CMU DSP

The Carnegie Mellon Synthesizable Digital Signal Processor Core

Alpha Version Documentation 1.12 1999/06/10

Chris Inacio
inacio@ece.cmu.edu

The CMU DSP Team

June 10, 1999

Contents

Chapter 1

Introduction

The CMU DSP is a synthesizable digital signal processor (DSP) core written in Verilog. The CMU DSP is modeled after the Motorola DSP56002 processor, however, the CMU DSP does not include all of the instructions available to the DSP56002 processor, nor does it implement necessary functions such as interrupts. The CMU DSP is complex enough to be used as a real example for research, but is not capable of being used in a commercial environment.

The CMU DSP can be modeled and tested at various different levels of representation and as such is useful as a benchmark for electronic design automation tools[1]. The benchmark is oriented as a single design description with various components to make the entire core. This differs from previous benchmark work which contained many small test cases.

Currently the CMU DSP is best modeled at the structural Verilog level and can be functionally tested at that level and below. There is a partially functional behavioral instruction set architecture (ISA) description that we hope to finish and add to the overall framework for testing behavioral compiler technologies.

The CMU DSP is currently being used within various research groups and classes at Carnegie Mellon University. Some of the research using the CMU DSP is in the area of formal verification, and we would like to add that capability to the testing framework as it becomes available. Some Carnegie Mellon Universityclasses are also doing ATPG using a part of the CMU DSP core as their test case. As more information in this area becomes available we will include it with the generally CMU DSP distribution.

Future plans for the CMU DSP are for it to be used in many areas of current computer research. The core itself is relatively stable, and small bug fixes are the current largest change occurring to the repository. We do plan to add more information and scripts for testing the CMU DSP core in different ways as those resources become available. We also hope to track publications referencing CMU DSP core and list them to the general community of CMU DSP users.

The CMU DSP was developed by William Dougherty, Chris Inacio, Ben Klass, David Nagle, Andrew Ryan, Herman Schmit, Don Thomas, Ying-Fai Tong, and Nitzan Weinberg at Carnegie Mellon University, with the help of many others. The CMU DSP was developed under L. Richard Carley's low power research as a test case for that

project. The CMU DSP project was sponsored by the Defense Advanced Research Projects Agency under order number A564 and the National Science Foundation under grant number MIP90408457.

1.1 Getting and Installing the Distribution

We recommend reading through this document before attempting to install and use the CMU DSP for anything. The documentation explains many nuances in the structure of the repository that need to be understood before the CMU DSP can be used with any ease.

The distributions create the CMU DSP repository in a sibling directory, see Section 3.4 on page 11 for information about the directory structure, to the one they are expanded in. We do not recommend rearranging the directory structure present in the archive files. Many of the scripts written for CMU DSP require the directory structure to be present in order to operate correctly.

You should check for newer versions of this documentation and the distribution from http://www.ece.cmu.edu/ lowpower/benchmarks.html. Good luck the using the CMU DSP from the CMU DSP team!

Chapter 2

Architecture

The architecture of the CMU DSP is briefly reviewed here, but for a more complete description of the DSP56002 architecture, please see the Motorola DSP56000 Digital Signal Processor Family Manual. There are some areas in which in the architecture of the DSP56002 is not equivalent to the architecture of the CMU DSP , but for most purposes, and most signal processing algorithms, the architectures are the same.

The CMU DSP is different from the Motorola DSP56002 mostly in components outside of the core. Frefcmudsp-block-diagram-busses shows a block diagram of the entire CMU DSP design using the Duet Epoch tools. The Synopsys tools do not generate enything more than the core currently. The differences outside the core are significant compared to Motorola's DSP56002, and the access to the internal memories is limited to the External Bus. For more information an the CMU DSP design outside of the core see Section 2.5 on page 7.

2.1 Overview

The CMU DSP architecture is a Harvard architecture designed to support the streaming nature of multimedia data. Most digital signal processors are designed without data caches and a large amount of bus bandwidth to support this feature. The common design goal, achieved with the CMU DSP , is to be able to read in two data in a single cycle and perform a multiply-accumulate instruction on that data. The Harvard architecture employed in the CMU DSP facilitates this goal.

