Cycle Accurate Simulator for TMS320C62x, 8 way VLIW DSP Processor

Vinodh Cuppu,
Graduate Student, Electrical Engineering,
University of Maryland, College Park
For ENEE 646 - Digital Computer Design, Fall 1999

This project report describes the first attempt and first release of a VLIW DSP processor simulation tool, which will be free and will be publicly available. It offers a complete cycle accurate, execution driven simulation environment of the Texas Instruments TMS320C62x series of very long instruction words DSP processors. The need for this arose due to unavailability (atleast from the searches conducted by this author) of any such simulation tool neither in the open source nor without cost academic use. This report describes the architecture of the processor the simulator models, a full description of how it can be used and the other pieces of software that might be needed to make use of it comprehensively. At the present moment, this tool can execute and gather statistics to a considerable degree of accuracy of execution.

1. Introduction

Microarchitecture simulation and performance analysis are valuable uses of simulation tools that microarchitecture designers and researchers use. Such tools very in the degree of accuracy of functional execution, timing and/or speed of execution. Although, many such tools exist [1,2,3], this author has not been able to find any in open source or any without cost for academic use, that models a reasonably complex VLIW (very long instruction word) processor.

Simulation tools are indispensable aides to both designers and researchers in computer architecture, due to their ability to study and validate new designs without the cost of actually building the hardware. They also provide for a good platform on which researchers can explore a wide spectrum of design choices, which might be practically not feasible, either due to immense silicon real-estate or due to shear amount of time needed to physically fabricate the choices on silicon. It is a widely accepted and known fact that hardware modeled in software provides considerable speed in validating the functionality and flexibility in removing bugs and rechecking. It is often a grueling task to fabricate a prototype on squashing every bug and re-testing the prototype for correctness. In addition, simulation tools also allow one to study the combined interaction of all the architectural features, before anything is built and can bring out potential bugs that might not have otherwise be detected.

The simulation tool models a recent DSP processor from Texas Instruments, the TMS320C62x, which is the fixed point processor of the C6x family. The floating point processor is the TMS320C67x. The differences between the two processors will be detailed in the architecture section.

The present technology revolution has put DSP processors in a strong position, due to their targeted wide application space. Current estimates put a 30% growth [4] in the DSP market, the fastest in the entire semiconductor market, for the next five years. Figure 1 shows the targeted market for DSP and mixed signal/analog [5]. These processors are finding uses in all types of communication, whether wireless, wire or satellite communications, automobiles for cruise control, anti-lock braking, mp3 and other music players, digital cameras, digital video, and every kind and form of digital communication out there.

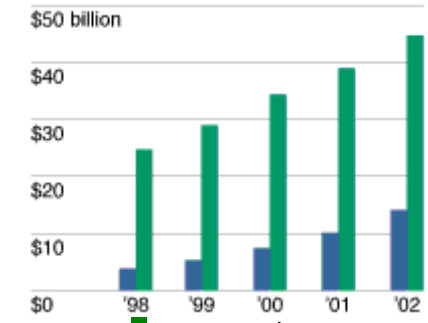

Figure 1 : Market for DSP and Mixed Signal/Analog Products [5]

The behavior of the applications that run on these processors pose possibly interesting behavior at the architecture level. This simulation tool can be used for microarchitecture exploration, performance analysis and application analysis on DSP processors. This is not a widely published area in the research community. The lessons learned in creating this tool can be used to create a simulation tool for the IA-64 in the future [6].

The rest of this report contains details about the architecture of the processor, installation and usage instructions. Section 2 contains a brief overview of the architecture of the ‘C6x. Section 3 describes installation and compilation instructions. Section 4 describes usage instructions and an overview of the capabilities of the simulation tool. Section 5 describes the architecture of the simulator, and some implementation details. Section 6 contains information about other software that might be used in conjunction with this simulation tool.

2. Architecture

The TMS320C6x (‘C6x) platform of digital signal processors is the newest of the TMS320 family of digital signal processors from Texas Instruments. The TMS320C62x (‘C62x) is the fixed-point device and the TMS320C67x (‘C67x) is the floating-point device. Both these feature the VelociTI™ architecture, which is a very long instruction word (VLIW) architecture. VLIW is an architecture where each instruction word has two or more instructions in it. The instructions in each instruction word can be specified to be executed in parallel or serial or any combination thereof.

