

# SPECIFICATION OF INTEL IA-32 USING AN ARCHITECTURE DESCRIPTION LANGUAGE \*

Jeff Bastian  
Texas Instruments  
12500 TI Blvd, MS 8714  
Dallas, TX 75243

Soner Önder  
Department of Computer Science  
Michigan Technological University  
Houghton, MI 49931-1295  
soner@mtu.edu

**Abstract** Designing, testing, and producing a new computer processor is a complex and very expensive process. To reduce costly mistakes in hardware, the microarchitecture is usually designed and tested with the aid of a software simulator. The FAST System enables microarchitects to develop architecture simulators rapidly and is less error-prone than using a high level language such as C. In this paper, we describe how the FAST System's Architecture Description Language (ADL) has been extended to facilitate the description of complex instruction sets such as Intel's IA-32 instruction set architecture. In this respect, we demonstrate that the notion of inheritance, a key concept in object oriented programming languages can be extended for *selective inheritance* to enable the specification of complex instruction set architectures in architecture description languages.

**Keywords:** Architecture Description Language, IA32, automatic simulator generation, cycle-accurate simulators.

## 1. Introduction

Micro-architecture exploration is a difficult, error-prone and development intensive endeavor. Traditionally, there has been three distinct approaches to micro-architecture exploration; namely, hand-coding a custom simulator, generation through a hardware description language and automatic generation through an architecture description language.

Custom simulators for a specific architecture are hand-coded in a general-purpose high-level-language, e.g. C. This group includes SimpleScalar, SuperDLX, SPIM, and URM [2, 5, 3, 9] among others. The second group in-

---

\*This work is supported in part by a grant from DARPA, PACC Award no. F29601-00-1-0183 to the Michigan Technological University and a CAREER award (CCR-0347592) from the National Science Foundation to Soner Önder.

cludes hardware description languages and simulators such as VHDL, VERILOG and ELLA [1, 12, 8, 4]. Most of the hand-coded simulators are very specific to the architecture they simulate which makes it difficult to make modifications to the ISA or the microarchitecture to see how the changes affect performance. Ranging from several thousand to 30,000 lines of C code and taking 12-24 man months to develop, these are complicated software systems. Such simulators embody problems of all large scale software projects, despite the best efforts spent to increase maintainability. Trying to study such an existing simulator's source code and make changes without breaking anything can be problematic at best. Similarly, hardware description languages are not suitable for micro-architecture exploration because they are designed to describe the hardware.

Architecture description languages on the other can specify the instruction set architecture (ISA), can generate support tools such as the assembler and the linker automatically and hide the details of instructions from the programmer. As a result, they enable a clean model of the micro-architecture operation. More importantly, they can specify and model the operation of the micro-architecture without tying it to a particular hardware implementation and therefore seamlessly map the instruction set specification to the micro-architecture specification.

Flexible Architecture Simulation Tool (FAST) and its description language *Architecture Description Language* (ADL) [6] is one such system, which has been in use by a number of universities to describe and simulate micro architectures of varying complexity. Thus, FAST fills in a gap between high-level architecture-specific simulators, and low-level hardware simulators. Doing so, it allows automatic generation of the necessary system tools (assemblers, linkers, and so on) through the ADL description.

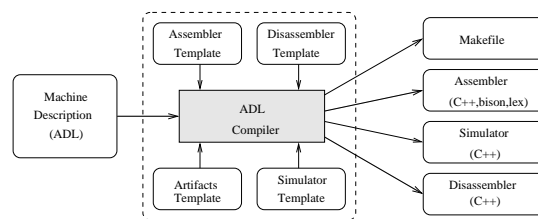


Figure 1. FAST System Components

The Flexible Architecture Simulation Tool (FAST) System shown in Figure 1 is a collection of four main components: (a) an ADL (Architecture Description Language) compiler; (b) support tools generated by the compiler (assembler, disassembler, linker, etc); (c) a cycle level simulator and debugger; (d) support tools for collecting and displaying statistics about the simulations.