The CMU DSP 's Harvard architecture consists of three sets of separated data and address busses. Two of the busses, X and Y are data busses, while the third bus, P, is for program memory. In order to move data from one memory space to another special instructions in combination with the Bus Switch Unit unit can be employed. See Section 2.4.1 on page 7 for more information about moving data between address spaces.

Address generation for the CMU DSP is done primarily by two different units. The Address Generation Unit generates the addresses for the X and Y data busses. The Address Generation Unit is capable of doing complex addressing tasks independently

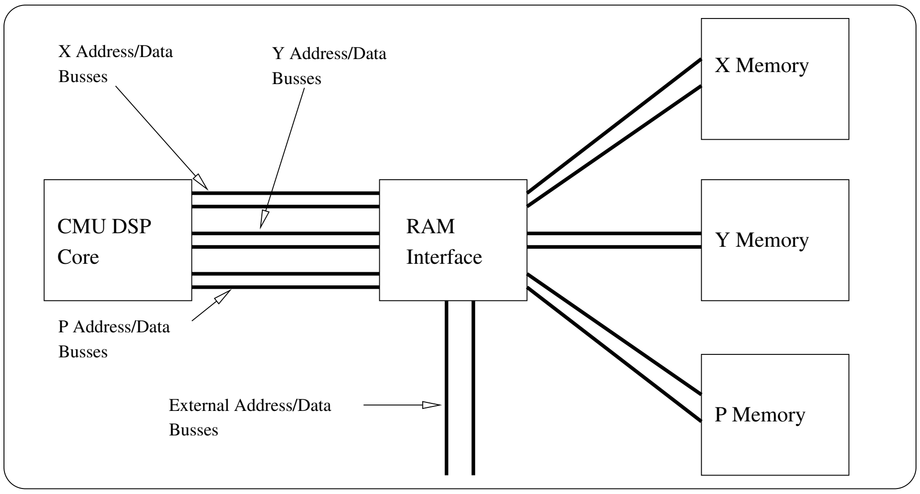

Figure 2.1: CMU DSP Block Diagram With Busses

of the Data Arthimetic and Logic Unit to maintain a datum per cycle per bus throughout program execution. For more information on the capabilites of the Address Generation Unit see Section 2.3 on page 6. The program address bus is generated by the Program Control Unit in most circumstances, jump instructions in which the target address doesn't fit into the instruction word is a notable exception. For more information about the Program Control Unit see Section 2.4.2 on page 7.

The data busses

2.2 Arithmetic and Data Logic Unit

The Data Arthimetic and Logic Unit (ALU) is the part which is generally of the most concern to the programmer. While the Address Generation Unit (AGU) is also a very programmable unit with many of its own adders, the ALU is responsible for the results that make the CMU DSP interesting.

The ALU contains the `X`, `Y`, `A`, and `B` registers along with the multiply-accumulate and adder units. For more information about this unit see [2]. For a list of supported instructions for the CMU DSP see Appendix A on page 23.

2.3 Address Generation Unit

The Address Generation Unit (AGU) would be the second major unit of programmer interest. The AGU generates the addresses for accessing the data memories. One important part of the AGU's address generation is its ability to operate independently of the ALU. The AGU can stream through the data memories in complex ways in order to support the algorithm design for the ALU. For more information about the AGU unit

