CHAPTER 7 CSL Register File

TABLE 7.1 Chapter Overview

7.1 Definitions

7.1.1 Abreviations description

TABLE 7.2 Register file abreviations used to registers/fields in diagrams

Abreviation	Description
ar	address range
ext	the register is external to the register file
o	output - the register/field is an output of the block
rn	register name
cn	field name
v	valid - generate a valid bit
ws	word size
nw	number of words

7.2 CSL Register File Overview

7.2.1 CSL Register File Specification Description

Register files are declared using the CSL memory map specification and the CSL register file constructs.The CSL memory map specification is used to create named registers_fields and fields inside of the register file. All memory map operations are supported inside of the register file. The register file's registers can be connected to the inputs and the outputs of the register file. fields within the registers can also be connected to the inputs or outputs of the register file.

All CSL register operations are supported inside of the register file including clear, set, enable, and soft reset. We use the term register_field to refer to either a register or field inside of a register. Note that the group operation can group registers and fields within registers together so that common operations such as clear or set can be performed on the registers. Read and write operations can trigger events. Read operations can generate valid bits. The register file decodes an address and if the write enable signal is set then writes a value to a register. The argument *a nll* expands to all registers or words in the register file.

Register files are either instantiated inside of an RTL module using CSL commands or the register file is a stand alone module which we connect to the design using the CSL interconnect commands Typical Register File configuration options:

- Single read port
- Single write port
- Multiple read ports
- Multiple write ports
- Connect individual registers to an output
- read/write registers from more than one source
- num_rd_ports
- num_wr_ports

7.3 CSL Register File Concepts

A register file (**rf**) block is used to store values in registers. The registers are addressable by the read and write address lines. The read enable signal (rd_en) is used to access the memory array and return the contents of the addressed word. The register file contains a memory array.

7.3.1 Register File array implementation choices

The memory array can be constructed from flip-flops or a SRAM (Static RAM) array. The choice is based on the size of the memory array and the available technology options. Less than 1k bits is implemented with a non-SRAM (FF or latch array) and greater than 1k bits should be implemented with SRAM.

TABLE 7.3 Memory imlementation

implementation type	valid required	rd_en required
sram	0	1
sram	1	1
ff	0	0
ff	1	1

7.3.2 Register File clock inputs

The register file is a clocked device. It has one clock (clk) input signal. The clock name does not need to be specified in the CSL register file specification if the register file is instantiated in a module with only one clock. The module's clock is automatically connected to the register file in this case. If there are multiple clocks in the module in which the register file is instantiated then the CSL clock command has to be used to specify the clock name which is connected to the register file.

7.3.3 Address decoders

The register file contains read and write address decoders. The decoded output of the write address decoder is qualified (anded) with the write enable signal. The decoded read address output may also be qualified with a read enable signal. But this is not required. An event detector detects when a certain register or range of registers has been read or written and asserts an event output bit.

FIGURE 7.1 Register File architecture for DFF array implementation

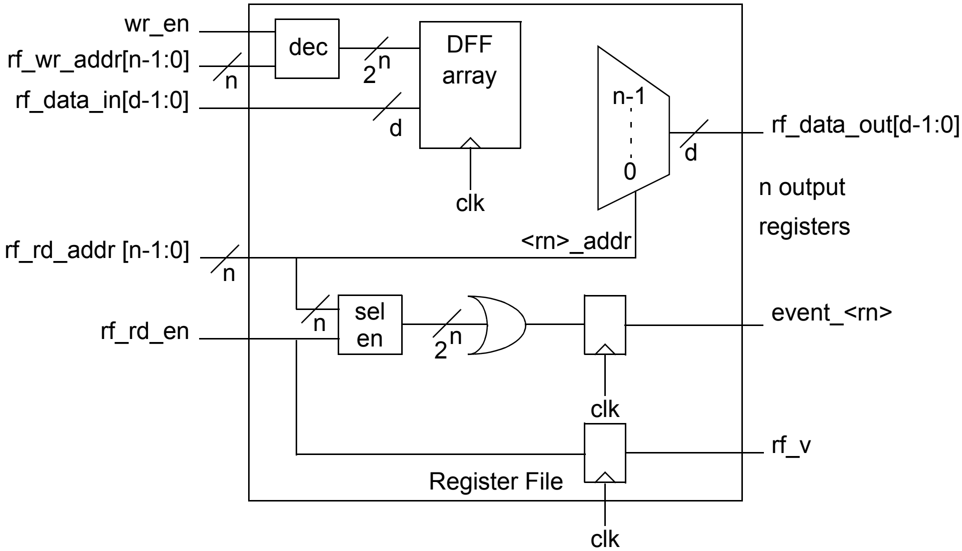
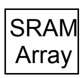