When using the FAST system, the first step is to describe the architecture in question using ADL. An architecture described by ADL is made up of two distinct sections - one section describes the ISA, and the other section describes how the microarchitecture works (e.g., what are the pipeline stages and what happens during each stage). The instructions are described in a declarative form, while the microarchitecture is done in an imperative form similar to other high-level imperative languages, like C. An example instruction specification is shown in Figure 2.

```

add rd rs rt
  emit opcode=_special rs rt rd shamt=0 funct=_add
  attributes ( exu      : integer_unit,
               dest_type : integer_register,
               lop_type  : integer_register,
               rop_type  : integer_register,
               i_type    : alu_type,
               dest_reg  : rd )
begin
  exact s_EX
  dest=lop + rop;
end;
end

```

Figure 2. MIPS add instruction in ADL

## 2. IA-32 Architecture

The latest incarnation of IA-32 as seen in the Pentium 4 processor has its roots in the 8086 and 8088 processors from 1978. The ISA embodies a variable length instruction set encoding and the processor supports many memory models including segmented memory. The architecture also include overlapping registers. There are very few, if any, wasted bits in a typical x86 instruction. All these properties make the Intel IA-32 architecture quite challenging for an ADL specification.

In addition to compact encoding, there are many addressing modes used in IA-32, which are, for the most part, independent of the instruction since they are encoded using the ModR/M byte (and an SIB byte if necessary). The challenging aspect of the many addressing modes in IA-32 is trying to define them succinctly in ADL, since the fields are mostly independent of the opcode. That is, the opcode alone does not indicate all of the fields that follow the opcode. The simplest way to approach this problem is to enumerate every possible variation of an instruction as if it were a separate instruction, since ADL allows instructions to be overloaded, just like functions in C++. However, this leads to the problem of having to overload the same instruction many times due to the many addressing modes. There are nine addressing modes, however, three modes (Base + Displacement, Base + Index + Displacement,

and  $\text{Base} + (\text{Index} * \text{Scale}) + \text{Displacement}$ ) can use either an 8-bit or a 32-bit displacement, giving us 12 effective modes. Furthermore, there are restrictions on when `%esp` and `%ebp` can be used for base or index registers. Treating these restrictions as special addressing modes (which would be necessary in the current version of ADL) gives us 6 additional special case modes, for a total of 18 addressing modes! Creating separate ADL instruction definitions for every combination of x86 opcode with addressing mode would generate thousands of ADL instructions. This is tedious and highly error-prone.

Another challenge is the overlapping registers. IA-32 includes eight 32-bit general purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. However, in order to maintain backwards compatibility, there are aliases for 8-bit and 16-bit parts of the registers. For example, AL, AH, and AX all describe different parts of the EAX register. AL and AH are two 8-bit registers that represent the lower and upper 8 bits of the 16-bit register AX, and AX is the low 16 bits of the 32-bit register EAX.

Finally, unlike RISC architectures, instructions in CISC machines can operate directly on memory. This leads to instructions that can have a variety of arguments. The `mov` instruction for example takes two arguments, a source and a destination, one of which must be a register, the other can be either a register or a memory address. The address can come in any one of the 18 modes. In addition to creating a plethora of instructions, this makes the pipeline specification extremely complex.

Although some of these issues have been addressed within the context of *instruction set specification* with the SLED approach [10, 11], as it can be seen, the approach taken by SLED is inadequate for automatic generation of simulators. Although one can describe x86 ISA in less than 500 lines of code in SLED, the language was only designed for encoding and decoding instructions (as the name implies). Many instructions in x86 are encoded/decoded the same way with the only difference being the opcode, so *patterns* are used to define many instructions in one line. On the other hand, in order to tie in the micro-architecture specification, one needs to be able to specify the semantics of each instruction. Since the semantics of each instruction are very different, attaching semantics to many opcodes cannot be done with one line, and an alternative technique must be sought.

### 3. Our Solutions

Most of the problems we encountered (overlapping registers, variable length instructions, etc.) were handled by a simple extension of ADL or modification of an existing syntax. The real problem with x86 was the combination of instructions with multiple addressing modes. With over 400 instructions and many of them using up to 18 modes for memory addressing, plus register-to-