The core of the cpu consists of 32 general purpose registers of 32-bit lengths, which can be combined to store 16 long values of 40-bit lengths. The core can execute up to eight instructions every cycle, one each for the eight functional units it has. There are two multipliers and six ALUs. The functional units are divided into two similar sets of basic functional units. Each set can access its own one half of the register file directly and the other half can be accessed using cross paths. The two halves are named ‘A’ & ‘B’. Figure 2 is the block diagram for the core.

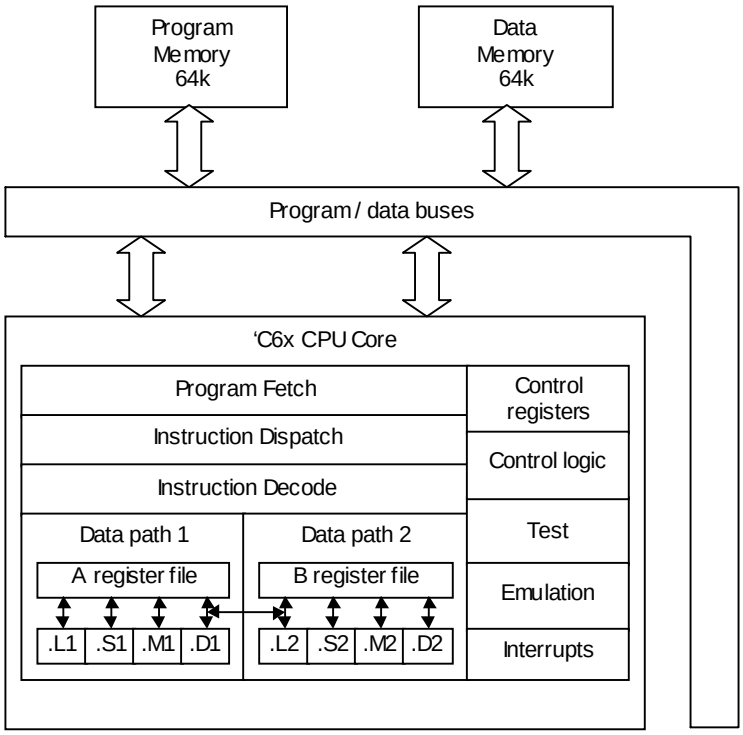

Figure 2 : Block Diagram of the 'C6x CPU core

The 'C6x devices come with on-chip program and data memory which can be configured to be a cache.

The CPU core contains a program fetch unit, instruction dispatch unit, instruction decode unit, thirty two 32-bit registers, two data paths each with four functional units, control registers, control logic, test, emulation, and interrupt logic.

The two data paths (A and B, also sometimes called data path 1 and data path 2), are the core logic where execution of instructions take place. Each data path has four functional units (.L, .S, .M and .D) and a register file with 16 registers. The basic arithmetic operations can be done in any of .L, .S or .D units. Branches are handled by the .S unit and multiplies are handled in the .M unit. Comparison operations are done in the .L unit and logical operations can be done either in the .L or .S units. A more detailed list of the operations performed in each function unit is given in table below.

Functional Unit	Operations performed in that unit
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations Leftmost 1 or 0 bit counting for 32 bits Normalization count for 32/40 bits 32-but logical operations
.S unit (.S1, .S2)	32-bit arithmetic and logical operations 32/40-bit shifts and 32-bit bit-field operations Branches Constant generation Register transfers to/from control register file (.S2 only)

.M unit (.M1, .M2)	16 x 16 bit multiply operations
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with a 5-bit or 15-bit constant offset

During execution, each functional unit reads from and writes to the register file in its own data path. However, the functional units .L1, .S1, .M1, .L2, .S2 and .M2 can access the register file of the opposite data path using cross paths called the 1X cross path and 2X cross path. Loads and stores are done by using the two load paths LD1 and LD2 and two store paths ST1 and ST2. For specific reference to instruction encoding for assignment to different functional units and registers, the author wishes to point to the technical brief [8] and the instruction set reference guide [9].

The addressing modes available are linear and circular addressing. All registers in the register file can perform linear addressing, but only eight (A4-7, B4-7) can perform circular addressing. All the load/store instructions use the AMR control register to determine the addressing mode for address calculation.

The standard on-chip memory that is part of the CPU core is a total of 128K bytes, 64K of which is program memory and the rest is data memory. The program memory can be configured to be a direct mapped cache with 32 byte line-sizes. When not used as a cache, the program memory returns 256 bits worth of data for every access, corresponding to the size of an instruction packet. Each instruction packet has eight 32-bit instructions. The data cache can be addressed to return 8, 16 or 32-bit worth of data on a read, or store the same sizes on a write. The data memory is divided into four banks of equal sizes and up to one memory access to each bank can be done in each cycle. Two accesses to the same bank on a single cycle will stall the CPU core.

