Debugger For Hw/Sw Cosimulation Environment

A Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Technology

by

Mayank Gupta

to the

Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
January, 1999

Certificate

Certified that the work contained in the report entitled "*Debugger For Hw/Sw Cosimulation Environment*", by *Mayank Gupta*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Rajat Moona

January, 1999

Preface

The developers of modern embedded systems feel the need of automatic development tools such as Functional Simulator, Timing Simulator, Cross compiler, Disassembler etc. for faster development. These tools help them in verifying their design early in the design cycle and for studying various design tradeoffs. Such tools put together can form an integrated environment, where the designers can specify their designs in a high level language and use these tools for verification and exploring various design parameters. These tools are also heavily used by researchers for specifying next generation Instruction Sets and microarchitectures.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Rajat Moona for his guidance, motivation and invaluable suggestions at all stages of this project. I would like to thank him for his enthusiasm throughout the discussions I had with him during the course of this project.

Apart from my project supervisor, I would like to convey special thanks to Souvik Basu and A. R. Rajiv for the help they have given during the course of this project. I would also like to thank my friends for the support they have given to me.

Contents

Chapter 1

Introduction

The developers of modern embedded systems feel the need of automatic development tools such as Functional Simulator, Cross compiler, Disassembler, Hardware Simulator etc. for faster development. These tools help them in verifying their design early in the design cycle and for studying various design tradeoffs. Such tools put together can form an integrated environment, where the designers can specify their designs in a high level language and use these tools for verification and exploring various design parameters. These tools are also heavily used by researchers for specifying next generation Instruction Sets and microarchitectures.

In this work we have developed an enhanced version of *Retargetable Functional Simulator* for our environment which uses *Sim_nML* as the language for describing processor models (Instruction Set). Sim_nML is an extension of nML machine description formalism and is powerful enough to specify instruction set of a processor. The earlier version of *Retargetable Functional Simulator* was developed by Y. Subash Chandra [1] as part of his mtech thesis. The *Functional Simulator* was earlier tested only Power-PC instruction set but now we have also tested it for ARM instruction set. For this we have specified the ARM Instruction Set in Sim_nML. We have also added a Debugger to this system which helps in tracing the instructions being executed and is quite helpful for debugging at the lowest level. As part of this work, we have integrated this *Functional Simulator* with TIPSIM (Hardware Simulation

Environment) to develop a prototype of a HW/SW simulation environment.

1.1 Organization of report

The rest of the report is organized as follows. Chapter 2 gives an overview of the processor description language. Chapter 3 gives an overview of the *Functional Simulator* and describes the difference between the present and the previous version of the *Functional Simulator*. In Chapter 4 we give an overview of the architecture and Sim_nML specification of ARM Instrution set. Chapter 5 discusses the debugger commands and chapter 6 gives an overview of the TIPSIM environment.

Chapter 2

Overview of Processor Description Language

This chapter gives an overview of the Sim_nML language used for processor description and the Intermediate Representation (IR) of this description.

2.1 Sim_nML

In Sim_nML the processor is described at instruction level. The instruction set is enumerated as an attribute grammar in a tree hierarchy capturing the semantics of the instructions at different levels of the hierarchy depending on the class of instructions.

Sim_nML defines a top level node called instruction and two kinds of productions or-rule, whose syntax is as follows:

op n0 = n1 — n2 — n3 ....

and and-rule whose syntax is as follows

op n0 (p1 : t1, p2 : t2, ...)

a1 = e1 a2 = e2 ...

where each ni is a non-terminal and each ti is a terminal. Each ai is a attribute and ei is it's corresponding definition. pi's are the parameters used in the attribute definitions.

A sample Sim_nML specification of some hypothetical processor is given as follows.

```
mode REG_INDIRECT ( i : card ( 5 ) ) = R [ i ]

op instruction ( x : instruction_action )
syntax = format (%s, x.syntax )
image = format ( %s, x.image )
action = {
   PC = PC + 1;
   x.action;
}

op instruction_action = add sub

op add ( x : REG_INDIRECT )
syntax = format ( add %s, x.syntax )
image = format ( 100000%s, x.image )
action = {AC = AC + x; }

op sub ( x : REG_INDIRECT )
syntax = format ( sub %s, x.syntax )
image = format ( 100001%s, x.image )
action = {AC = AC + x; }
```

The *syntax* attribute gives the assembly language syntax of the instruction. The *image* attribute defines the binary image of the instruction. The *action* attribute

gives the semantic action associated with the instruction. *Mode* rule specifies the addressing modes for the processor.

