cycles for 1 forward link emitter is about 530 000 cycles by frame. But the short code generators are the same for the 64 channels of one CDMA channel and the long code varies only in user's mask. If we subtract the number of cycles of both I and Q short code generators (about 295 000) and the number of cycles of the long code generator register updates (about 200 000) from 530 000 we obtain 35 000 cycles specific to each channel. So with 4 000 000 cycles available on 1 DSP every 20ms, it should be possible to implement up to 100 CDMA logical emitters (excluding FIR). This means a whole CDMA forward link channel emitter on one DSP.

REFERENCES

[1] *http ://www.cdg.org*
[2] *http ://www.qualcomm.com*
[3] *http ://www.ti.com*
[4] Flikkema, P. *Spread Spectrum Techniques for wireless communications,* IEEE Signal Processing Magazine, p 26-36, May 1997.
[5] Simon, Marvin K., Omura, Jim K., Scholtz , Robert A., Levitt, Barry K. *Spread spectrum Communications Handbook*, McGraw-Hill, 1994.
[6] Viterbi, Andrew J., *Principles of Spread Spectrum Communications*, Addison-Wesley, Reading, MA,1995.
[7] *TMS320C6X CPU end Instructions Set Reference Guide, Texas Instruments Inc., 1997.*
[8] *TMS320C6X Programmer's guide Preliminary, Texas Instruments Inc., 1997.*
[9] *TMS320C6X Optimizing C Compiler, User's guide Preliminary, Texas Instruments Inc., 1997.*
[10] *TMS320C6X C Source DEbugger, User's guide Preliminary, Texas Instruments Inc., 1997.*
[11] *TMS320C6X DSP Design Workshop, Student Guide, Texas Instruments Inc., 1997.*
[12] *TMS320C6201 Digital Signal Processor Product Preview, Texas Instruments Inc., 1997.*