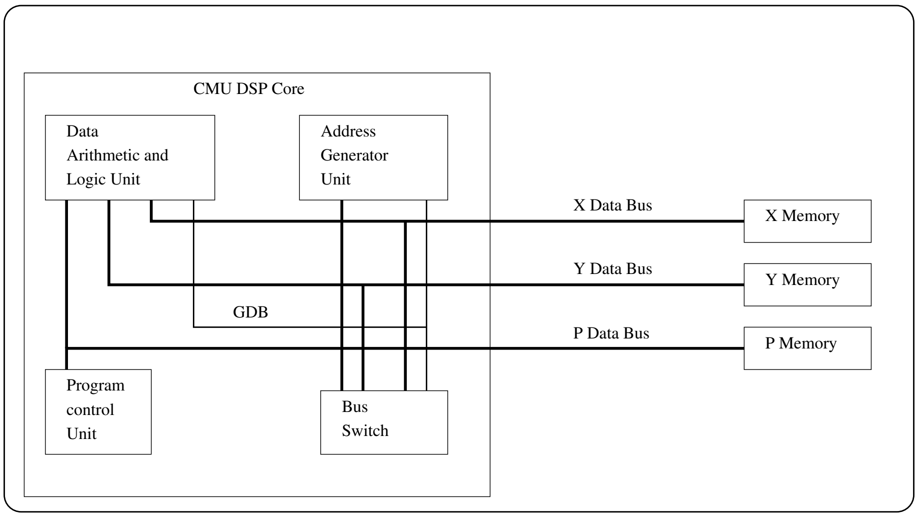

Figure 2.2: CMU DSP Databus Diagram

see [2]. For a list of supported instructions see for the CMU DSP see Appendix A on page 23.

2.4 Other Units

2.4.1 Bus Switch

2.4.2 Program Control Unit

This unit contains the program counter (PC) and the flag bits for controlling the CMU DSP . It is important to read the supported instructions with relation to use of this unit. For more information, please see [2].

2.5 Memory and External Interface

The memory and external interface on the CMU DSP is a very limitied design. The design goals were to provide a method to run test programs through the CMU DSP core to measure power consumption. In order to load a program into the on-chip memories, the core access to those memories must be disabled. Disabling the core for external access implies that the CMU DSP design is not suitable for too much more than running test programs.

Not finished:
In order to turn on the external port you must pull the `E_ACCESS` line low. This turns on the External Bus has the controller to the memories and stops the clock to the core. The core is also tri-stated and the core interface is tri-stated. The using the

MEM_SELECT lines, you select either the X, Y, or P memory. All memory timing and issues are the responsibility of the external user. The memories are asynchronous. When the chip is not in external access mode, the MEM_SELECT lines select which bus to snoop. If the bus is driven while the core is active, chip damage may result. Bus snooping allows the external interface to watch core access to any of the three memories. The user may change which bus to snoop during core execution.

Chapter 3

File Repository

The file repository holds all the Verilog source files, all of the scripts used to build and simulate, and all of the documentation available for the CMU DSP . This chapter will explain how to get access to the repository and the organization of the repository. This chapter will also attempt to explain the rational behind the structure that we used.

3.1 Availability

The CMU DSP is available over the Internet from http://www.ece.cmu.edu/ low-power/benchmarks.html. You will be able to download a compressed archive of the development tree. This archive will be updated periodically to include bug fixes and patches, however, the tree is in a stable form at this time. We do not expect to make any major changes to the tree format or the contents in the near future. The archive will be available in UNIX `tar/GZip` format. We may also make it available using the `zip` format.

3.2 Verilog Source Files

The source files for the CMU DSP are kept in a somewhat uniform directory structure. The files of most interest are the files in `cmudsp/structural`. These files contain the structural description of the CMU DSP .

In Figure 3.1 a partial diagram of the CMU DSP repository is shown. The figure displays part of the structural tree of the repository and goes further into the Data Arthimetic and Logic Unit part of the core. The Data Arthimetic and Logic Unit component of the CMU DSP has a directory structure the is very similar to the other components of the CMU DSP core. The descriptions below generally apply to all of the different core components.

Some properties to note are that below each major part of the core, (and the RAM Interface also,) exist a `before_syn` directory. This directory contains the Verilog source code that requires a compiler such as Synopsys to map the design to the appropriate gate library. Also, looking at the file names in the `before_syn` directory