register arguments, we could easily end up with thousands of descriptions if we were to enumerate each permutation as a separate instruction.

```
# create groups of registers.
# Registers may exist in multiple sets.
typeset
  reg32 {%eax,%ecx,%edx,%ebx,%esp,%ebp,%esi,%edi},
  reg16 {%ax,%cx,%dx,%bx,%sp,%bp,%si,%di},
  reg8  {%al,%cl,%dl,%bl,%ah,%ch,%dh,%bh},
  reg32_noESP {%eax,%ecx,%edx,%ebx,%ebp,%esi,%edi},
  reg32_noESBP {%eax,%ecx,%edx,%ebx,%esi,%edi},
  reg32_noEBP {%eax,%ecx,%edx,%ebx,%esp,%esi,%edi},
  scaleVals { 1, 2, 4, 8 };

#- Instruction fields
type .....
  s8      signed integer  variable    8 bit,
  s16     signed integer  variable   16 bit,
  s32     signed integer  variable   32 bit;

addressing modes
  disp32      s32,
  base        "[" reg32_noESBP "]" ,
  base_index  "[" reg32_noEBP [-+] reg32_noESP "]" ,
  base_index_scale "[" reg32_noEBP [-+] reg32_noESP * scaleVals "]" ,
  index_disp32  "[" reg32_noESP * 1 [-+] s32 "]" ,index_scale_disp32
                "[" reg32_noESP * scaleVals [-+] s32 "]" ,
  ... ;
```

Figure 3. Use of typesets to describe addressing modes

Our careful study of the SLED encoding scheme where patterns are extensively used lead us to believe that it may be possible to define an encoding pattern for the addressing modes and let each instruction inherit the right pattern? This turned out to be the key idea for the x86 extensions to ADL: treat instructions as objects and use multiple inheritance with a twist! The encoding patterns would be defined by a series of templates, and the real instructions would inherit the properties from these patterns. The *objects* in ADL are the instruction templates and the instructions themselves. Templates differ from normal instructions in two ways.

Fields in instruction templates can be grouped or made optional with the use of regular-expression like syntax [7]. Parentheses group fields together, (*field1 field2 ...*), to indicate that all of the fields in the group must appear together. That is, *field1* cannot exist without *field2*, and vice-versa. This is useful for larger fields like the ModR/M byte in IA-32, which consists of three smaller fields: the 2 bit mod field, the 3 bit reg/op field, and the 3 bit r/m field, used to describe how memory and/or registers will be addressed. A ‘?’ following a field indicates the field is optional. For example, the SIB byte is optional depending on the ModR/M byte, thus, it appears as (scale index base)? . A ‘?’ is

really just shorthand for  $\{ n, m \}$  syntax (where  $n=0$  and  $m=1$ ) which says the previous item must appear at least  $n$  times but no more than  $m$  times. Finally, a  $|$  indicates logical-or, useful for fields that vary in size. Some instructions have 8-bit immediates, others 16-bit, and others 32-bit, and others none at all, so, putting it all together:  $(\text{imm8} | \text{imm16} | \text{imm32})?$

Templates can inherit properties from other instruction templates and override fields or sections from the parent. This allows creation of a *master template*. A master template is really just another template (i.e., it is not a special type of template), but it helps the programmer avoid syntactical errors. In IA-32, there is one general instruction format:

Prefixes	Opcode	ModR/M	SIB	Disp	Immediate
----------	--------	--------	-----	------	-----------

In this format there may be up to 4 prefixes, where each prefix is 1 byte long. The opcode field can be 1 or 2 bytes long and is followed by the optional ModR/M and SIB bytes. Displacement and the Immediate fields can be anywhere from 0 to 4 bytes. Templates and instructions inherit the properties illustrated in Figure 4.

The first item to notice is on the 3<sup>rd</sup> line, no arguments are given to the generic instruction name `intel`. The non-existent arguments will be overridden by the following templates. The `emit` line, on the other hand, defines every possible field that might be emitted by a descendant and uses the `?` and  $\{n,m\}$  modifiers to indicate optional fields. Only two attributes are defined at this time; more will probably be needed when the microarchitecture is implemented. Finally, there are three pipeline stages used to execute the instruction, an execute stage and a memory access before and after the execute stage for those instructions that need it (more on this below). Again, the pipeline stages may change with implementation of the microarchitecture.

An instruction that inherits from this `intel` master template is free to override the arguments, any of the emit fields, any of the attributes, or any pipeline stage. (Note that if a pipeline stage is overridden, the entire stage must be overridden, even if only one line is changed.) Inheritance is indicated by the `inherits` keyword following the instruction's arguments as shown in Figure 4.

