CSL Example Unit

CHAPTER 1 CSL Example

TABLE 1.1 Chapter Overview

1.1 CSL Example

We show how to use the chip specfication language (CSL) to design an example chip. By reading this example we believe you will come to the conclusion that by using CSL to design this sample design we have saved 95% of the time that we would have spent using pure Verilog and C++ with no automation.

The first step when designing a chip is to draw the block diagram for the chip. Blocks are referred to as units in the CSL lanaguage. We will design a simple chip chip. The name of the top level is *chip*. The name of the units inside are *pc*, *im*, *rf*, and *alu*.

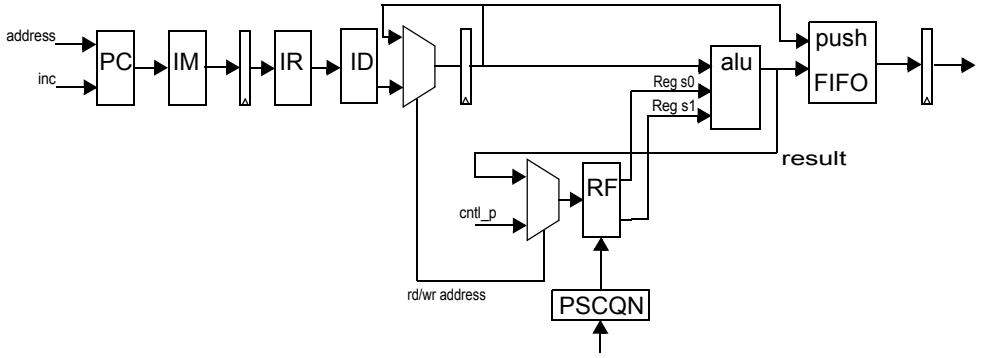


FIGURE 1.1

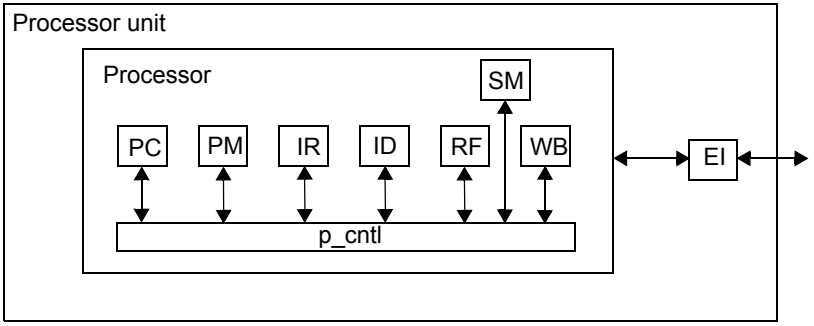


FIGURE 1.2 hierarchy of units

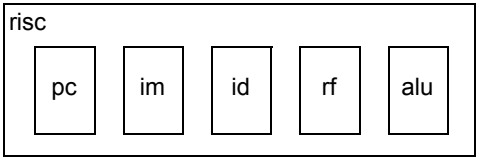


Table 1.2 shows the unit name and unit abreviation.The next step is to define the memory map for

TABLE 1.2 Risc units

Unit name	abreviation
program counter	pc
instruction memory	im

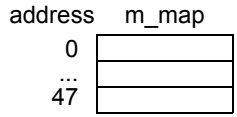
TABLE 1.2 Risc units

Unit name	abreviation
instruction decoder	id
register file	rf
arithmetic logic unit	alu

the chip (csl_memory_map).

We will create a memory map with the name m_map with an address range from 0 to 47. The width of the memory words is 32 bits. The memory words are word aligned so we set the address increment to 1.

FIGURE 1.3 A memory map with 0-47 address space



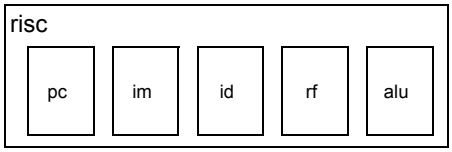
CSL CODE

```
//a memory_map with an address range from 0 to 47 with an increment 1
csl_memory_map chip_mem_map( 0, //lower range value
                            47, //upper range value
                             1  //address increment amount
                            );
chip_mem_map.set_width(32); // set the memory word width to 32-bits
```

The next step when designing a chip is to define the hierarchy for the chip and the interface for each unit using the csl_interconnect specification.

Create the CSL units and add the units to the chip using the CSL add_instance method..
Create the top level unit and the child unit.

FIGURE 1.4 Top level unit

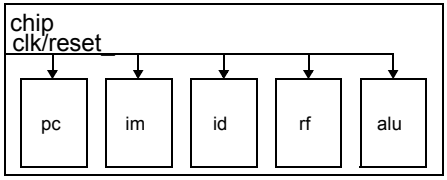


CSL CODE

```
csl_unit chip;
csl_unit pc; // program counters
csl_unit im; // instruction memory
csl_unit id; // instruction decoder
csl_unit rf; // register files
csl_unit alu; // ALU
//instantiate the childdren in top
scope chip {
// the following lines of code add instances to the chip
   pc pc0;
   im im0;
   id id0;
   rf rf0;
   alu alu0;
}
```

The next step when designing a chip is to connect the clock and reset signals to each unit (csl_interconnect).

FIGURE 1.5 Clock and reset connections



CSL CODE

```
csl_unit chip {
    csl_port clk(input);
    csl_signal clk;       // create a new signal clk in the scope chip
    chip() {
      clk.set_type(clock);  // set the type of the signal clk to clock
                            // add the port clk to the chip interface
    clk.connect(chip.**); // use the wildcard operator "**" to refer to
    }           // all modules under chip connect clock to all
                           // modules
    csl_signal reset_;       // create a new signal reset_ in the scope
                             //chip
    reset_.set_type(reset);  // set the type of the signal reset_ to
                             //clock
    add.input(reset_);       // add the port reset_ to the chip
                             //interface
   reset_.connect(chip.**); // use the wildcard operator "**" to refer
                             //to
                          // all modules under chip connect clock to all
                            // modules

}
```

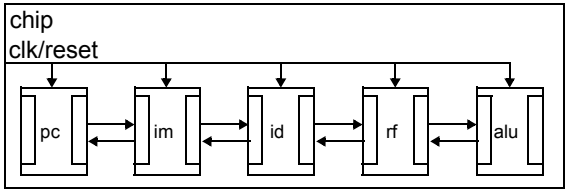
The next step when designing a chip is to add interfaces to each unit and connect the interfaces between the units (csl_interconnect). Each unit's interface is connected to the next unit's interface (Figure 1.6 interface connections). An interface is an object which holds ports and other interfaces.