2.2 Intermediate Representation(IR)

The information in the Sim-nML specification is captured into a set of tables in IR. Each table consists of fixed or variable size records representing a particular type of information. In this way it is easy to extract out the information needed and reduces the work of the tools which intend to use Sim_nML for processor specification. The format of this IR has changed after the development of old $Fsimg$ and so the old version of the $Fsimg$ is not compatible with the new version of IR. The new version of $Fsimg$ is compatible with this new version of IR.

2.3 Differnce between old and new version of IR

As shown in Figure 1 the old version of IR generator had essentialy three parts. The first part parses the Sim_nML specification file and does type checking, the next part flattens the Sim_nML hierarchy based on the *image* and the *syntax* attribute and the third part generates the appropriate tables. By flattening, we mean to enumerate all possible instructions by traversing the Sim_nML hierarchy. In the new version of IR generator, the middle portion has been completely eliminated and the corresponding changes has been done in the format of the generated tables of IR.

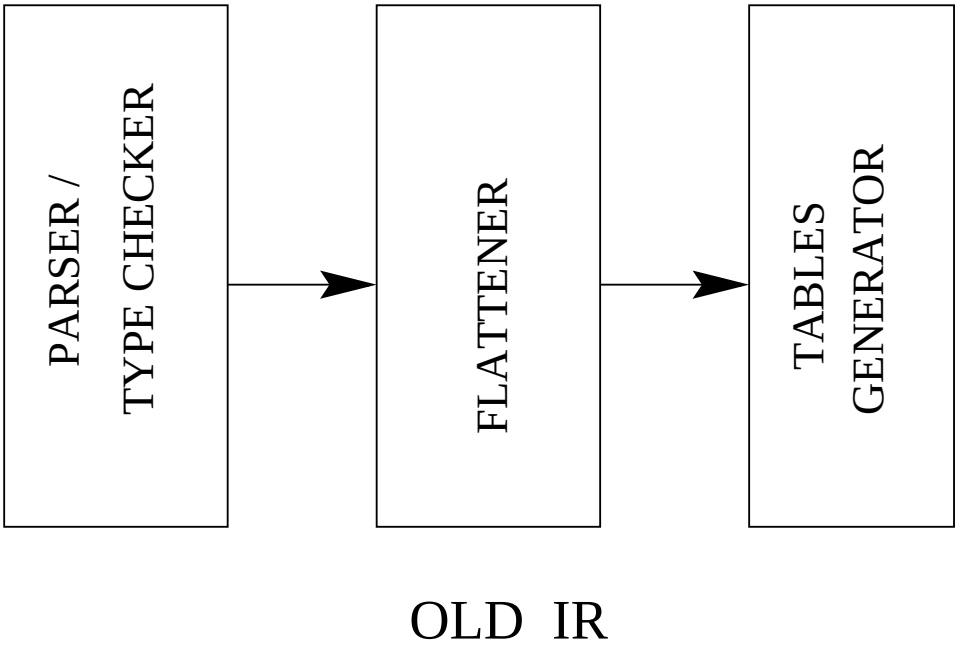

Figure 1: Block Diagram for Old IR Generator

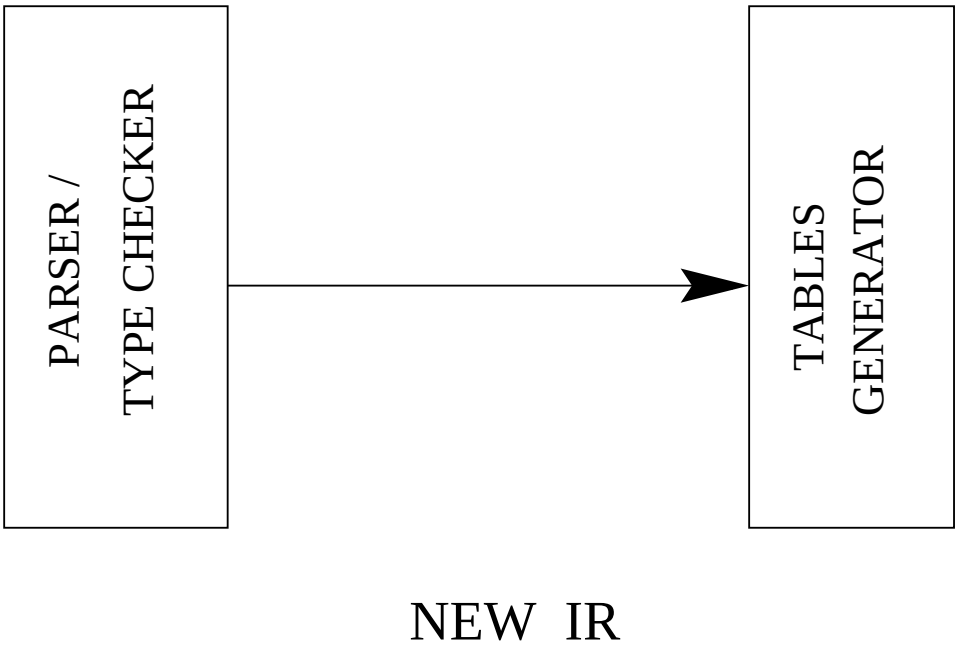

Figure 2: Block Diagram for New IR Generator

Chapter 3

Retargetable Functional Simulator(Fsimg)

In the past, functional simulators (or ISS) were made for a specific processor architecture. However, with ever increasing special purpose processors and the need to specify Instruction Set of processors which have not been realised in hardware, a strong need is being felt for retargetable functional simulator generator. Such a tool can be used to generate Instruction Set Simulator(ISS) for a particular processor. The idea behind such a tool is to use processor models for generating processor specific part of ISS. Figure 3 shows the block diagram of the new Fsimg and subsequent subsections describe the functionality of each part the Fsimg.

3.1 Overview of Functional Simulation(Fsim) Generation

Fsimg first reads the IR file to construct the tables. These tables capture the Sim_nML description of the processor's instruction set. The Fsimg then finds all the instructions by flattening the *image* attribute of the Sim_nML description. The *image* attribute captures the binary coding of instructions. This part of the

