CHAPTER 1 CSL Instruction Set Architecture Generator

TABLE 1.1 Chapter Outline

1.1 Definitions
1.2 CSL Instruction Set Architecture (ISA) Overview
1.3 CSL ISA Concepts
1.4 Restrictions in positioning of a filed
1.5 CSL ISA Examples

1.1 Definitions

In the following table are the definitions for the terms used in this chapter.

TABLE 1.2 Definitions

Term used	Description
ISA	Instruction Set Architecture
assembler	A program that translates assembly-language instructions into machine-language instructions.
assembler code	A low-level language used in the writing of computer programs
instruction	A single operation of a processor within a computer architecture
instruction format	The organization of an instruction
datapath	A computation unit
opcode	The portion of an instruction that specifies the operation to be performed

1.2 CSL Instruction Set Architecture (ISA) Overview

microcode and micro engine/processor generator

We will design a microcode and micro engine/processor specfication langauge (the CSL microcode language). The CSL ISA language will describe a microcode instruction set. The specification of the instruction set will include the opcodes and other fields in the instructions. Furthermore, the specification of the instruction set will also specify the hardware elements which are controlled by the instruction set.

The cslc will convert the CSL ISA specfication into the following components:

- microcode documentation
- microcode compiler/assembler
- verilog or C++ microcode micro engine or processor
- verilog or C++ microcode controlled related datapath elements and/or signals to control datapath elements

CSL ISA specification will be compiled by the cslc into assembler code and constants (using CSL ISA specifiction).

Once the assembler has been generated then the user can write programs in the new assembly langauge and assemble program into a binary file.

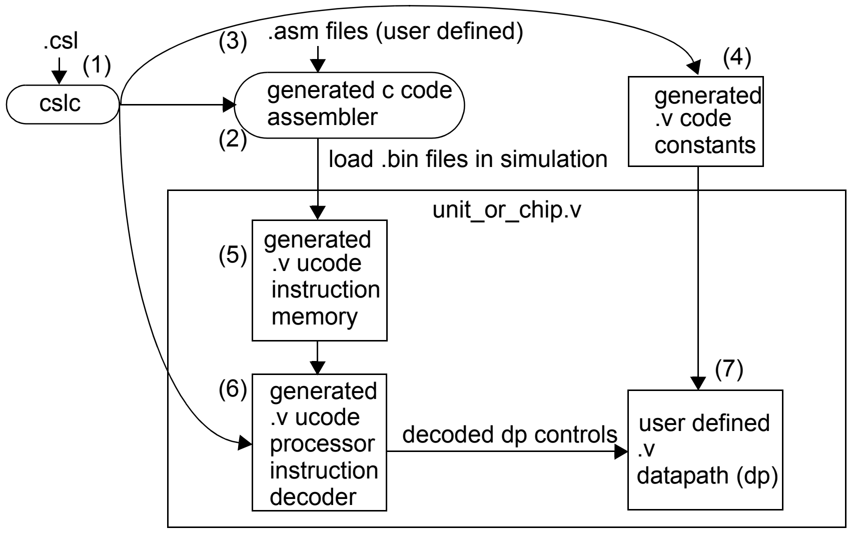
1.3 CSL ISA Concepts

1.3.1 ISA

An instruction set or instruction set achitecture (ISA) is (a list of) all instructions, and all their variations, that a processor can execute. ISA is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.

The usage model in 1.1 CSL ISA Usage Model shows how the assembler and verilog code is created, how the user writes asm code which is assembled and how the simulation loads the binary code in the instruction memory and executes the binary.

FIGURE 1.1 CSL ISA Usage Model


.csl
(1)
cslc
(3)
.asm files (user defined)
generated c code assembler
(2)
(4)
generated .v code constants
load .bin files in simulation
unit_or_chip.v
(5)
generated .v ucode instruction memory
(6)
generated .v ucode processor instruction decoder
decoded dp controls
(7)
user defined .v datapath (dp)

1.3.2 ISA Instruction

An instruction is a single operation of a processor within a computer architecture.The types of instruction allowed are defined and determined within the ISA. Instructions contain opcodes, each opcode is associated with an instruction format.