Connectivity objects (e.g. signal, port, interface, ...) are connected to interfaces.

The unit interface creation is deferred. In the following pages the ports are added to the unit interfaces.

Note: 1.6 interface connections shows the five units in the chip. Each unit in the chip contains one or more pipestages which comprise the processor pipeline.

FIGURE 1.6 interface connections



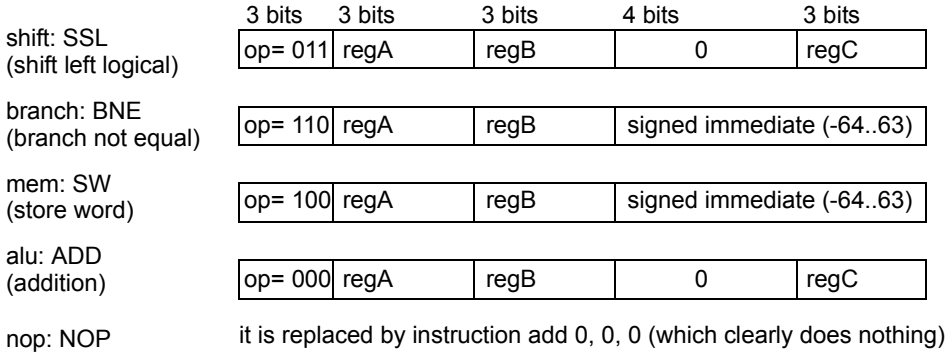
CSL CODE

```
// Connect each unit's interface to the next unit's interface
    scope chip {
       pc_0.connect(im_0);  // connect pc_0's interface to im_0's interface
        im_0.connect(id_0);  // connect im_0's interface to id_0's interface
        id_0.connect(rf_0);  // connect id_0's interface to rf_0's interface
       rf_0.connect(alu_0); // connect rf_0's interface to alu_0's inter-
                            // face
    }
```

The next step is to define the instruction set architecture (ISA) for the unit (csl_isa).

Create instuction formats.

FIGURE 1.7 RISC 16 ISA



CSL CODE

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

//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 {
  csl_field op(13,15);         //define format1 fields
  csl_field rega(10,12);       //define format1 fields
  csl_field regb(7,9);         //define format1 fields
  csl_field imm(0,6);          //define format1 fields

  format1() {                  //the default constructor
    set_width(16);             //set the width of the field
    set_field_position(imm,0); //set the position of the first(LSB)
field
```

```
    set_next_field(imm,regb);  //set the next field
    set_next_field(regb,rega); //set the next field
    set_next_field(rega,op);   //set the next field
    op.set_enum(opcodes);      //set what enum to use with op field
  }
};

//define instruction format 2 (to be used by shift and alu commands)
csl_isa_instruction_format format2 {
 csl_field unused(3,6);  // define format 2 fields that will replace
                         // imm field
  csl_field regc(0,2);    // define format 2 fields that will replace
                         // imm field

  format2() {             //the default constructor
    set_width(16);        //set the width of the field
    extend_format(format1);  //extend(copy all fields)from format1
    replace_field(imm,csl_list(unused,regc)); //replace the imm field
width unused and regc ones
    set_field_position(regc,0); //set the position of the first(LSB)
field
    set_next_field(regc,unused);              //set next field
    set_next_field(unused,regb);              //set next field
    set_next_field(regb,rega);                //set next field
    set_next_field(rega,op);                  //set next field
  }
};

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

//define the shift instruction
csl_isa_instruction shift {
  sll() {                           //default constructor
    set_instruction_format(format2); //set the instruction format to
```

```
                                    //use(format2)
  }
};

//define mem instruction
csl_isa_instruction mem {
  sw() {                            //default constructor
    set_instruction_format(format1); //set the instruction format to
                                    //use(format1)
  }
};

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

//define risc 16 isa
csl_isa risc16_isa {
  alu    add;                      //instantiate alu    instruction
  shift  sll;                      //instantiate shift  instruction
  mem    bne;                      //instantiate mem    instruction
  branch bne;                      //instantiate branch instruction

  risc16_isa() {                   //default constructor
    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 mnemonics for each instruction
   generate_decoder(decoder_unit); //set where to generate the decoder
                                    //logic
    print();                   //prints in a file the ISA description
                               //(instructions and their formats)
  }
};
```

The risc chip pipeline is a set of connected pipestages in each of the units in the example design. The next step, which is optional, is to declare a pipeline using the csl_pipeline command, and to declare the pipestages using the csl_pipestage command. The pipestages are then added to the pipeline. The csl_pipeline command creates a new pipeline object.

State elements in each unit are assigned to a pipestage object which are part of the processor pipeline.

Each subsequent pipestage is connected to the previous pipestage using the set_previous_pipestage method. Previous pipestages can be connected to subsequent pipestages using the set_next_pipestage method. Pipestages are either automtaically assigned a pipestage number based on the previous pipestage number incremented by one or are explicitly assigned a new pipestage number. The first pipestage's number is initialized with 0.

Pipestages can be named. The pipestage number and/or the pipestage name may be used in the generated Verilog variable names. The use of the pipestage name and number is controlled by the use_pipestage_name and use_pipe_stage_number methods.

CSL CODE