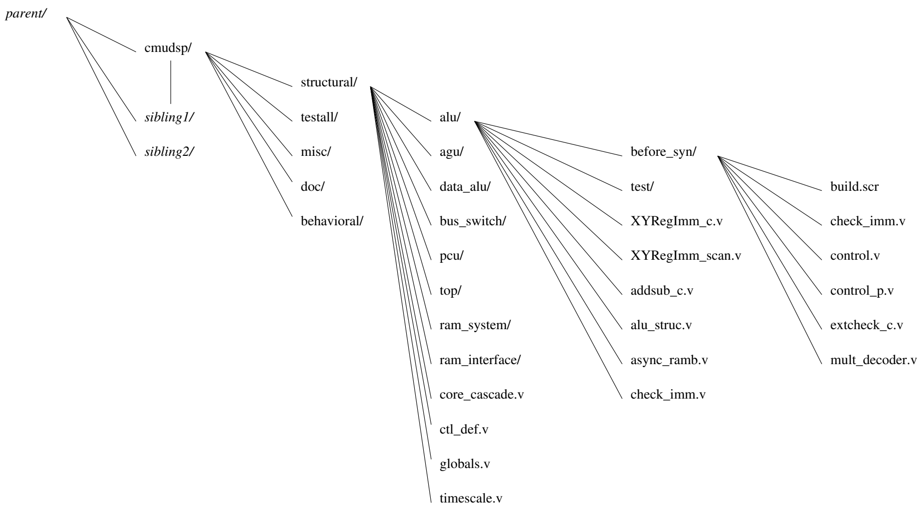

Figure 3.1: CMU DSP repository partial file tree

and the files in the directory above it, one will notice that there are duplicate names. The CMU DSP team keeps a current compiled version of each of the files in the `before_syn` directory in the core component directory above it. The gate library used as a target to compile the files is the Duet Epoch HP14B library. Building the structural files using Synopsys is aided by the file `build.scr` in the `before_syn` directory which is a command script for Synopsys 's `dc_shell` tool.

3.3 Documentation

The publicly available documentation for the CMU DSP is available in the `doc` subdirectory. The information contained in the `doc` directory is in one of three different formats. This documentation is produced using LaTeX2e and the source files and the processed output are available in the `doc` subdirectory. Some diagrams and informational files in the `doc` subdirectory were created using Adobe Framemaker. These files may or may not be available in other formats. Lastly, some files in the `doc` subdirectory are plain text files.

The text files are often the files that change the most with changes in the CMU DSP . Plain text files are the easiest to change and update as the small details in the design change. The files created using Framemaker are richer in content and usually contain diagrams or other graphics which are better produced in the desktop publishing tool. (Some in the CMU DSP may contend that everything should be done in Framemaker, but this author would disagree.) Lastly, LaTeX is used for this document. LaTeX allows easy decomposition for the sections and is easily used with `CVS`.

3.4 Test Scripts and Files

The scripts and test files are generally held in two directories below the `cmudsp` directory. Within the `testall` directory, the `runtest` script, see Section 4.5 on page 14, and its data files are kept in there. Briefly, `run-test` is used to do automated functional testing the CMU DSP by running programs on it. The other directory with current scripts is the `misc` directory. Please see Chapter 4 on page 13 for more information about other scripts.

The *sibling* directories in Figure 3.1 are noted because many of the scripts written by the CMU DSP team expect to be run from within a sibling directory. The scripts use relative paths to go up one directory and then down into the `cmudsp` directory and below. Some scripts have environment variables which can easily be edited to change this behavior, however, those scripts may call other tools which are not as easy to change. See Chapter 4 on page 13 for more information about the available scripts and their uses.

3.5 Miscellaneous Files

The Miscellaneous files section, the `misc` subdirectory, contains a script used in the processing of the Verilog files, the Verilog mode for EMACS, and the C source code to the utilities built for the CMU DSP project.

- **`cp2build`** — This scripts copies the Verilog source files from the repository, runs them through the `Verilog PreProcessor` and flattens the hierarchy for using a CAD tool on the design. See 4.2 on page 13 for documentation of `Verilog PreProcessor`.
- **verilog-mode.el** — This is the Verilog mode for the EMACS editor.
- **c-code** — This subdirectory contains the directories for the C source code to `coff_text_dump` , `reformat` , and `Verilog PreProcessor` tools. See Sections 4.4, 4.3, and 4.2 respectively for more information on pages 14, 14, and 13.