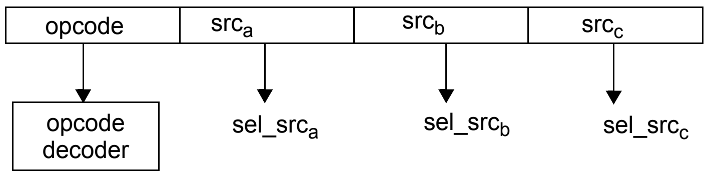
1.3.2.1 Instruction Format

Instruction formats specify the field names ? in the decoded that the instruction is interpreted by the ISA. This is done using format fields(subrange of the instruction).

FIGURE 1.2 Instruction Format

opcode	src_a	src_b	src_c

FIGURE 1.3 Istruction Decoder



1.3.2.2 Instruction fields (fields)

An format instruction no way/might/could contain multiple fields. This fields will select the datapath of this instruction.
The leftmost field is the opcode, an abbreviation of Operation Code. The opcode specifies the operation to be performed.

There may be multiple opcodes

Instructions contain opcodes; each opcode is associated with an instruction format

FIGURE 1.4 Opcode

opcode	fields

FIGURE 1.5 Instruction Formats

Type	Payload	
ALU opcode	subop	(fields)+
mem	subop	(fields)+
branch	subop	(fields)+

1.3.2.3 Assembler

An asssembler is a low-level language. An assembler creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications.

1.3.2.4 Select Lines

1.3.2.5 Decoding Instructions

The processor instruction decoder is a combinatorial circuit .Its purpose is to translate an instruction code into the select lines which drive the processor data path.

1.3.3 ISA decoder and signal select logic

An ISA contains instruction formats and instructions which contain fields. The ISA is associated with an instruction register (ir).

```
wire [ir.get_width() - 1 : 0)] ir;
```

The ir register represents a complex set of formats with possible overlapping fields and many different enums. CSLC will generate the decoders and all possible fields that can be connected to the ir when the following command is used.

```
ir.isa_name.generate_decoders();
ir.isa_name.generate_signals();
```

The fields can have associated enums. The enums can be decoded. The fields with associated enums are connected to decoders which generate the decoded signals. The name of the generated decoder output is the following:

```
(wire ir(.<field_name>)+_<enum_name>;)+ // one for each enum in field
```

The order of the enum signals is largest value to smallest value so the the left shift sets the correct signal.

```
assign { ir(.<field_name>)+_<enum_name>,
(ir(.<field_name>)+_<enum_name>)*} = 1'b1 = ir(_<field_name>)+;
```

The fields without associated enums are either constants or select values. The fields without associated enums are connected to wires. The name of the generated wire is:

```
wire [hid_field_name.get_width(0 - 1 : 0] hid_field_name_sig =
ir(_<field_name>)+;
```

The generated values when need to be pipelined to their downstream destination pipestage. Assume that a pipestage has been created and that the ir has bee added to a pipestage.

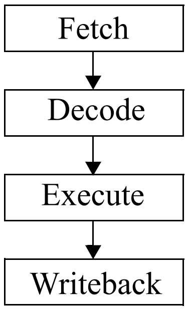
```
ir.isa_name(.field_name)+.send_generated_signals_to_destination_pipest
age(pipestage_name);
```

The above command will select the generated signal(s) associated with the ir.isa_name(.field_ name)+ and add the ir.isa_name(.field_name)+_<pipestage_suffix> to each pipestage between the current pipestage and the downstream destination pipestage creating a pipelined connection between the current pipestage and the consumer pipestage. The consumer pipestage is the pipestage that uses the signals.

1.3.4 RISC Pipeline

The reduced instruction set computer, or RISC, is a CPU design philosophy that favors a simpler set of instructions that all take approximately the same amount of time to execute. This is accomplish by using pipeline architecture. A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements. Pipelining reduces cycle time of a processor and hence increases instruction throughput, the number of instructions that can be executed in a unit of time. Pipelining doesn't decrease the time for a single datum to be processed; it only increases the throughput of the system when processing a stream of data. The single datum still travels through the entire pipeline.