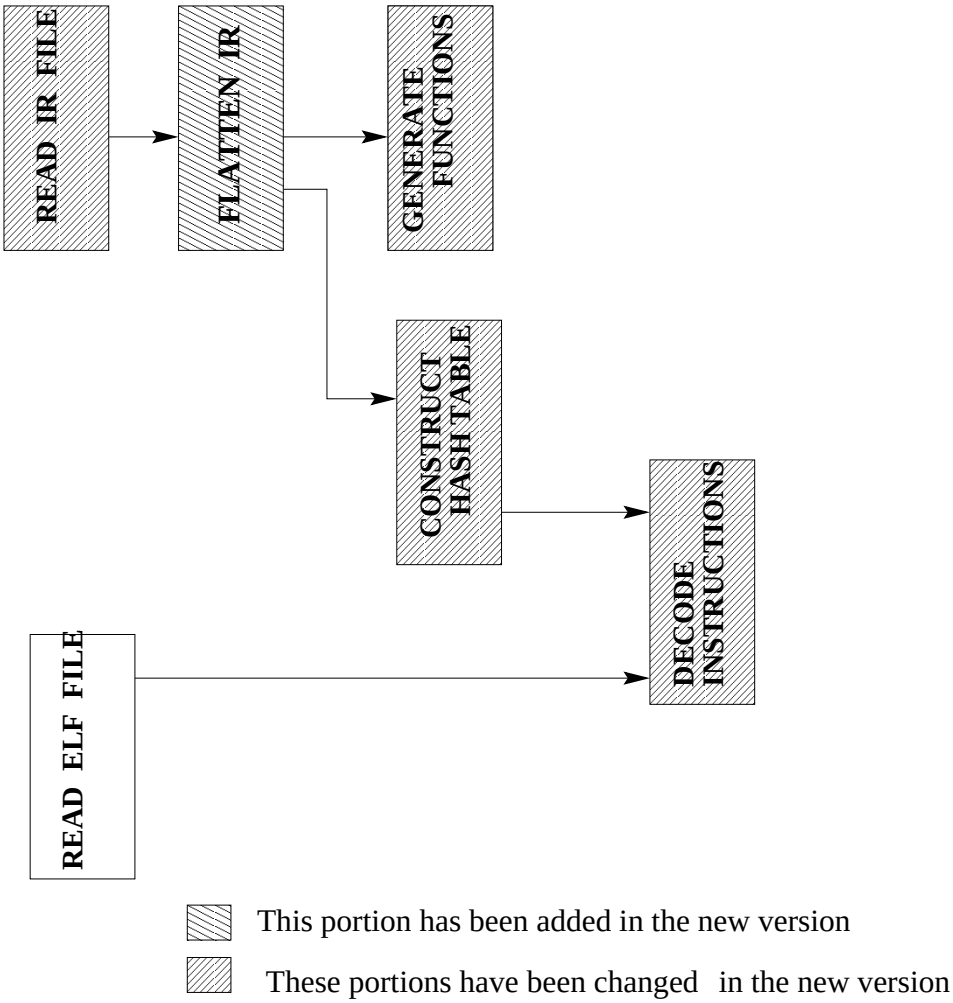

Figure 3: Block Diagram Of Retargetable Functional Simulator

Fsimg was not present in the earlier description since the *image* attribute was already flattened in the IR as described in the previous chapter. A hash table is then constructed from the flattened images of the instructions. This hash table helps is quickly decoding the instructions of the ELF file.

After this, the action attribute of the Sim-nML specification which captures the semantics of the instructions is flattened. For this the Fsimg converts all instructions semantics into respective functions by flattening the hierarchical description of action attribute. The Fsimg then decodes the instructions from the program using

the hash table, extracts the parameters for the instruction and generates a call to the respective function with these parameters. All these calls to functions are captured into a table called function-pointer-table whose entries are basically a set of parameters and a function pointer pointing to the respective function. The entries in this table are in the order of the instructions in the program. The simulation starts by calling the function for the first instruction along with its parameters. The called function returns the index of the next instruction into the function-pointer-table. In this way simulation continues till the program is terminated. Along with this table Fsimg generates data structures for the memory, registers and other memory elements in the processor, and a driving routine for the simulation which initializes the memory and registers. The driving routine also calls the first instruction of the program.

3.2 Structure of Functional Simulator(Fsim)

The functional simulator generated by the *RetargetableFunctionalSimulator* has the following five components contained in separate files as shown in Figure 4 .

1. A set of functions one for each instruction in the processor description in Instr.c, Instr1.c ... .
2. A function-pointer-table corresponding to instructions in the program in Funcs.c .
3. The memory image of the program in memory.img . This corresponds to the data portion in the ELF file.
4. Data structures for registers and other memory elements in Types.h and Vars.h.
5. The driving routine in Fsim.c .
6. The function for handling calls to dynamic functions in dyn.c and dyn.h .