```
csl_pipeline chip_pipeline(5);
```

The next step is to declare any components which are automatically generated

First add the **pc**-program counter register and surrounding logic using a csl_register command (Figure 1.8 automatically generated components). Declare a program counter register with the name pcr (Figure 1.9 program counter register and surrounding logic (SHRINK FIGURE)).

FIGURE 1.8 automatically generated components

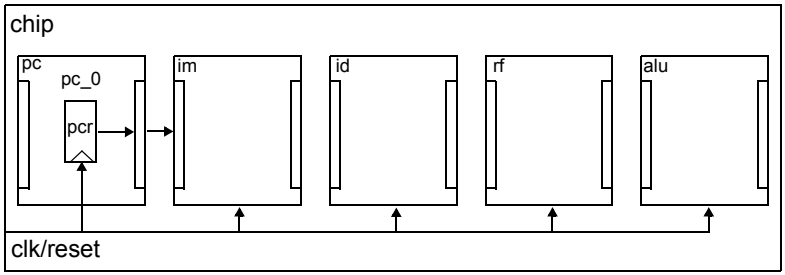
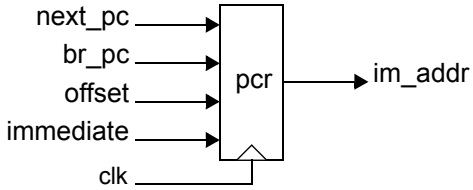


FIGURE 1.9 program counter register and surrounding logic (SHRINK FIGURE)



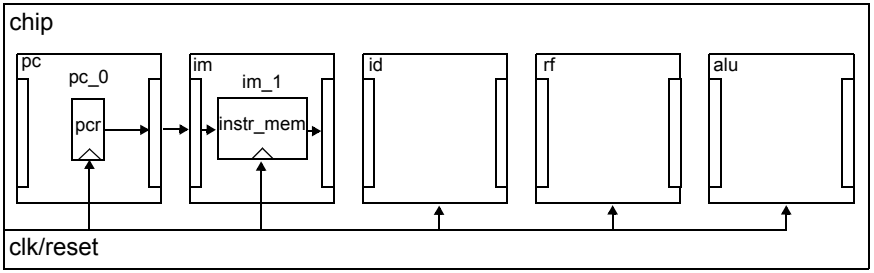
In this example we create an instance of a program counter register called *pcr*

CSL CODE

```
csl_register pcr;
pcr.set_width(instruction_width);
pcr.set_type(pc); // set register type to program counter
pcr.connect_clock( clk );
csl_pipestage pcr_pipe;
pcr_pipe.set_pipestage_number(0);  // initialize the pipeline number
//pcr_pipe.set_next_pipestage(im);
pcr_pipe.set_pipestage_name("pc"); // set the pipestage name
chip_pipeline.add_pipestage(pc);
```

Add the **im**-instruction memory (csl_sram).

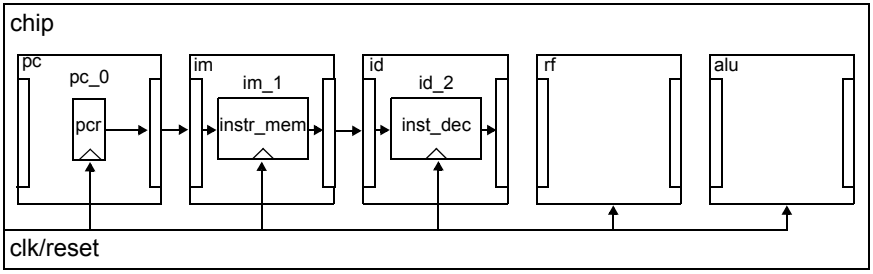
FIGURE 1.10 automatically generated components




```
// generate an instruction memory unit
csl_int instr_mem_num_wds(1 < 10); // 1K memory words
csl_sram instr_mem(instr_mem_num_wds, instruction_width);

// connect the read and write ports of the instruction memory to the im
interface
instr_mem.connect_wr_data(im_wr_data);
instr_mem.connect_wr_addr(im_wr_addr);
instr_mem.connect_wr_en   (im_wr_en );

instr_mem.connect_rd_addr(pc_addr);
instr_mem.connect_rd_en(pc_v);
instr_mem.connect_rd_addr(pc_addr);

// This is done automatically by the cslc pipeline engine
// when a pipeline attribute is added to a state element
// instr_mem.set_pipestage_valid_input(pc_v);
// instr_mem.set_pipestage_valid_output(im_v);

csl_pipestage im;
// The pipestage number is automatically set by cslc
// the pipeline engine
im.set_pipestage_name("im"); // set the pipestage name
im.set_previous_pipestage(pc);
im.set_next_pipestage(id);
chip_pipeline.add_pipestage(im);
```

Add the id-instruction decoder (csl_isa) method. Create the instruction set. Then create the instruction decoder in a separate unit and the pass through logic that extracts each of the fields for each format. Create an instruction format type field. Include the unit in the id unit.

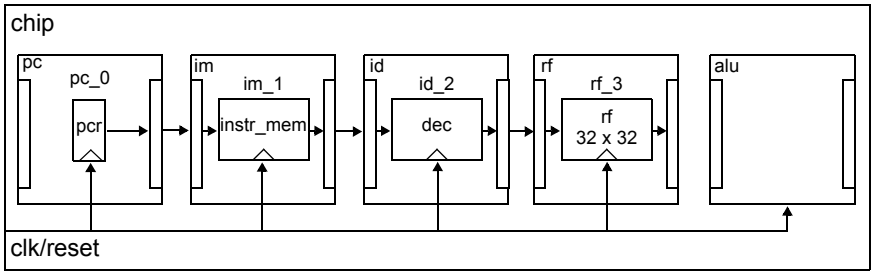
FIGURE 1.11 automatically generated components




```
// generate an instruction decoder module named "inst_dec" for the
chip_isa ISA
chip_isa.gen_decoder_unit(inst_dec);

// add output pipestage delay of 1
inst_dec.set_output_delay(1);

// create a signal group using the inputs of the inst_dec
csl_signal_group inst_dec_in(inst_dec.get_inputs());
// create a signal group using the outputs of the inst_dec
csl_signal_group inst_dec_out(inst_dec.get_outputs());