FIGURE 1.6 Basic RISC Computer Pipeline F-D-E-W



1.4 Restrictions in positioning of a filed

TABLE 1.3 Restrictions in positioning of a field

		P	P	P
	so	sn	sp	sfp
child.set_offset	A	A	A	A
parent.set_next	I	X	A	I
parent.set_previous	I	A	X	I
parent.set_field_position	A	I	I	X

Where: so = set_offset(), sn = set_next(), sp = set_previous(), sfp = set_field_position()
p = parent, A = Allowed, X = not applicable, I = Illegal

See the csl_isa commands

1.5 CSL ISA Examples

EXAMPLE :

```
////////////////
// RISC 16  ISA//
////////////////
```

CSL CODE

```
//define the unit to be used by the decoder
   csl_unit decoder_unit;

 //define the opcodes enum (to be used by op field)
   csl_enum opcodes {
     ADD = 0,
     SLL = 3,
     SW  = 4,
     BNE = 6
   };

 //define instruction format 1 (to be used by branch and mem commands)
   csl_isa_instruction_format format1 {
 //define format1 fields
    csl_field op(13,15);
    csl_field rega(10,12);
    csl_field regb(7,9);
    csl_field imm(0,6);
//the default constructor
      format1() {
//set the width of the field
      set_width(16);
//set the position of the first(LSB) field
      set_field_position(imm,0);
//set the next field
      set_next_field(imm,regb);
      set_next_field(regb,rega);
      set_next_field(rega,op);
//set what enum to use with op field
      op.set_enum(opcodes);
      }
   };

//define instruction format 2 (to be used by shift and alu commands)
//it is derived from format1 and contain all its fields
```

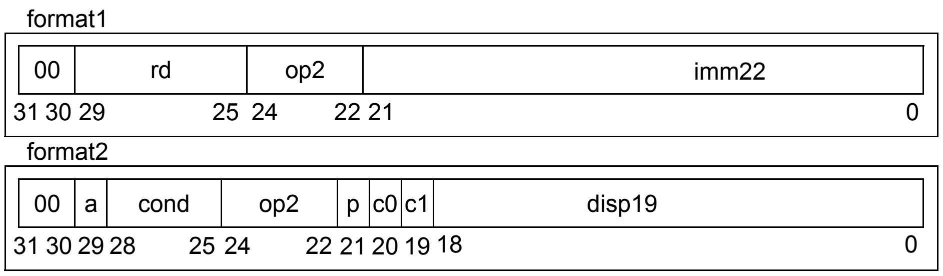
```
    csl_isa_instruction_format format2 : format1 {
//define format 2 fields
      csl_field unused(3,6);
//that will replace imm field
//define format 2 fields
      csl_field regc(0,2);
//the default constructor
      format2() {
//set the width of the field
        set_width(16);
//replace the imm field width "unused" and "regc" ones
        replace_field(imm,csl_list(unused,regc));
//set the position of the first(LSB) field
        set_field_position(regc,0);
//set next field
        set_next_field(regc,unused);
        set_next_field(unused,regb);
        set_next_field(regb,rega);
        set_next_field(rega,op);
      }
    };

 //define alu instruction
     csl_isa_instruction alu {
//default constructor
      alu() {
//sets the instruction format to use: format2
        set_instruction_format(format2);
      }
    };

 //define the shift instruction
    csl_isa_instruction shift {
//default constructor
      shift() {
//sets the instruction format to use: format2
        set_instruction_format(format2);
      }
    };

 //define mem instruction
     csl_isa_instruction mem {
//default constructor
```

```
      mem() {
//sets the instruction format to use: format1
        set_instruction_format(format1);
      }
     };

//define the branch instruction
    csl_isa_instruction branch {
//default constructor
      branch() {
//sets the instruction format to use: format1
        set_instruction_format(format1);
      }
    };

 //define risc 16 isa
     csl_isa risc16_isa {
//instantiate alu instruction
      alu add;
//instantiate shift instruction
      shift sll;
//instantiate mem instruction
      mem bne;
//instantiate branch instruction
      branch bne;
//default constructor
      risc16_isa() {
//set mnemonics for each instruction
        add.set_mnemonic("add");
//set mnemonics for each instruction
        sll.set_mnemonic("sll");
//set mnemonics for each instruction
   mem.set_mnemonic("mem");
//set mnemonics for each instruction
        bne.set_mnemonic("bne");
//set where to generate the decoder logic
        generate_decoder(decoder_unit);
//prints in a file the ISA description - (instructions and their formats)
        print("risc16_isa.txt");
      }
     };
```