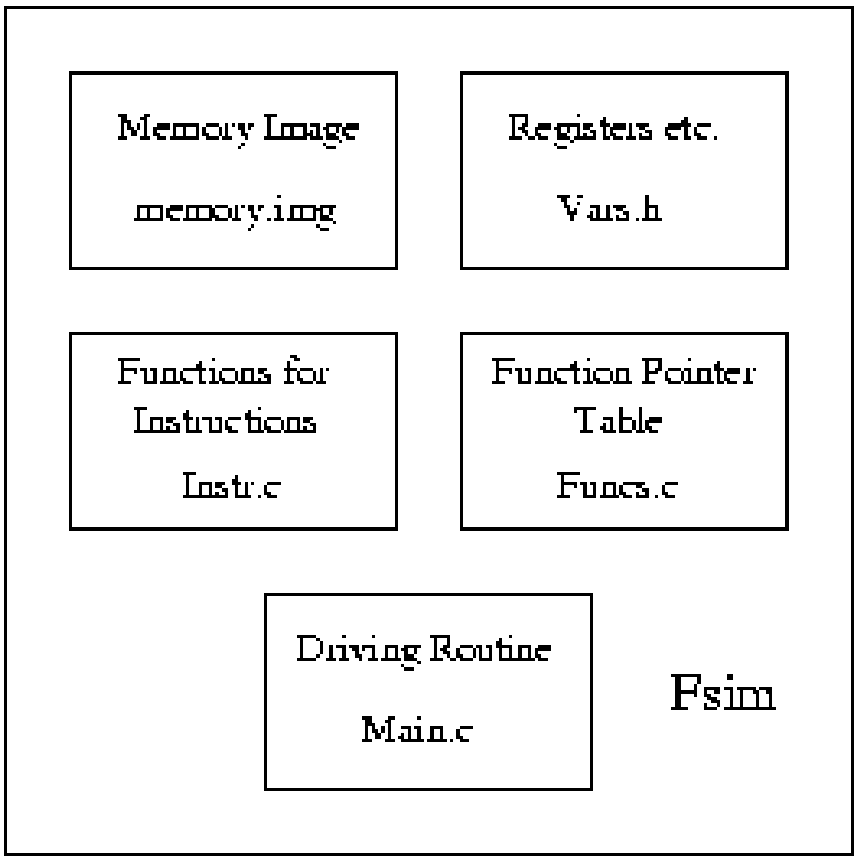

Figure 4: Fsim Components

The function-pointer-table is an array of structure whose members are an array of parameters and a pointer to a function. The C declaration of function-pointer-table is as follows.

```
struct func_ptr {
    uint64 p [MAX_PARAMS];
    int (*func)(uint64 *);
};

struct func_ptr Func_Pointers [MAX_POINTERS] = {
    {{13, 0, 388}, Fun38},
```

```
        {{13, 13, 41916}, Fun146},
    };
```

The function pointed to by the function pointer takes a pointer to the parameters and returns the index of the next instruction. When the simulation starts, the driver routine initializes the conceptual program counter (PC), stack pointer (SP), link register (LR) and memory. At this point PC points to the first instruction of the program and calls the respective function with parameters. The function called performs the semantic action associated with the instruction and returns the next instruction index to the driver routine. The driver routine uses this index to call the next function. In this way the instructions get executed (simulated) until the program terminates. The code for the driver routine looks like the following.

while((index = Func_Pointers[index].func(Func_Pointers[index].p)) != -1);

3.3 Detailed Algorithim ForCode Generation

3.3.1 Extracting instructions and Flattening

The first step of the functional simulator is to read the IR tables into the internal data structures. After this the image attribute has to be flattened in order to get the binary code of the instructions. It is used for decoding the instruction. This is heirarically spread over the specification tree. The earlier IR used to stored the images of all possible instructions after flattening the above heirarcy while the present IR stores this information as specified in the Sim_nML specifications without flattening.

So the present Fsimg obtains the images for all instructions by flattening the image attribute specified in IR, which was earlier done by the IR generator. The image attribute is flattened in the bottom up fashion which means that the $Rules(AndandOr)$ at the lower levels of the heirarchy are flattened before the $Rules$ at the upper level. First of all the Or rules are flattened and then the And rules are flattened. While this flattening is done additional information is generated for each instruction which