We add these tools to the path of the people working on the CMU DSP project. Some scripts, such as `cp2build` cannot be run from any directory, and must be run from a sibling directory of the `cmudsp` directory.

Chapter 4

Scripts and Tools

4.1 Introduction

The design flow of tools at Carnegie Mellon University includes tools from Cadence, Synopsys, and Duet. The CMU DSP uses at least Verilog-XL, Design Compiler, and Epoch from these companies respectively. Most of the scripts and tools were developed to aid the design group in building the chip and are designed for these tools. (There are some exceptions to make the system more flexbile.) Generally, if not stated otherwise, the scripts and tools assume these tools and your mileage may vary with other tools. We are willing to accept contributions for support of other environments, of course.

4.2 `Verilog PreProcessor`

The `Verilog PreProcessor` tool was created to support more than one design flow within the Verilog source files using Verilog's `` `ifdef ``, `` `define ``, `` `endif `` and `` `else `` preprocessor statements. This tool would not be necessary if the Synopsys tools supported the Verilog preprocessing statements. The code base was written first for the Duet Epoch design flow[1]. We developed this tool to support building the Synopsys version.

The `Verilog PreProcessor` tool is very limited in its syntax and capabilities. If you push it too hard **it will break.** The `Verilog PreProcessor` tool is not very flexible, and anyone who is interested may extend it to their heart's desire, but please send us back the patches. The preprocessor was designed to solve a single problem and be flexible enough to use reliably, and it does that.

The syntax of `Verilog PreProcessor` is very simple. It accepts defining "words" on the command line using the `-D` option, i.e. `-DEPOCH` would define `EPOCH` in the `Verilog PreProcessor` dictionary. You can put as many `-D`'s on the command line as you wish. The only other argument which `Verilog PreProcessor` takes is the name of input files. If an option given to `Verilog PreProcessor` doesn't start with `-D` then it is considered to be a Verilog file for preprocessing. Output of the preprocessed files is sent to `stdout`.

If you want a Verilog preprocessor that extends the capabilities of the macro language in Verilog, check the Surefire Verfication web site, in their Verilog resources section, `http://www.surefirev.com/resources.html`

4.3 reformat

The `reformat` command is used to convert the output from the Verilog simulations to be the same format as the output from the Motorola DSP Simulator. The output from the Verilog simulations does not include leadings zeros, "$" in front of values to indicate hexadecimal, etc. The `reformat` tool simply reads in the Verilog outputted file and reformats it so that the common utility `diff` may be used to compare the files between the Verilog simulations and the Motorola simulations.

`reformat` was designed for the output from the Cadence Verilog-XL Verilog simulator and may be sensitive to small changes in the output from other simulators.

The `reformat` tool gets called automatically from the `run-test` testing script. For more information on `run-test` see Section 4.5 on page 14.

4.4 coff_text_dump

`coff_text_dump` converts Motorola DSP COFF files into three text files representing the `X`, `Y`, and `P` memories. The CMU DSP uses a Harvard memory architecture with two data memories, `X` and `Y` and the program memory `P`. For more information on the architecture of CMU DSP see Chapter 2 on page 5

`coff_text_dump` outputs three files: `xmem_file`, `ymem_file`, and `pmem_file`. These files correspond to the memories `X`, `Y`, and `P`. The files are formatted so that the Verilog command `$readmemh` can be used to load the data in the files into memories within the Verilog simulator. For more information on this, read the source code file `test.v` in the `testall` directory. Briefly, the data from the three memory files is read into the behavioral memories defined in the file, if the external interface option is chosen, see Section 2.5 on page 7 for a descrption of the external interface and see Section 4.5 on page 14 for the external interface option, then the behavioral memories are used as drivers to the structural memories.

The `coff_text_dump` command works only on fully resolved, (linked,) Motorola DSP COFF files. It will give an error message if you try to run it on a file that is not fully resolved. In Motorola file extensions, `coff_text_dump` only works on files ending in `cld` and not on files ending in `cln`.