FIGURE 7.2 Register File architecture SRAM array implemetation



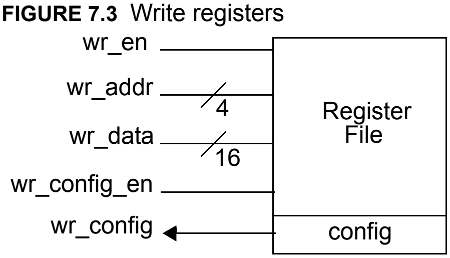
The SRAM contains the addr decode and word select logic there is no external maximum element when an SRAM is used.

7.3.4 Register File operations

7.3.4.1 Write registers

Figure 7.3 shows the write side of a register file which contains a named register. The named register can be written either using the wr_addr, wr_en, and wr_data signals or the wr_config_en and wr_data signals.

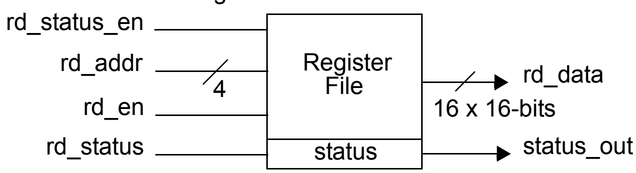
FIGURE 7.3 Write registers



7.3.4.2 Read registers

In a register file, individual registers can be named. The "named" can be accessed using the rf_addr and rd_en or read via a signal connected from the named register to the rf output called register_name.out
The status register can be read via rd_data or status_out signals. In Figure 7.4 we should see the read side of a register file which contains a named register. The named register can be read either using the rd_addr, rd_en signals or rd_status.

FIGURE 7.4 Read registers



The register with the symbolic name status can be read by data_out or status_out.

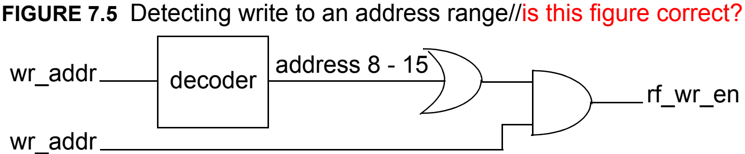
7.3.5 Register File addressing using relative address

EXAMPLE :
base address + register offset
The base address is 32000000. The register file address range is 0-63. The mask for the address range is 3200_0000. The address range is 3200_0000 to 0x3200_0063. A sparse address range may be specified : 0-3 and 10-5.

7.3.5.1 Register file address space

The Register file address space is the range of numbers from min_addr to max_addr. If the min_addr is greater then zero or the max_addr is not equal to a power of two, then the write/read address is checked for validity by a range checker. The range checker is implemented with a decoder. The output of the decoder is ored and output of the orgate is anded with the wr_en.
The Register file address space (i.e. starting and ending address) is defined with CSL memory map operations. Note that the register file memory map can have discontinuities or gaps in the address space. Named registers or registers with fields can be defined using CSL register construction.

FIGURE 7.5 Detecting write to an address range//is this figure correct?



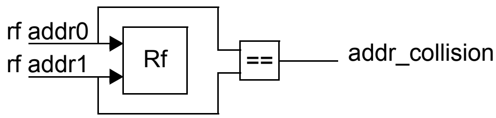
7.3.5.2 Errors

7.3.5.2.1 Address collision Detection logic

Register files with more than one write port will detect multiple writes to the same address location during the same cycle. This condition generates an error.
A register can be set to a constant value by setting the constant attribute to a value.

```
register_name.set_initial(numeric_expression);
register_name.set_constant_value(numeric_expression);
```

FIGURE 7.6

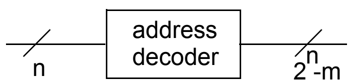


7.3.6 Address decoder

There can be one or more address decoders per register file.

An address decoder may decodes a subset of addresses from a total range (decode only 2^n-m addresses from an n bit input bus).

FIGURE 7.7 Incomplete address space



- n is the width of the bus
- m are the number of unused addresses in the address spaces

In Figure 1.6 the addr bus is n bits wide. However, m addreses are not present in the rfaddr.space. Therefor the rfaddr decoder has 2^n-m outputs.