// connect the signal groups to the id unit interface
id.add_interface(inst_dec_in);
id.add_interface(inst_dec_out);

csl_pipestage id;
// The pipestage number is automatically set by cslc
// the pipeline engine
id.set_pipestage_name("id"); // set the pipestage name
id.set_previous_pipestage(im);
id.set_next_pipestage(rf);
chip_pipeline.add_pipestage(id);
```

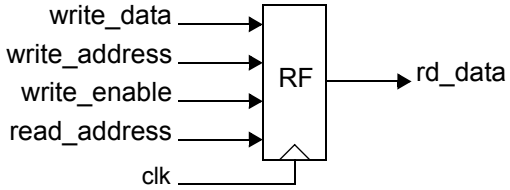
Add the **rf**-register file (csl_register_file).

FIGURE 1.12 automatically generated components



Declare a new register file with the name register_file_name.

FIGURE 1.13 Register file with no special options



In this example we simply create an instance of a register file called *reg_file*

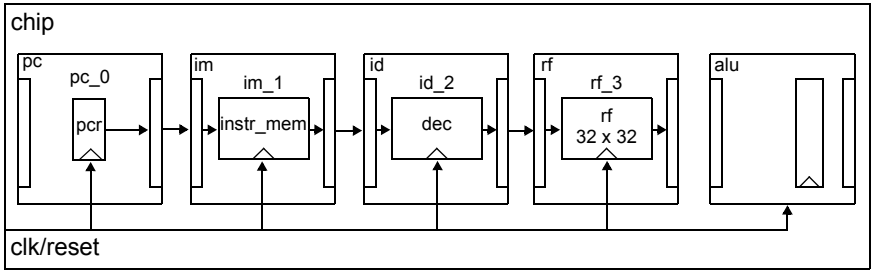
CSL CODE

```
csl_register_file reg_file;
reg_file.set_width(32);
reg_file.set_depth(4);
reg_file.connect_clock( clk );

// delay the outputs of the register file
// add a pipeline to the output of the unit with a delay of 1
reg_file.add_pipeline_delay(output, 1);

reg_file.gen_unit(regfile); // generate a register file unit named
                            // "reg_file"
// create a signal group using the inputs of the inst_dec
csl_signal_group rf_in(regfile.get_inputs());

// create a signal group using the outputs of the inst_dec
csl_signal_group rf_out(regfile.get_outputs());
```

```
regfile.add_interface(rf_in);
regfile.add_interface(rf_d0.get_output());

rf.set_address(31,0); // add the register file to the chip's address
space

csl_pipestage rf;
// The pipestage number is automatically set by cslc
// the pipeline engine
rf.set_pipestage_name("rf"); // set the pipestage name
rf.set_previous_pipestage(id);
rf.set_next_pipestage(alu);
chip_pipeline.add_pipestage(rf);
```

Add the pipestage to the ALU using the csl_pipestage specfication.

FIGURE 1.14 add pipestage

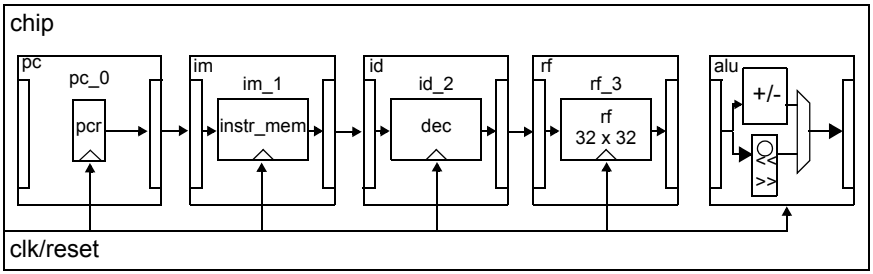


CSL CODE

```
csl_pipestage alu_pipe;
//The pipestage number is automatically set by the cslc pipeline engine
alu_pipe.set_pipestage_name("alu"); // set the pipestage name
alu_pipe.set_previous_pipestage(rf);
chip_pipeline.add_pipestage(alu_pipe);
csl_register reg(alu.get_width());
reg.add_to_pipestage(alu);
```

Add the **alu**-shift register (csl_register).

FIGURE 1.15 Automatically generated components



Create a shift register that has serial input and parallel output.

FIGURE 1.16 shift register

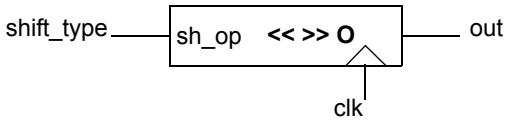
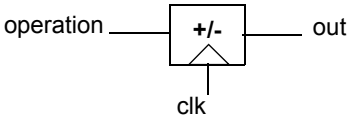


FIGURE 1.17 Arithmetical unit



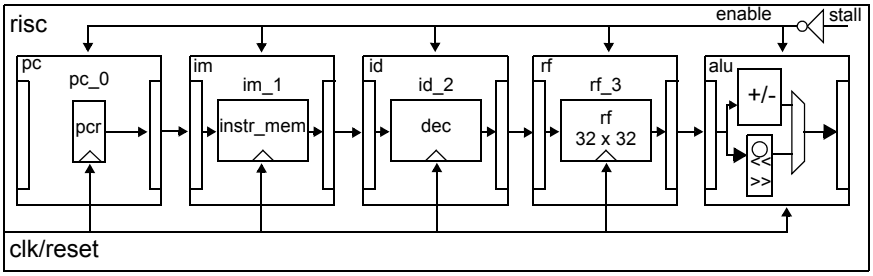
In this example we simply create an instance of a shift register called shr and enable all shift operations.

CSL CODE