4.5 run-test

The `run-test` script is designed to automate the running of the functional test on the CMU DSP core. `run-test` calls the Verilog simulator to run a program on the core. At the end of the simulation, the memory system of the CMU DSP is written into a text file, `verilog_output_dump`. The output file is reformatted using `reformat`

, see Section 4.3 on page 14. The reformatted output file can then be compared using the standard Unix utility `diff` to see if the results are the same.[1]

Currently, `run-test` supports four test programs, all DSP kernel applications. The programs are two types of finite impulse response (FIR) filter, a Fast Fourier Transform (FFT), and an adaptive FIR using least means squares (LMS). The difference between the two FIR filters is in the number of taps that are used, the first version, (named simply FIR,) uses only four taps, and the second version uses sixty-four taps, named FIR64.

`run-test` also chooses a abstraction level at which to simulate the CMU DSP . There are currently three levels implemented, `rtl`, which is actually the behavioral level of the core only, `precascade`, which is the behavioral level of the entire design, using Verilog descriptions of the Duet Epoch library components used, and `postcascade`, which is a fully structural level description with back-annotated resistor-capacitor delay used.

In order to use `run-test` you must choose one and only one of the programs to simulate and one and only one level at which to simulate the system. For example, the simplest (and fastest,) simulation that you can do is:

```
run-test fir rtl
```

This simulation will run the FIR filter with four taps and simulate it on just the core using behavioral memory. (The four tap FIR filter was chosen due to its small size and relatively short simulation time, while exercising the most critical parts of the DSP for power estimation.)

`run-test` also takes another set of non-required options. Not all of these options work with every combination above and none are required. The first option `interface` cannot be used with the `rtl` level of simulation. The `interface` option tells the Verilog stimulus code to load the structural memory models using the external interface to the CMU DSP , see Section 2.5 on page 7 for more information. The `scan` option is a simple (and currently broken,) test to make sure that the scan chain designed into the CMU DSP actually works. (Currently, either the scan chain doesn't work, the test doesn't work, or both.) The `scan` option is supposed to simply scan out the contents of the scan chain, store them in a behavioral register, and scan them back in without changing them. I don't recommend trying this option unless you are very familiar with the inner construction of the CMU DSP until it is working again. The `shm` option creates a SHM wave file for use in debugging the CMU DSP . Lastly, the `cycle_dump` option creates a text output dump file also for debugging the CMU DSP .

`run-test` also works with the Model Technology Verilog compiler and simulator by Mentor Graphics. In order to use `run-test` with the Model Technology tools, you add the optional argument `modeltech`. The `modeltech` argument **must come before** the application argument in order to work properly.

[1] Actually, it isn't really that easy; there will always be differences in the output files because uninitialized memory in the Verilog simulator has the value `X` while uninitialized memory in the Motorola simulator has the value `0`. See some unwritten section for details on constructing your own tests and all the details.

4.6 cp2build

This copies the Verilog source code into another directory for building and runs the preprocessor on the files while coping them over. It can select Duet Epoch or Synopsys . The default for this tool, if no command line options are given is to copy the repository into a new subdirectory valled `verilog` from the current directory and put all of the Verilog source files in there after preprocessing them. This tool must be run from a sibling directory of the repository unless variables within the script are edited.

4.7 Building the Tools

Some of the tools, `cp2build` and `run-test` , are C-Shell scripts and do not require building. The other tools, `Verilog PreProcessor` , `reformat` , and `coff_text_dump` are written in C and require compiling before they can be used. The tools are built using GCC, GNU Make, and Flex. The C source code attempts to be very ANSI standard C, so it should be portable to most C compilers. Some of the more complicated make files may not work with other make utilities and the lexer descriptions may take advantage of Flex only features. All of the development tools mentioned, however, are freely available on the Internet if they are not already installed on your system.

In order to build the software, go into the repository into the `misc/c-code` subdirectory. Go into each subdirectory, `coff_dumper`, `qdvpp`, and `reformat` and type `make` in each. The only program that will build without warnings is the `reformat` program.