indentifies a unique path in the heirarchy for that instruction. This additional information is used by the *ActionFlatener* for generating functions for each instruction and for mapping the value supplied by the instructions in the Elf file to the corresponding parameters in the Sim_nML heirarchy. The structures used for storing this additional information is as follows :

```
typedef struct{
  uint32 depth;
  uint32 list_and[MAX_DEPTH];
  uint32 list_sub[MAX_DEPTH];

  uint32 len[MAX_PARAMS+1];
  int32 and_rule[MAX_PARAMS+1];
  int32 sub_rule[MAX_PARAMS+1];
}dotExpr_Ent;

typedef struct{
  uint32 depth;
  uint32 list_and[MAX_DEPTH];
  uint32 list_sub[MAX_DEPTH];

  uint32 no_params;
  int32 X[MAX_PARAMS];
  int32 Y[MAX_PARAMS];
  int32 type[MAX_PARAMS];
  int32 val1[MAX_PARAMS];
  int32 val2[MAX_PARAMS];
}dotPart_Ent;

typedef struct{
  uint32 msb;
  uint32 lsb;
```

```
  uint32 no_flatrules;
  char **images;
  dotPart_Ent *dot_expr;

  uint32 *index;
}andPart_Tbl_Ent;


typedef struct{
  bool IsFlat;

  uint32 no_flatrules; // No of sub-rules in the flattened image
  char **images;  // Flattened images
  dotExpr_Ent *dot_expr; // Inorder traversal of heirarchy for the corresponding imag

  uint32 no_params;
  int32 *X;
  int32 *Y;
  int32 *type;
  int32 *val1;
  int32 *val2;

//used while flattening
  uint32 no_andParams;
  int32 len[MAX_PARAMS + 1];

// used if a part of the image of this AND rule is referred
// by some higher level rule in its image using bit selection operator.
  int32 no_parts;
  andPart_Tbl_Ent part[MAX_PART];
```

```
}andFlat_Tbl_Ent;
```

3.3.2 Hashing

After this image masks are computed for image of each instruction. Image mask is basically a bit string that has ones for the bit positions representing instruction and zeros in the parameter bit positions. In order to decode an instruction, we and it bit-wise with each instruction mask and compare the result with the image attribute in the IR. This process of decoding is, however time consuming. To improve the decoding performance the images are hashed into a hash table.

First a global mask is computed by bit-wise anding all instruction masks. The global mask therefore represents the opcode field of the instructions. First level hashing is done based on this opcode field. However it is not very useful because instructions may not have distributed evenly to all the buckets. It may result in one bucket having large number of instructions while some other buckets having no instructions. For this reason instructions in each bucket are further hashed based on the remaining fixed fields of those instructions. This comes from the observation that when we hash on opcode all instruction of particular type say integer instructions gets hashed to same bucket. Now these instruction have additional fields to identify different instructions amongst themselves. These fields are called sub-opcodes. In this way hashing is done several levels until single instruction is hashed to a bucket.

3.3.3 Generating Functions for instructions (Action Flattening)

After instruction hashing, Fsimg generates the functions for the instructions in the description. The semantics of an instruction is captured in the action attribute, which is hierarchically spread. Starting at the top node till the leaf node, the definition in the action attributes is captured as a C function. All the attribute definitions are available in the PREFIX ATTRIBUTE DEFINITION table in the IR. The definitions are in the prefix notation which are converted in to the infix notation during

the code generation. The generated functions take pointer to parameters and returns the index of next instruction into the function-pointer-table.

3.3.4 Instruction Decoding and Function Pointer Table Generation

The Fsimg reads the required information from the given program which is in ELF format into the internal data structures. Depending on the processor type, number of function-pointer-table entries are calculated. If it is a processor with fixed instruction length, then the calculation is as follows.

```
no-of-entries = text-section-size-in-bytes / instruction-length
```

If it is a processor with variable instruction length, then the calculation is as follows.

```
no-of-entries = text-section-size-in-bytes
```

Now the decoding of the instructions in the program is done using the hash table created earlier. Once a instruction is recognized the operand values are extracted from the instruction and a function-pointer-table entry is generated with these values and the corresponding function for the instruction. In case of a processor with variable instruction length, number of entries generated are equal to the length of the instruction in bytes. In this manner all the instructions in all text sections of the program are decoded and function pointer table is generated. If an instruction gets unrecognized then a dummy entry is created in that position of the function-pointer-table. If the control reaches this entry during the simulation, then it generates an error message and simply returns the index of next instruction to the driving routine. This may lead to incorrect results and unpredicted behaviour of the Fsim. To avoid this the specification has to cover all the instructions needed for running the program.