```
csl_register shr;
shr.set_width(32);
shr.connect_clock( clk );
shr.set_type(sft); //register type is shifter
shr.set_clock(clk);
shr.set_output(sr_out, 1);
shr.set_shift_type(shl, 0); //shift logical left and assign opcode
shr.set_shift_type(shr, auto); //shift logical right, auto increment
opcode
shr.set_shift_type(sar, auto); //shift arithmetic left
shr.set_shift_type(sal, auto); //shift arithmetic right
shr.set_shift_type(ral, auto); // rotate arithmetic left
shr.set_shift_type(rar, auto); // rotate arithmetic right
```

The next step is to connect the inverse of the stall signal to the pipeline (csl_interconnect and csl_pipeline).

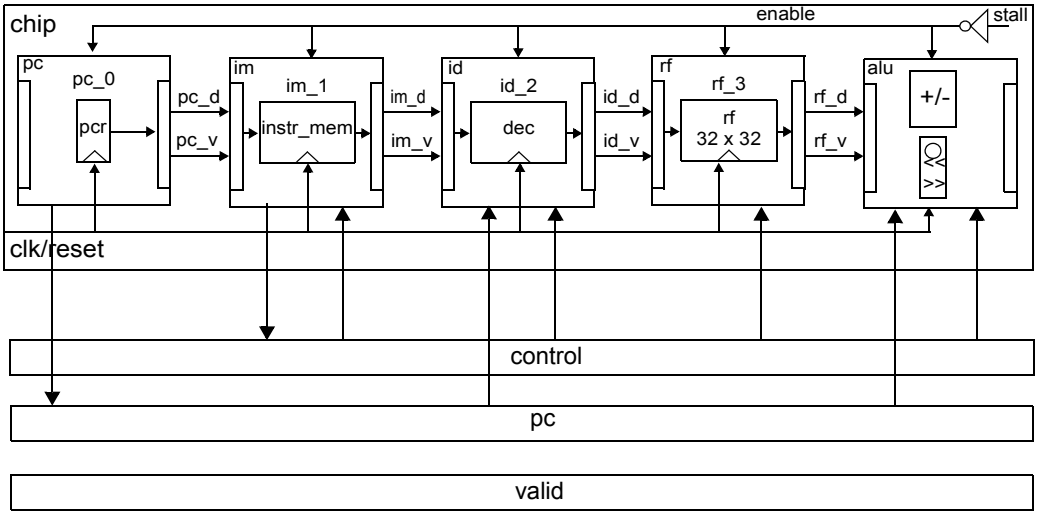
FIGURE 1.18 add leaf level logic



Add CSL example

The next step is to write the logic for the unit and include it in a leaf level unit that was generated from the csl_interconnect specification.

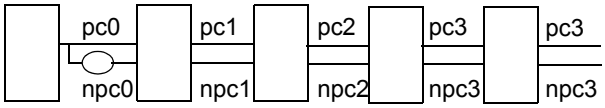
FIGURE 1.19 add leaf level logic



Add CSL example

The next step is to add the pc and next pc logic to the pipeline.

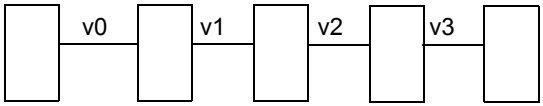
FIGURE 1.20 pc and nextpc pipelines



Add CSL example

The next step is to add the valid logic to the pipeline.

FIGURE 1.21 valid pipeline



EXAMPLE :

CSL CODE:

```
// Add a valid bit to the pipeline
risc_pipline.add_valid(input_valid_bit);
//valid needs to be qualified with stall
```

The next step is to write the csl_testbench and csl_verification_component specifications which will be used to verify the chip

We will now define the testbench clock that drives the DUT, the ALU in this case.

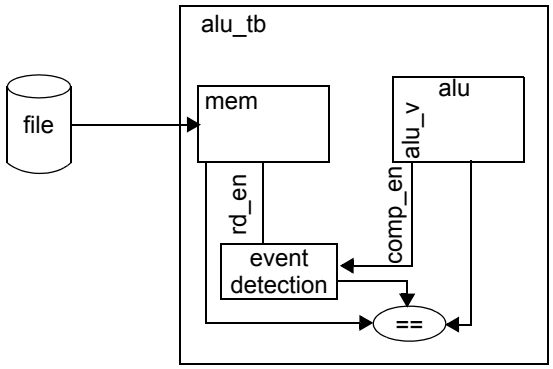
CSL CODE

```
csl_testbench tb;
//add an instence of the ALU to the testbench
tb.add_dut_instance(ALU, DUT);
//create a clock generator and clock signal
csl_clock clk_gen;
clk_gen.set_timebase(ms);
clk_gen.set_period(10);
clk_gen.add_clk_out(clk);
tb.add_signal(clk);
//add the clock signal to the testbench
tb.add_clock(clk);
```

NOTE: fix using signal generator!

We will now define the testbench event that triggers the expect vector and DUT output comparison

FIGURE 1.22 ALU testing using vectors



CSL CODE

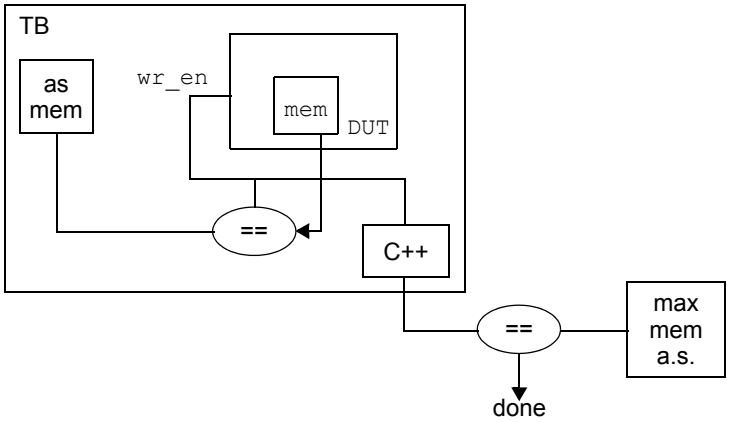
```
csl_event comp_en(alu.alu_v);
csl_vector alu_exp_vec;
alu_exp_vec.add_event(comp_en);
```

We will now define the testbench stimulus and expect vectors for the tested ALU unit (csl_verfication_components).

```
csl_vector alu_stim_vec, alu_exp_vec;
alu_stim_vec.set_stimulus_filename("alu_stim.vec");
alu_exp_vec.set_expected_filename("alu_exp.vec");
tb.add_vector_instance(alu_stim_vec);
tb.add_vector_instance(alu_exp_vec);
```

We will now define the testbench architectural state for the register file unit testing. The archictural state of the RF will be compared with the expected architectural state read from memory.

FIGURE 1.23 Register file architectural state testbench



CSL CODE

```
csl_testbench tb;
//add an instence of the RegisterFile to the testbench
tb.add_dut_instance(RF, DUT);
//create the architectural state
csl_arch_state rf_exp_as;
rf_exp_as.set_expected_filename("rf_exp.as");
//set the maximum number of arch states
as_a.set_max_num_states(50);
tb.add_arch_state_instance(rf_exp_as, rf_exp_as0);
```

The next step is to write the C++ simulator code that models the design under test. Use the memory map constants to map the addresses in the C++ simulator to the same addresses in the RTL DUT.

C++ CODE

```
//C++ code goes here
```

The next step is to integrate the C++ vector and architectural state readers and writers into the C++ simulator.

C++ CODE