Chapter 5

CMU DSP Testing

5.1 Introduction

The testing plan originally formulated for the CMU DSP was designed to catch design errors in our architecture. The design group decided to achieve this goal by simulating our design at a functional level and running programs on the design. The results of running programs on our design would be compared against the results of running the same program on the Motorola simulator. Our development plan included using commercial cell libraries an large degrees of automation, so testing for circuit level errors was not a priority for us. In the case of a circuit level error, we had included in the design a scan chain which would hopefully enable us to determine where in the processor the design or circuit flaw was in order to correct the design when the chip was produced.

We have learned not to trust our design tools that closely. Part of our testing plan had always been to simulate the entire chip for a few cycles using a SPICE level simulation. We had hoped to use an accelerated SPICE simulator for digital circuits, but learned that one should be very careful in which simulator one trusts. We found various errors in the circuit level, and went through great effort to eliminate them.

What we are currently providing as a test suite is the functional testing. We are not sure what the coverage of our test suite is. We don't believe the suite to have great coverage since most of applications were chosen because they are digital signal processing kernels. There are some sanity tests which were not fully developed into the test suite written by various members of the team while implementing their components. Hopefully, in the future, more complete functional coverage will be included.

5.2 Writing a new functional test

5.2.1 Overview

There are a few steps required in creating a new test and adding it into the test suite so that it can be automatically tested. The first step, however, is to write a program for the Motorola 56002 in either assembly or C language.[1] Using the Motorola tools, (and their documentation,) create a linked DSP COFF file. This is generally accomplished by writing an assembly file, compiling it with `asm56000` and linking it with `dsplnk`. At this point, you can simulate your program using the Motorola 56000 Simulator and capture the results. Using the Carnegie Mellon University provided tool, `coff_text_dump` (see Section 4.4 on page 14), you can create the memory files which the Verilog simulation will read in to initialize the memory. (For more information on simulating the core using Verilog see Section 4.5 on page 14.) After simulating the core using Verilog, you will need to manually compare the `diff` results of the memory dumps generated by the Motorola Simulator versus the Verilog simulation. The primary reason this is necessary is that unitialized memory in the Verilog simulation are noted as `x's` while in the Motorola Simulator unitiailized values are all zeroed. After verifying the the differences are all neglible, you create a standard diff file, the baseline, which can be used in the future instead of manual comparing the results. In order to add the new program to the testing system, the script will require some small edits for the new program.

5.2.2 Writing a new program

This topic is really beyond the detail of this manual. All of the example programs provided were written using a simple text editor in assembly language and the Motorola 56000 assembler. For more information on writing programs for the 56002 and the CMU DSP please see the Motorola assembly documentation.[4] There are “conventions” that the Carnegie Mellon University team has used in their programs that you may like to follow. (It will be easier to use the hints that follow if you do use these conventions.) The general outline of a program usually looks something like the following:

```
org     p:$0
        jmp     start

        org     p:$50
start

         ⋮

endp    jmp     endp
        nop
        nop
        nop
```

5.2.3 Motorola Simulation Traces

In order to compare the results of the CMU DSP against the Motorola simulator, we must dump the initial and final state of the memory while using the Motorola sim-

[1] While Motorola's C compiler should work, the CMU team has never tried to run a C compiled program on the CMU DSP .

ulator. This is usually accomplished using small command scripts for the Motorola Simulator.[3] This example script to capture the memory dumps and simulate the program:

```
1   load firt_test.cld
2
3   log s new-fir64-test-input.dump
4   display x:$0..$3ff
5   display y:$0..$3ff
6   log off
7
8
9   break endp
10  log s motsim_dump.dump
11  trace 90000 H
12  step
13  step
14  step
15  log off
16
17  log s new-fir64-test-output.dump
18  display x:$0..$3ff
19  display y:$0..$3ff
20  log off
```

Line 1 in the example loads in the program, in this case called `firt_test.cld`, a version of the 64-tap FIR filter.

Starting on line 3 are the simulator commands that create a memory dump of the initial state of the memory. The range of the memory dump captures only the first 1024 words and only the data memory. We do not compare the program memory since the instruction formats are different and we only capture the amount of memory equal to the amount the CMU DSP has available to it.