Being a 8-way VLIW processor, each instruction packet (32 bytes) contains 8 instructions, which need not necessarily be executed together. The least significant bit of each 32-bit instruction, called the parallel execution bit, is used to encode whether the next instruction can be executed in parallel with the current instruction. This prevents the encoding of NO-Ops in the program binary and overcomes the disadvantage of binary bloating that VLIW machines could have.

Each instruction being execution can be a predicated instruction. The registers A1-2 and B0-2 are used as predicate registers if necessary. This removes a number of branches from the execution stream (not quantified in this report).

The ‘C6x processor pipeline is 11 stages in length. Four of the pipeline stages are used for instruction read, two for instruction dispatch and decode and the final five stages are used for instruction execution. Most instructions take only 1 execution stage to complete, with the exception of loads taking all five execution stages, stores taking three execution stages and multiplies taking two executions stages. Figure 3 shows the ‘C6x pipeline.

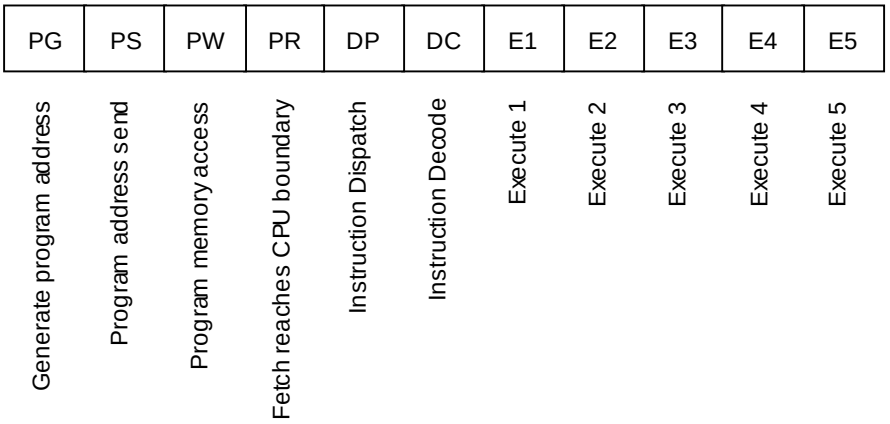

Figure 3 : The ‘C6x 11-stage pipeline

During the PG stage, the next address to read from is calculated and sent to the memory system during PS. Reading from the program memory takes place during PR and the 32 byte instruction packet that was read arrives in the core during the PR stage. Individual instructions in the instruction packet are dispatched to the corresponding functional units during the DP stage and they are decoded during the DC stage. Execution of the instructions in the functional units begins during the E1 stage and most instructions complete execution during this stage. Branching instructions are resolved during the E1 stage and if they are taken, the target address is calculated and during the next cycle, this target address will start the pipeline with the PS stage. Branching instructions have a 5 cycle delay between resolving and actual execution of the target. The 5 cycles may be filled in with instructions as branch delay slots. For more details on execution details of instructions, the authors wishes to point to the Instruction Set Reference Guide [9].

3. Installing and Compiling

The typical distribution package will be in the form of a tarred and gzipped file, c6xsim-1.0.tar.gz. Unpacking this package can be done by issuing the command :

tar xf - c6xsim-1.0.tar.gz | gunzip -c

This should create a directory c6xsim-1.0 and unpack files inside this directory. The following table lists the source files in the package with the content of each source file. In addition to these source files there are a number of headers files that contain constants, macro definitions and function prototypes.

Source file	Code in source file
c6000.c	Main simulator
cload.c	COFF loader. Reads in COFF binary (target ‘C62x) and store it in simulation memory
decode.c	Modules to decode and print out instructions being executed
coffload.c	Independent COFF binary reader. Can be used for verification of binaries
readmmap.c	Reads an initialization memory and register file map.
createmmap.c	Creates an initialization memory and register file map. Can

```
be used to initialize the memory to hold certain data that
might not be loaded by the program, to simulate a live
system.
```

The simulator has been written to be easily portable to any host system. It has been completely written in ANSI C. It has been verified with GCC on a number of platforms including x86-linux, x86-solaris, sparc-solaris, axp. The simulation tool is completely host endian independent. Although the 'C62x binary should be compiled to be big-endian.