```
//C++ code goes here
```

MOVE THE FOLLOWING TO THE APPROPRIATE DOCS

Engine data frame input

FIGURE 1.24

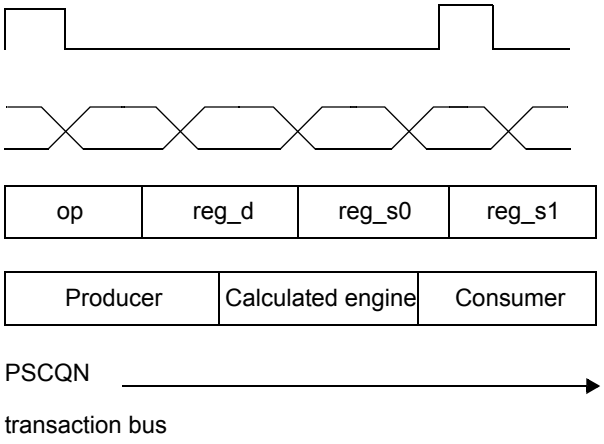
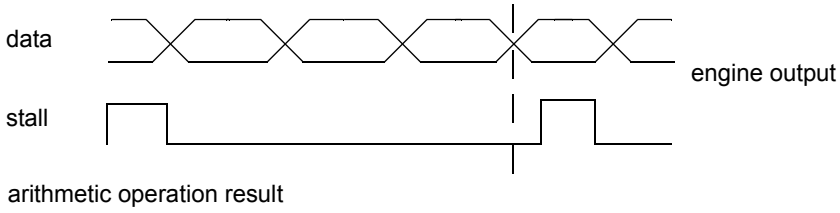


FIGURE 1.25



Instruction set
PSQN counts the number of DF's and the instruction type frequency

TABLE 1.3

operation	
rd	read data out of register file procedures are valid ? op
wr	write instruction to P.M. or data to register file
+	addition
-	substraction
cnt	number of packets in frame one valid transaction per frame

The counters can be read or cleared
P.M. = Program Memory

CHAPTER 1 CSL Microengine

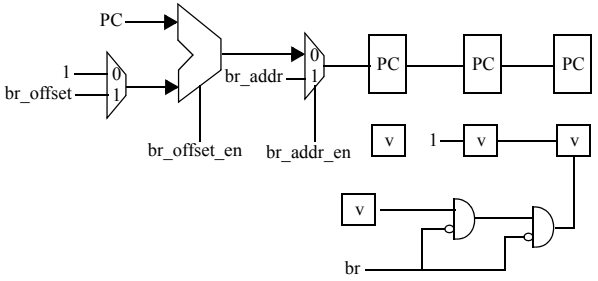
TABLE 1-1CHAPTER OUTLINE.

1.2 Definition

1.3 CSL Microengine Overview

1.4 CSL Concepts

FIGURE 1.26



Verilog Code:

```
//THIS CODE WAS GENERATED USING THE FASTPATHLOGIC CSL COMPILER
/THIS CODE WAS GENERATED USING THE FASTPATHLOGIC CSL COMPILER
//COPYRIGHT (c) 2005, 2006 FastpathLogic Inc

module mengine(clock,
            reset_,
            ir,
            rf_ctrl,
            ctrl);
 input [0:0] clock;
 input [0:0] reset_;{//THIS CODE WAS GENERATED USING THE FASTPATHLOGIC CSL COM-
PILER
                /THIS CODE WAS GENERATED USING THE FASTPATHLOGIC CSL COMPILER
                //COPYRIGHT (c) 2005, 2006 FastpathLogic Inc

                module mengine(clock,
                            reset_,
                            ir,
                            rf_ctrl,
                            ctrl);
                 input [0:0] clock;
                 input [0:0] reset_;
                 input [31:0] ir;
                 output [25:0] rf_ctrl;
                 output [5:0] ctrl;
                 mmux mmux( );
                 mpc mpc( );
                 mrom mrom( );
                 mir mir( );
                 mdecode mdecode( );//COPYRIGHT (c) 2005, 2006 FastpathLogic Inc

                module mengine(clock,
                             reset_,
                             ir,
                             rf_ctrl,
                             ctrl);
                  input [0:0] clock;
                  input [0:0] reset_;
                  input [31:0] ir;
                  output [25:0] rf_ctrl;
                  output [5:0] ctrl;
                  mmux mmux( );
                  mpc mpc( );
                  mrom mrom( );
                  mir mir( );
```

```
  mdecode mdecode( );
  mctrl mctrl( );
endmodule

module mmux(ir_mux_jmpto,
        ml_start,
        ir_mux_addr,
        mux_pc_next_addr);
  input [5:0] ir_mux_jmpto;
  input [5:0] ml_start;
  input [0:0] ir_mux_addr;
  output [0:0] mux_pc_next_addr;
  //behavior
  assign mux_pc_next_addr = ir_mux_addr ? ml_start : ir_mux_jmpto;
  //end behavior
endmodule

module mpc(clock,
        reset_,
        ldmpc,
        mux_pc_next_addr,
        pc_rom_addr);
  input [0:0] clock;
  input [0:0] reset_;
  input [0:0] ldmpc;
  input [0:0] mux_pc_next_addr;
  output [5:0] pc_rom_addr;
endmodule

module mrom(pc_rom_addr,
        rom_ir_instr);
}
input [31:0] ir;
output [25:0] rf_ctrl;
output [5:0] ctrl;
mmux mmux( );
mpc mpc( );
mrom mrom( );
mir mir( );
mdecode mdecode( );//COPYRIGHT (c) 2005, 2006 FastpathLogic Inc

module mengine(clock,
        reset_,
        ir,
        rf_ctrl,
        ctrl);
 input [0:0] clock;
```