Line 9 provides some more interesting commands to use in order to simulate the program. In order to not have to know how many cycles the program will run for in order to build these scripts we set a break point using the `break` command. By convention, the last line in our programs is labeled `endp`. We usually log, line 10, the trace of the entire program which can aid in debugging the core or the program, but it does take up quite a bit of disk space. Next we use the `trace` command and 90000 cycles should be enough for any program you want to run on a Verilog simulator. It is important to use the `H` parameter, otherwise `trace` won't stop at the break point. We step the program three more times, lines 12–13, to ensure that the pipeline has cleared and turn the log file off.

The end of the script, lines 17–20, are almost the same as lines 3–6, and capture the data memory at the end of the simulation.

Chapter 6

Building CMU DSP using Synopsys

6.1 Introduction

The CMU DSP was original built using Duet Epoch as the main underlying tool, however, more recently that tool's future has come into question and the Carnegie Mellon University team has decided to make Synopsys the standard way of building the core. Futher, by using Synopsys and a cell library developed at Carnegie Mellon University, we are able to distribute full geometry and Spice files for the CMU DSP without worry of licensing issues.

The CMU DSP , using the `Verilog PreProcessor` is still portable to other environments, but the most complete detail is using the source files to build the core is given to the Synopsys design flow.

6.2 Creating a Synopsys Work Area

Like most testing and other work areas related to the CMU DSP core, this must also be created in a sibling directory to the repository. Create a sibling directory in order to build a flattened preprocessed version of the CMU DSP source code tree. (You will have to have compiled `Verilog PreProcessor` previously for this to work correctly. See Section 4.7 on page 16 for more information on building the tools.)

When in your working directory, copy the script file `cp2build` from the repository `misc` directory. Then simply execute the `cp2build` script. For more information on the `cp2build` script see Section 4.6 on page 16. The flattened hierarchy of Verilog files preprocessed for Synopsys DesignWare will be in a subdirectory named `verilog`.

Appendix A

Supported Instructions

The CMU DSP does not support all of the instructions supported by the Motorola DSP56002. This appendix tries to summarize the instructions which are supported.[1]

A.1 ALU Instructions

The supported instructions include: `ADD, ADDL, ADDR, AND, ASL, ASR, CLR, EOR, LSL, LSR, MAC, MACR, MPY, MPYR, NOP, NOT, OR, RND, ROL, ROR, SUB, SUBL, SUBR, TFR`. Both the long version of the add and subtract are also supported. The unsupported instructions include: `ABS, ADC, ANDI, CMP, CMPM, DEC, DIV, INC, NEG, NORM, ORI, SBC,` T**cc**, `TST`. The `move` instructions supported are the ALU following:

`MOVE X: Y:`	full support
`MOVE X:`	into `X` memory: `x0`, `x1`, `y0`, `y1`, `a`, `b`
	into registers: `x0`, `x1`, `y0`, `y1`, `a1`, `b1`, `a`, `b`
`MOVE Y:`	into `Y` memory: `x0`, `x1`, `y0`, `y1`, `a`, `b`
	into registers: `x0`, `x1`, `y0`, `y1`, `a1`, `b1`, `a`, `b`
`MOVE R`	from AGU to ALU: `x0`, `x1`, `y0`, `y1`, `a1`, `b1`, `a`, `b`
	from ALU to AGU: `x0`, `x1`, `y0`, `y1`, `a1`, `b1`, `a`, `b`
`MOVE I`	into registers: `x0`, `x1`, `y0`, `y1`

[1] This list of instructions may not be totally complete, it is however, to the best of our knowledge the set of instructions that is working and tested.

Bibliography

[1] Chris Inacio, Herman Schmidt, David Nagle, Andrew Ryan, Ying-Fai Tong, Don Thomas, and Ben Klass. CMU-DSP Vertical Benchmarks for CAD. In *DAC-36*, June 1999.

[2] Motorola, Inc. *DSP56000 Digital Signal Processor Family Manual*, 1995.

[3] Motorola, Inc. *Motorola DSP Simulator Reference Manual*, 1995.

[4] Motorola, Inc. *Motorola DSP Assembler Reference Manual*, 1996.