Compiling the simulation tool requires an ANSI C compatible C compiler, eg. GCC. If the host system has **make** available, the included **Makefile** can be used to ease the compilation process, else compiling and linking c6000.c, cload.c, decode.c and readmmap.c, will give the simulator. The author recommends compiling with optimization turned on for reasonable simulation speed. On a dual Pentium II 333 Mhz host machine, with the simulator running on just one processor, the simulator executes about 100,000 machine cycles per second.

4. Using the simulation tool

The simulator can be run using the command :

./c6xsim <binary file> <simulator options>

The binary file is the compiled program to run on the 'C62x processor. This binary file format for this processor is in the COFF binary file format. The simulation tool has modules that can read in a COFF binary, decode the code in the binary and store the read in binary into simulated memory for the processor. The binary also has tags to identify the starting point of the program execution.

The simulator options available are :

-d *level of debug information to be output*
0 (default) : print no debug information
1 : print register file contents every machine cycle
2 : print pipeline trace
3 : print memory accesses
4 : print instructions in the binary at load time
these options can be used together as -d 1234 or any combination thereof

-i *input file for the binary that will run on the simulator. This can be an input for the binary. if none is given and if the binary needs an input, stdin is assumed*

-o *output file for the binary that will run on the simulator. Gives a good way to manage and separate output from the binary and output from the simulator*

-m *mmap file contains the memory and register file initialization values. This can be used to recreate a real system, which might have other data in memory*

5. Simulator Architecture

To save on runtime memory usage and reduce the number of times memory copies are done, the simulator executes pipeline stage by pipeline stage backwards. This helps save runtime memory usage, since two copies of the entire pipeline state need not be stored in memory. This also helps prevent copying from new pipeline state to old pipeline state. Going backwards in the pipeline execution, helps in preventing some complications that arise in the execution of branches. To recollect, branches are resolved and the

target address is calculated in the E1 pipeline stage. During the next machine cycle, the target address calculated is in the PS pipeline stage and not the PG pipeline stage. If we were to go from PG to E5 in sequential order, this would have been a complication. Reversing this order has helped preventing this complication.

The simulated processor is capable of addressing 2 ^ 32 (4G) bytes of memory, as defined by the 32-bit addresses. To prevent from allocating this much memory to the simulated processor, the memory is divided using segments. Each segment is 64K bytes in size and there are 64K possible segments that can be addressed. This 32-bit address is split into half and the higher half is used to identify the segment and the lower half is used as an offset inside the segment. If a previously untouched segment of memory was addressed, a 64K byte chunk of memory is allocated for the simulated processor. Pointers to the first byte of all allocated chunks of memory are kept track of.

The simulator is written in a modular fashion, and the author believes that it is fairy easy to understand the source code.

6. Other software

The generation of binary for the 'C62x processor needs a compiler that is capable of generating code for this architecture. The author is aware of only the compiler that is available from Texas Instruments for a price. Evaluation versions of this compiler can be downloaded from
http://www.ti.com/sc/docs/tools/dsp/6ccsfreetool.htm

7. Summary

The simulation tool for the TMS320C62x DSP processor has provided the author with valuable experience in building simulators modeling a complex hardware. It is hoped that this tool will come of use to anyone wanting to study the architectural and, application behavior on this processor.

8. References

[1] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Computer Architecture News, Vol. 25, No. 3, June 1997, pp. 13-25; also extended version Computer Sciences Tech. Report No. 1342, University of Wisconsin-Madison, June 1997

[2] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta, "Complete Computer Simulation: The SimOS Approach," In IEEE Parallel and Distributed Technology, Fall 1995.

[3] C. Bechem, J. Combs, N. Utamaphethai, B. Black, R.D. Shawn Blanton, and J.P. Shen, "An Integrated Functional Performace Simulator," IEEE MICRO, May/June 1999

[4] Why TI focuses on DSP, http://www.ti.com/corp/docs/investor/dsp/why.htm

[5] DSP Overview at Texas Instruments, http://www.ti.com/sc/docs/products/dsp/overview.htm

[6] Intel® Itanium™ Processor & IA-64 Architecture, the future of computing for servers and workstations, http://developer.intel.com/design/IA64/index.htm

[7] Jim Turley and Harri Hakkarainen, "TI's New 'C6x DSP Screams at 1,600 MIPS," Microprocessor Report, February 1997.

[8] TMS320C6000 Technical Brief, Texas Instruments Literature Number SPRU197D, February 1999.

[9] TMS320C6000 CPU and Instruction Set Reference Guide, Texas Instruments Literature Number SPRU189C, March 1999.