The template `intel_r8_bis` has two arguments, an 8-bit destination register, `rd8`, and a memory location addressed by `base_index_scale` mode. It inherits from the `intel` master template and then defines exactly which fields will be emitted for this type of instruction. The `scale`, `index`, and `base` functions are built-in to the ADL language and, with the help of the regular expressions for the addressing modes, return the respective values for `scale`, `index`, and `base`. Finally, the `s_MEM` pipeline stage is used to load a byte into a temporary pipeline register which will be used by the `s_EX` stage in instructions that inherit from this template. (The code to actually load from memory will be

```

instruction template
begin
  intel          # no arguments given
  emit prefix1? prefix2? prefix3? prefix4?
  opcode{1,2}
  (mod reg_op rm)?
  (scale index base)?
  (disp8 {textbar} disp16 {textbar} disp32)?
  (imm8 {textbar} imm16 {textbar} imm32)?
  attributes
    (i_class : intel_class,
     op_type : intel_ops )
  begin
    exact s_MEM_LD
      # read from memory, if necessary
    end;
    exact s_EX
      # execute stage
    end;
    exact s_MEM_ST
      # write back to memory, if necessary
    end;
  end, ...

intel_r8_bis rd8 base_index_scale inherits intel
  emit opcode=0xF1 mod=00 reg_op=rd8 rm=100
  scale=<base_index_scale.scale>
  index=<base_index_scale.index>
  base=<base_index_scale.base>
begin
  exact s_MEM
    # calculate address = base + (index * scale)
    # load from memory: temp = dcache[address]
  end;
end,

```

Figure 4. Instruction templates and Using inheritance

specific to how the microarchitecture is implemented, so for now we describe what has to be done in comments.)

**Memory Addressing and Conditional Inheritance:** Once all the templates are defined as shown above, the final step is to create *conditional inheritance*. This borrows from the idea of multiple inheritance, except instead of inheriting all of the features from the parents, it only inherits from the one parent with the best fit. The best fit is determined by the arguments to the instruction. (Note that this implies the inherited arguments must be unambiguous.)

This allows us to create one template for each addressing mode. Each template will have the emit fields defined and other common properties. The child that inherits from the template then overrides the emitted opcode field and defines in the pipeline stage what exactly the instruction does (i.e., the semantics that make languages like SLED unfeasible for our work). The common ad-

addressing modes are then combined using the conditional inheritance feature into one template which the real instructions will inherit from. To reinforce the idea that this is not traditional multiple inheritance, the `|` operator (logical or) is used to split parents. For example, instructions that have a 32-bit register for a source and a 32-bit word in memory for a destination would inherit from the `intel_r32_rm32` template that is shown in Figure 5.

```
intel_r32_rm32 inherits
(
intel_r32_d32 | intel_r32_b | intel_r32_bi | intel_r32_bis |
intel_r32_id32 | intel_r32_isd32 | intel_r32_bd8 | intel_r32_bid8 |
intel_r32_bisd8 | intel_r32_bd32 | intel_r32_bid32 | intel_r32_bisd32 |
intel_r32_b_esp | intel_r32_b_ebp | intel_r32_b_bd8_esp | intel_r32_bd32_esp |
intel_r32_bi_ebp | intel_r32_bis_ebp
)
mov inherits intel_r32_rm32 emit opcode=0x8B
begin
    exact s_EX
        rd32 = temp;
    end;
end,
```

Figure 5. Conditional Inheritance and example instruction using inheritance

Each of the 18 templates `intel_r32_rm32` inherits from define an addressing mode (there are 12 modes plus 6 special modes for using `%esp` or `%ebp` as a base register). A real instruction then inherits from `intel_r32_rm32` as shown in Figure 5. To see this in action, consider the following x86 instructions:

```
mov %eax, DWORD PTR [%esp - 4]
mov %eax, DWORD PTR [%esp + %ebp*4 - 4]
mov %ax, WORD PTR [%esp - 4]
```

The instruction is the same in both cases, `movl`, but the arguments differ. However, they differ in a unique and unambiguous way which allow the compiler to match it against only one parent. The first instruction matches `intel_r32_rm32` and its parent `intel_r32_bd8` (base + 8-bit-displacement) (technically it also matches `intel_r32_bd32`, but the compiler will be smart enough to choose an 8-bit-displacement if it can via a *pragma*). Likewise, the second instruction matches `intel_r32_rm32` but with a different parent, `intel_r32_bisd8`. The third instruction matches none of the parents in `intel_r32_rm32`, so the compiler looks for another instruction to match against (which, in this case, will be `mov inherits intel_r16_rm16` and its parent `intel_r16_bd8`)<sup>4</sup>.