3.3.5 Generation of Types and Memory Image

The Sim-nML types are converted into corresponding C types, like unsigned int for card and int for int etc. But the problem comes with the sizes of these declarations. Sim-nML allows the declaration of variables of arbitrary bit sizes. Consider the following Sim-nML declaration.

```
mem TEMP [ 1 , int ( 4 ) ]
```

This declares TEMP as a memory location of type integer and size 4 bits. We have to allocate exactly 4 bits for the correctness of the value held by this location. For this the C feature of bit-fields inside the structure declaration is used. For the above declaration the code generated is as follows.

```
typedef char int8;
typedef struct {
        int8 val:4;
}Int4;

Int4 TEMP;
```

Whenever TEMP occurs in any attribute definition, TEMP.val is generated in that place. Thus whenever a variable is declared which is not a multiple of 8 bits, nearest C-data structure larger than the one being used in Sim-nML, for example, a 12 bit variable in Sim-nML is declared using int16 type.

Fsimg composes the memory image for Fsim by combining all the data sections of the program and is written to a file. When Fsim starts it loads this memory image in to its memory. All memory references are redirected relative to the location where it is loaded.

Finally code for the driver routine is generated which consists of the code that initializes the PC, SP, LR and memory and the code for the simulation as we have seen earlier.

Chapter 4

ARM Specification in Sim_nML

In this chapter we present a brief overview of ARM architecture [2] and discuss the ARM specifications in Sim-nML.

4.1 Overview of Architecture

The ARM is a RISC processor, and have the following features :

- A large uniform register file.
- A load-store architecture.
- Uniform and fixed length instruction fields.

4.1.1 Registers

ARM has thirty-one, 32-bit registers. But at any one time, only sixteen are visible. The other registers are used to speed up exception processing. All register specifiers in the ARM instructions can address any of the 16 registers. The sixteenth register(i.e R15) is used as the program counter and fifteenth(R14) register is used as the link register. Moreover, R13 is generally treated as stack pointer. Apart from this, there is a status register called CPSR which stores the condition code, Processor Mode and Interrupt enable bits.

4.1.2 Instruction Set

The ARM instruction set can be divided into four broad classes of instruction:

- dataprocessing
- branch
- load and store
- multiply

All ARM instructions may be conditionally executed depending upon the 16 condition conditions. Also there are a number of addressing modes for each type of instructions.

4.2 Overview of Specification

The description of instruction heirarchy is as follows. The top level node is instruction. The instructions are divided depending on their instruction type as described in the previous section. Around 8144 instructions have been specified for the ARM instruction set which covers all the user level instructions.

Chapter 5

Debugger

On entering the debugging mode, the debugger displays a command line as shown below :

```
Executed last instruction <Opcode of Instruction> at address <address>
Type ‘‘help’’ to see the commands
Type ‘‘quit’’ to exit from the debugger
```

The debugger supports single stepping and can be used to set breakpoints. It can also be used to display the present contents of the registers and memory memory locations.

The list of commands supported by the debugger and their functionality is described below :

```
step [N]            -  This command executes next N instructions and then
                       enters the debuging mode. The default value of N is 1.

break <address>     -  This sets a breakpoint at the specified address

delete <address>    -  This removes the breakpoint from the specified address

display             -  This command shows the breakpoints.
```

dr [<register-list>] - This command shows the content of the registers. If no registers are specified then it shows content of all the registers.

dmw [address] [len] - This command displays the value of the ‘‘len’’ words in memory starting at the specified address. The default value of len is one and the default value of address is the previously specified address.

dmh [address] [len] - This command displays the value of the ‘‘len’’ half words in memory starting at the specified address. The default value of len is one and the default value of address is the previously specified address.

dmb [address] [len] - This command displays the value of the ‘‘len’’ bytes in memory starting at the specified address. The default value of len is one and the default value of address is the previously specified address.