```
  input [0:0] reset_;
  input [31:0] ir;
  output [25:0] rf_ctrl;
  output [5:0] ctrl;
  mmux mmux( );
  mpc mpc( );
  mrom mrom( );
  mir mir( );
  mdecode mdecode( );
  mctrl mctrl( );
endmodule

module mmux(ir_mux_jmpto,
            ml_start,
            ir_mux_addr,
            mux_pc_next_addr);
  input [5:0] ir_mux_jmpto;
  input [5:0] ml_start;
  input [0:0] ir_mux_addr;
  output [0:0] mux_pc_next_addr;
  //behavior
  assign mux_pc_next_addr = ir_mux_addr ? ml_start : ir_mux_jmpto;
  //end behavior
endmodule

module mpc(clock,
           reset_,
           ldmpc,
           mux_pc_next_addr,
           pc_rom_addr);
  input [0:0] clock;
  input [0:0] reset_;
  input [0:0] ldmpc;
  input [0:0] mux_pc_next_addr;
  output [5:0] pc_rom_addr;
endmodule

module mrom(pc_rom_addr,
            rom_ir_instr);
  input [5:0] pc_rom_addr;
  output [31:0] rom_ir_instr;

  reg [31:0] inst_rom[0:63];
  reg [31:0] instr;
  integer i;
  //behavior
  assign rom_ir_instr = inst_rom[pc_rom_addr];
```

```
  initial begin
     instr = 0;
      for(i=0; i<=63; i=i+1) begin
         inst_rom[i] = instr;
         instr = instr + 1;
      end
   end
   //end behavior
endmodule

module mir(clock,
          reset_,
          ldmir,
          rom_ir_instr,
          ir_mux_jmpto,
          ir_dec_instr);
  input [0:0] clock;
  input [0:0] reset_;
  input [0:0] ldmir;
  input [31:0] rom_ir_instr;
  output [5:0] ir_mux_jmpto;
  output [24:0] ir_dec_instr;
endmodule

module mdecode(ir_dec_instr,
              dec_ctrl_sgn,
              ctrl);
  input [24:0] ir_dec_instr;
  output [5:0] dec_ctrl_sgn;
  output [5:0] ctrl;
endmodule

module mctrl(clock,
            reset_,
            ldmpc,
            ldmir,
            dec_ctrl_sgn,
            ir_mux_addr);
  input [0:0] clock;
  input [0:0] reset_;
  output [0:0] ldmpc;
  output [0:0] ldmir;
  input [5:0] dec_ctrl_sgn;
  output [0:0] ir_mux_addr;
endmodule
```

CSL code:

```
//-----------------------------------------------------------------------
// Copyright (c) 2005, 2006, 2007 Fastpath Logic
// All Rights Reserved.
// This is UNPUBLISHED PROPRIETARY SOURCE CODE of Fastpathlogic;
// the contents of this file may not be disclosed to third parties,
// copied or duplicated in any form, in whole or in part, without the prior
// written permission of Fastpathlogic.
//
// RESTRICTED RIGHTS LEGEND:
// Use, duplication or disclosure by the Government is subject to
// restrictions as set forth in subdivision (c)(1)(ii) of the Rights in
// Technical Data and Computer Software clause at DFARS 252.227-7013,
// and/or in similar or succesor clauses in the FAR, DOD or NASA FAR Supplement.
// Unpublished rights reserved under the Copyright Laws of the United States
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
// design: Microcode Engine
// author: Catalin Lipsa
//-----------------------------------------------------------------------

csl_interface me_ifc{
//  csl_port_list (input, 1, csl_list(clock, reset_));
    csl_port ir_2 (output, 32);
    csl_port pc_2 (output,  6);
    csl_port v_2  (output,  1);
};
csl_unit mengine {
//csl_port_list    (input, csl_list(clock, reset_));
  csl_port ir      (input, 32);
  csl_port rf_ctrl (output, 26);
  csl_port ctrl    (output, 6);

  mmux    mmux;
  mpc     mpc;
  mrom    mrom;
  mir     mir;
  mdecode mdecode;
  mctrl   mctrl;

//    csl_signal alu_opcode = ir_3.alu_fmt.select_field(op2);

//create pipeline
/*
    csl_pipeline  me_pipeline(4);
```

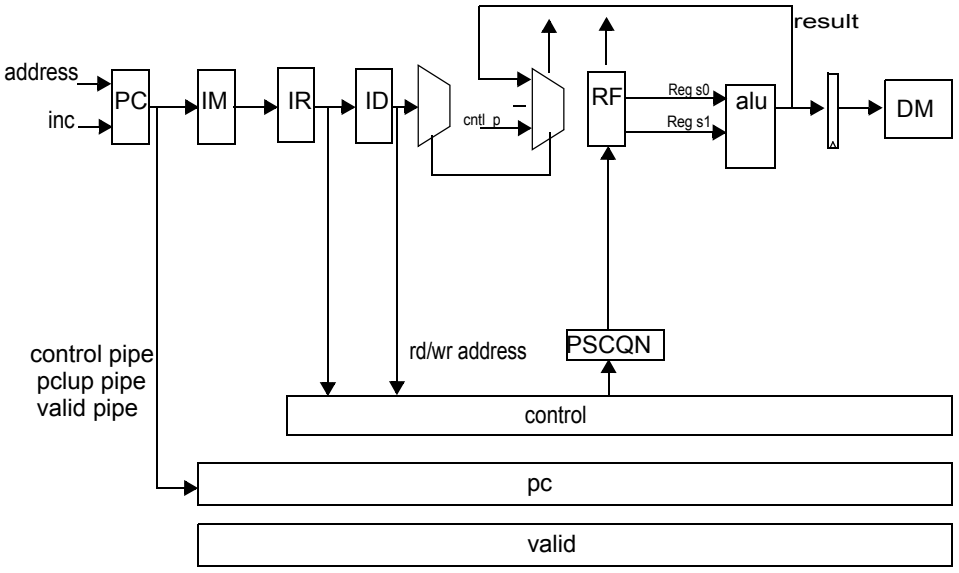
```
   csl_pipestage fetch(0);
   csl_pipestage decode(1);
   csl_pipestage rf_read(2);
   csl_pipestage execute(3);
   void initialize(){
      me_pipeline.add_pipestage_list(fetch, decode, rf_read, execute);

      me_pipeline.add_signal( 6, pc, 0, 2);
      //pipeline_name.add_signal(br, signal_name, start_stage, end_stage);
      me_pipeline.add_signal(32, ir, 1, 3);
      me_pipeline.add_signal( 1, v,  1, 3);
      me_pipeline.add_signal_list(1, csl_list(alu, mem), 2, 3);