**Overlapping Registers:** C style unions and typesets were introduced to deal with overlapping registers. First, the physical registers are defined as generic

<sup>4</sup>Due to space limitations, not all templates are shown. Template names follow the same convention that is used with `intel_r8_bis`, namely, operand size, addressing mode.

32-bit registers, then a union is used to define the symbolic registers and which parts of the physical register are used in [start\_bit, length] notation and finally, a typeset is used to group the registers together into logical sets.

```

register file
  gpr [8, 32] # 8 registers, 32 bits each
  %reg0 0, %reg1 1, %reg2 2, %reg3 3,
  %reg4 4, %reg5 5, %reg6 6, %reg7 7;

union
  %eax %reg0[31, 32], %ecx %reg1[31, 32], ... # 32 bit registers
  %ax %reg0[15, 16], %cx %reg1[15, 16], ... # 16 bit registers
  %al %reg0[7, 8], %cl %reg1[7, 8], ... # 8 bit low registers
  %ah %reg0[15, 8], %ch %reg1[15, 8],
  %dh %reg2[15, 8], %bh %reg3[15, 8]; # 8 bit high registers

typeset
  reg32 { %eax, %ecx, %edx, %ebx, %esp, %ebp, %esi, %edi },
  reg16 { %ax, %cx, %dx, %bx, %sp, %bp, %si, %di },
  reg8 { %al, %cl, %dl, %bl, %ah, %ch, %dh, %bh };

```

Typesets are also used to place registers into special groups. For example, the %esp and %ebp registers cannot be used as a base register in base addressing mode, so a special group is created minus those registers.

```
typeset reg_noESBP { %eax, %ecx, %edx, %ebx, %esi, %edi };
```

Typesets can be used to group more than just registers. The opcode prefixes are defined as bitconstants and are split into four groups and only one prefix from each group can be used in an instruction. The typeset is used to define the groups.

```

typeset
  prefix_lock_repeat { _lock, _repne, _repnz, _rep,
    _repe, _repz },
  prefix_segment_branch { _cs_seg, _ss_seg, _ds_seg,
    _es_seg, _fs_seg, _gs_seg,
    _b_not_taken, _b_taken },
  prefix_operand { _operand_size },
  prefix_address { _address_size };

```

## 4. Conclusions and Future Work

The x86 is a powerful and compact ISA, but it's this same compactness that makes it so difficult to work with (in compilers, in simulators, and more). We have shown that by introducing the notion of conditional multiple inheritance we have tackled the most difficult challenges of x86 within the realm of an architecture description language. Our future work on IA-32 on FAST can be broken into two broad areas, namely the implementation of the language constructs in the ADL compiler and the completion of the micro-architecture specification.

## References

- [1] J.R. Armstrong and F.G. Gray. *Structured Logic Design with VHDL*. New Jersey: Prentice Hall, 1993.
- [2] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, V. 2.0. Technical Report 97-1342, Computer Sci. Dept., Univ. of Wisconsin Madison, 1997.
- [3] J.R. Larus. SPIM S20: A MIPS R2000 Simulator. Technical Report 90-966, Computer Sci. Dept., Univ. of Wisconsin Madison, 1990.
- [4] J.D. Morison and A.S. Clarke. *ELLA2000 A language for Electronic System Design*. McGraw-Hill, 1993.
- [5] C. Moura. SuperDLX a generic superscalar simulator. Technical Report 64, School of Computer Science, McGill University, 1993.
- [6] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.
- [7] Robert Pastel. Describing vliw architectures using a domain specific language. Master's thesis, Michigan Technological University, 2001.
- [8] D.L. Perry. *VHDL*. McGraw-Hill, 1991.
- [9] David Poplawski. The unlimited resource machine (urm). Technical report, Michigan Technological University, 1995.
- [10] Norman Ramsey and Mary F. Fernandez. The new jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference, New Orleans, LA*, pages 289–302, January 1995.
- [11] Norman Ramsey and Mary F. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.
- [12] D.E. Thomas and P.R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.