Chapter 6

TIPSIM

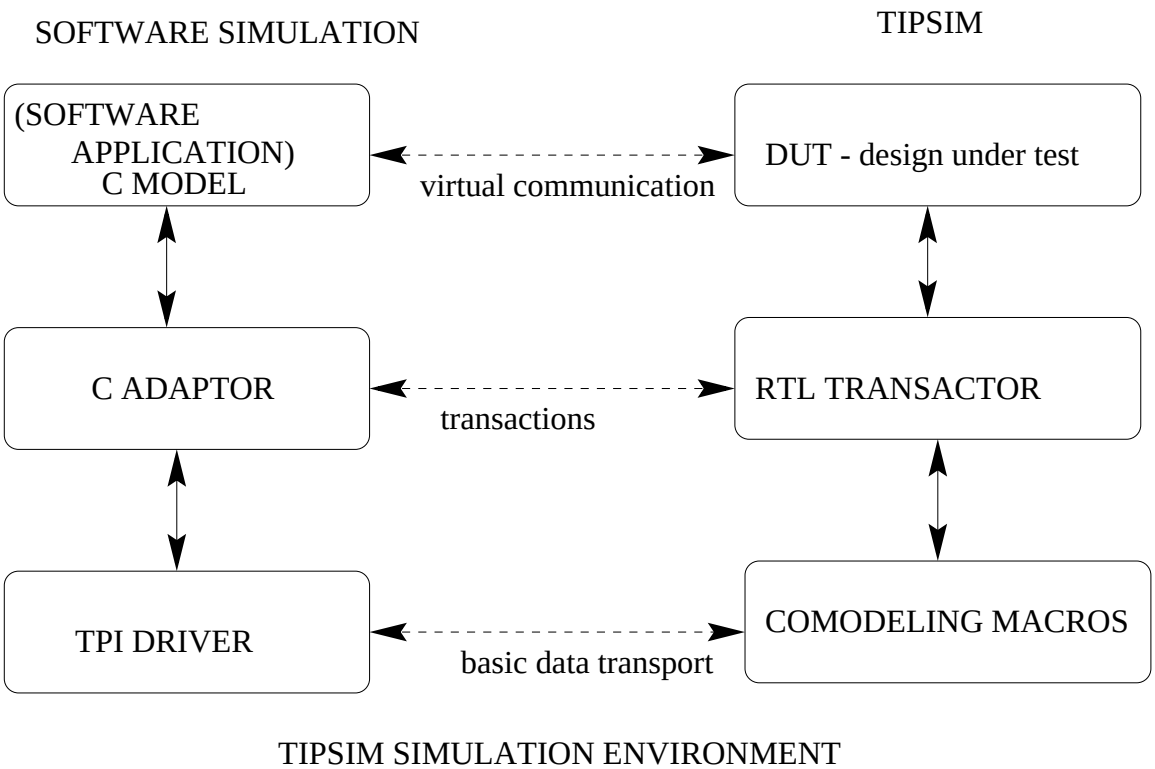

Figure 5: Protocol stack of TIPSIM cosimulation environment

TIPSIM provides low level abstraction using which communication can be established between a software application and the Verilog hardware simulator. The

TIP API of TIPSIM provides a library for sending data to the hardware simulation environment and the *COMODELING MACROS* provide the necessary Verilog code for sending data to the sofware simulation part. Thus these two provide a basic data transport link between the hardware and software simulation environment. The *C ADAPTOR* acts as a link between the C Model(application software) and the TIP API inorder to send the data to the hardware part. The *RTL TRANSACTOR* does the same thing for the hardware design.

So for making a HW/SW cosimulation environment using TIPSIM we have replaced the C Model with our Functional simulator. For this, we have implemented C Adaptor to make it compatible with the Functional simulator.

Chapter 7

Conclusions

7.1 Conclusion

As a part of this work we have developed an enhanced version of Retargetable Functional Simulator for our environment which uses Sim_nML as the language for describing processor models (Instruction Set). We have described almost the entire ARM Instruction Set in Sim_nML, which shows that Sim_nML is powerful enough to describe the semantics of an Instruction Set. Also, a debugger has been added which increases the utility of the *Functional Simulator*. We also have integrated the *Functional Simulator* with TIPSIM (Hardware Simulation Environment) to develop a prototype of a HW/SW cosimulation environment.

7.2 Future Work

Following points can be considered as an extension to this work.

- The algorithim used for generating functions for instructions is quite inefficient since it generates a lot of redundant code. So one can try to find a more efficient algorithim which reduces the amount of code being generated.
- The debugger can be futher enhanced to do symbolic debugging.

- One can work on the prototype of the Hw/Sw cosimulation environment developed to further enhance its utility and performance.

Bibliography

[1] Y. Subhash Chandra. Retargetable functional simulator. Master's thesis, IIT Kanpur, 1999.

[2] Dave Jaggar. *ARM Architecture Reference Manual.* Prentice Hall.