      //create program counter register
      pc_0.set_type(pc);
      //connect branch control
      pc_0.connect_br_en(br_2);
      pc_0.connect_br_addr(br_addr_2);
   }

//create decoder
   csl_enum op_enum(csl_list("br_2", "alu_2", "mem_2"));
   ir_1.opcode.set_enum(op_enum);
   csl_signal op_dec; // width is created by the gen_decoder option
   ir_1.opcode.gen_decoder(op_dec);
*/
};


csl_unit mmux{
//csl_port_list           (input, 6, csl_list(ir_mux_jmpto, ml_start));
  csl_port ir_mux_addr     (input, 1);
  csl_port mux_pc_next_addr (output, 1);
};
csl_unit mpc{
//csl_port_list           (input, csl_list(clock, reset_));
  csl_port ldmpc           (input, 1);
  csl_port mux_pc_next_addr (input, 1);
  csl_port pc_rom_addr     (output, 6);
};

/*csl_register mrom{
  csl_port pc_rom_addr  (input, 6);
  csl_port rom_ir_instr (output, 32);
  void initialize(){
    csl_register
    set_type(rom);
```

```
   set_word_width(32);
   set_addr_width(6);
   set_data_sgn(ir_1);
   set_addr_sgn(pc_0);
  }
};*/

//Register File
/*csl_register_file mrf{
  void initialize(){
   set_width(32);
   set_depth(32);
   connect_rd_addr(i2.alu_fmt.select_field(src_a));
   connect_rd_addr(i2.alu_fmt.select_field(src_b));
   connect_wr_addr(i2.alu_fmt.select_field(dest ));
   connect_rd_data(rd_data_a_3);
   connect_rd_data(rd_data_b_3);
   connect_wr_data(wr_data_3);
  }
};*/

//ALU
csl_unit malu{
   csl_port alu_3      (input, 1);
   csl_port alu_opcode (input, 3);
//    csl_port_list       (input, 32, csl_list(rd_data_a_3, rd_data_b_3));
   csl_port wr_data_3  (output, 32);
};

//Memory unit
csl_unit mmem{
   csl_port mem_3       (input, 1);
   csl_port mem_opcode  (input, 3);
   csl_port rd_data_a_3 (input, 32);
   csl_port wr_data_3   (output, 32);
};

//ROM memory - Data Memory
/*csl_memory mram{
  void initialize(){
   set_type(ram);
   set_word_width(32);
   set_addr_width(6);
   //connection needed!
  }
};*/
```

```
csl_unit mir{
//csl_port_list        (input, csl_list(clock, reset_));
  csl_port ldmir        (input, 1);
  csl_port rom_ir_instr (input, 32);
  csl_port ir_mux_jmpto (output, 6);
  csl_port ir_dec_instr (output, 25);
};

csl_unit mctrl{
//csl_port_list        (input, csl_list(clock, reset_));
//csl_port_list        (output, csl_list(ldmpc, ldmir));
  csl_port dec_ctrl_sgn (input, 6);
  csl_port ir_mux_addr  (output, 1);
};

csl_unit mdecode{
  csl_port ir_dec_instr (input, 25);
  csl_port dec_ctrl_sgn (output, 6);
  csl_port ctrl         (output, 6);
};
```

I.D.=Instruction Decoder

FIGURE 1.27



```
csl_packet pkt;
csl_dataframe df;
csl_reg pc "program counter";
csl_reg ir "instruction register";
csl_mem im "instruction memory";
csl_rf rf
         : rf.valid (v);
csl_pscqn pscqn;
                 pscqn.input (rf.dataout)
                 pscqn.input (rf.v)
csl_fifo fifo
csl_pipeline p0
csl_pipeline p1
p0.stall (p1.stall & id.v);
pc.in (pcmux.out);
pcmux.select ();
pcmux.ino ();
pcmux.in1 ();
im.addr (pc.out);
im.wr_en ();
im.data_in ();
im.addr ();
im.rd_en ();
im.wr_en ();
irmux.select ();
irmux.im0 ();
irmux.in1 ();
ir (irmux);
id (iradr);
// id is an instruct decoder
//if is a case statement
//the case items are grnerated from the .op field in the packet class
hierarchy
p0 (id.v, id.cntl);
rf.addr (rfaddrmux);
rf.sc0 (id.src0);
rf.sc1 (id.src1);
rf.rd_en (id.rd_en);
rf.dst (id.rfdst);
rfaddrmux.in0 ();
```

```
rfaddrmux.in1 (alu.out, alu.v & cntl.rfdstop);
// data one ? not? select
// the select for the rfmux is one not
// the data and one not? select are pointed together
alu.src0
alu.src1
csl_data_frame_df;
df.num_pkts (4);
df.payload (pkt, pkt, pkt, pkt);
//data frame contains 4 packets of type pkt
//note that the data frame payload can contain any number of packets
//the order that the packets are listed in is the order that the pack-
ets are sent
csl_pkt pkt;
pkt.field (a, 16);
//    <name><range>
pkt.field (b, 16);
//ock contains 2 fields which are 16 bits wide
```

Memory Map
Register File
16 registers
Program Counter
Instruction Register
Instruction Memory

```
csl_mem_map mem_map;
mem_map.base_addr (10);
// starting address
mem_map.address (rf_mem_map,[0-15]);
//                                    <name> <range>
mem_map.address (im_mem_map, [32-46]);
mem_map.address (pc_mem_map, 100);
mem_map.address (ir_memmap, 101);
// first create the csl objects then create the memory locations
// or create the address space and associate the objects with addresses
im.address (im_mem_map);
ir.address (ir_mem_map);
rf.address (rf_mem_map);
im.address (im_mem_map);
```

The csl_mem_map is used to specify the memory map for a chip design or unit. The csl_mem_map memory space is associated with csl_objects and third party IP or user RTL addresses spaces.

FIGURE 1.28

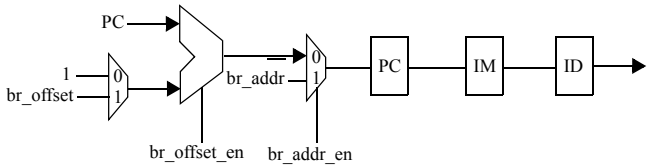


TABLE 1.4

1.5 CSL Microengine Command Summary

1.6 CSL Microengine Commands