EXAMPLE :

//
// Sun Open Sparc SETHI and Branches Instruction Formats EXAMPLE: //
//

FIGURE 1.7



CSL CODE

//------------------------
//define enums:
//------------------------

```
    csl_enum opcodes {
      BR   = 0,
      CALL = 1
    };

    csl_enum opcodes2 {
//dummy
      A0 = 0,
      A1 = 1
    };

    csl_enum cond {
//branch always
      BPA  = 8,
//branch never
      BPN  = 0,
//branch on not equal
      BPNE = 9,
//branch on equal
      BPE  = 1,
//branch on greater
```

```
    BPG  = 10
  };

//-----------------------
//define decoder units:
//-----------------------

  csl_unit decoder {
    //...
  }

//-----------------------
//define formats:
//-----------------------
  csl_isa_instruction_format format1 {
//define fields
    csl_field op(30,31,opcodes);
//define fields
    csl_field rd_f1(25,29);
//define fields
    csl_field op2(opcode2);
//define fields
    csl_field imm2(21,0);

      format1() {
//set the width of the format
        set_width(32);
      }
  };

//format2 is derived from format1
  csl_isa_instruction_format format2 : format1 {
    csl_field a(29, 29);
    csl_field cond(25, 28);
    csl_field c1(21,21);
    csl_field c0(20,20);
    csl_field p(19,19);
    csl_field disp19(0,19);

    format2() {
      set_width(32);
      replace_field(rd, csl_list(a,cond));
      replace_field(disp22, csl_list(c1,c0,p,disp19));
      }
    };

//-------------------------
//define affected register:
//-------------------------

    csl_register cond_reg;

//-----------------------
//define instructions:
//-----------------------

//branch always
    csl_isa_instruction branch_always {
      branch_always() {
        set_instruction_format(format2);
        //set_field(cond, BPA); - to be fixed
      }
    };


//branch never
    csl_isa_instruction branch_never {
      branch_never() {
        set_instruction_format(format2);
        //set_field(cond, BPN); - to be fixed
      }
    };


//branch on not equal
    csl_isa_instruction branch_on_not_equal {
      branch_on_not_equal() {
        set_instruction_format(format2);
        //set_field(cond, BPNE); - to be fixed
      }
    };


//branch on equal
    csl_isa_instruction branch_on_equal {
      branch_on_equal() {
```

```
      set_format(format2);
      //set_field(cond, BPE); - to be fixed
    }
  };
```

//branch on greater

```
  csl_isa_instruction branch_on_greater {
    branch_on_greater() {
      set_instruction_format(format2);
      //set_field(cond, BPG); - to be fixed
    }
  };
```

//-----------------------
//define ISA:
//-----------------------

```
  csl_isa open_sparc_isa {
```

//instantiate instruction

```
    branch_always ba;
```

//instantiate instruction

```
    branch_never bn;
```

//instantiate instruction

```
    branch_on_not_equal bne;
```

//instantiate instruction

```
    branch_on_equal be;
```

//instantiate instruction

```
    branch_on_greater bg;
    open_sparc_isa() {
```

//sets the decoder's name

```
      set_decoder_name("instr_decoder");

      ba .set_asm_mnemonic("ba");
      bn .set_asm_mnemonic("bn");
      bne.set_asm_mnemonic("bne");
      be .set_asm_mnemonic("be");
      bg .set_asm_mnemonic("bg");
```

//sets the decoder output signals prefix

```
      set_decoder_out_name_prefix("prefix");
```

//sets the decoder output signals sufix

```
      set_decoder_out_name_suffix("suffix");
```

//generates the decoder with corresponding signals (ready to be connected)

```
      generate_decoder(decoder_unit);
```

```
//prints in a file the ISA description (instructions and their formats)
        print("open_sparc_isa.txt");
```