7.3.6.1 Register file address errors

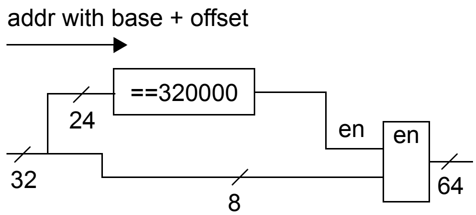
A checker will validate the addresses presented to the register file. If an address does not fall in the register file address range and the rd_en or wr_en bits are set then an error will be generated.

7.3.6.2 Accessing the register file in the global memory map

The register file is accessed with the base address + offset.
If the base address of the register file is 0x32000000 and the address range is 0-63 than the mask for the address is 0x3200_0000.
The address range in the global map is 0x3200_0000 to 0x3200_0063.

6 bits are used as the address offset into the register file.

FIGURE 7.8



7.3.7 Address options

The first address for the register file can be specified to enable read and writes.

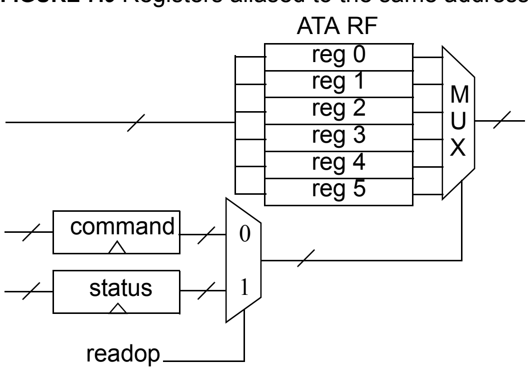
- Read address bypass logic

Writes to the register file during cycle n, and reads from the same address during cycle n+1, will result in the write data being forwarded.

7.3.8 Register Aliases with the same address within Register Files

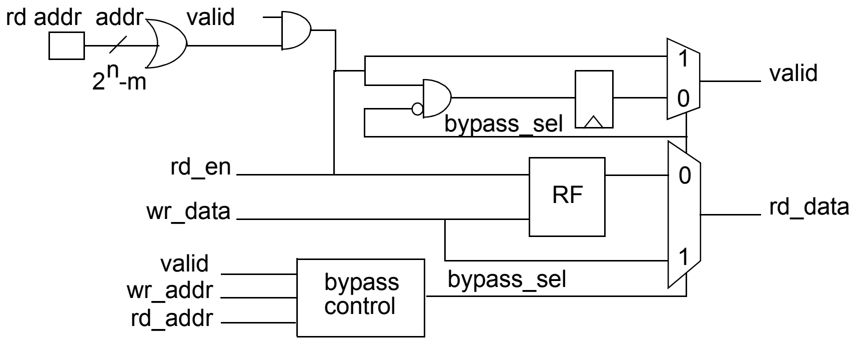
The same address may have different contexts depending on the address(?). For example in ATA (Advanced Technology Attachment) the address space has 0-15 logical addresses but the register file is only 8 addresses. The ATA registers have different contexts depending on whether the current operation is a read or is a write.For example, register address 7 is a status register and register address 15 is a command register, but they are both the same physical register. Register file outputs can be either a register or a field. Register file inputs can be either internal to the units which contains the register file and a register external to the register file but input to RF. Register address 7 is not an internal register instead Register address 7 is an input into the RF.

FIGURE 7.9 Registers aliased to the same address Register Bypassing



Some implementations of register files are optimized so that the read request bypasses the register file to save one clock cycle if the register being read has the same address as the value being written into the register file.

FIGURE 7.10 Register file bypass



The equation that implements the bypass_sel logic is

```
bypass_sel = (wr_addr == rd_addr) ? ~valid : 1'b0;
```

If the previous cycle did not have a valid read and the same address is being used to write and read, bypass logic is designed to forward writes to the output of the of register file. If there is valid data on the output then the bypass is disabled. A truth table for the bypass_control block is given in the next table :

TABLE 7.4 Truth table for bypass_control

rd_addr == wr_addr	valid	bypass_sel
0	0	0
1	0	1
0	1	0
1	1	0

TABLE 7.5 Register file bypass pipeline example transactions

wr_en	rd_en	wa	ra	w_eq_r	bp_sel	data_mux_output	v_mux_output
1	1	r4	r2	0	0	x	x
1	1	r2	r5	0	0	r2	1
1	1	r3	r0	0	0	r5	1
1	1	r1	r6	1	0	r0	1
1	0	r6	r7	1	0	r6	1
1	1	r5	r5	1	1	r5	1
1	1	r4	r4	0	1	r4	1

TABLE 7.6 RF bypass select truth table

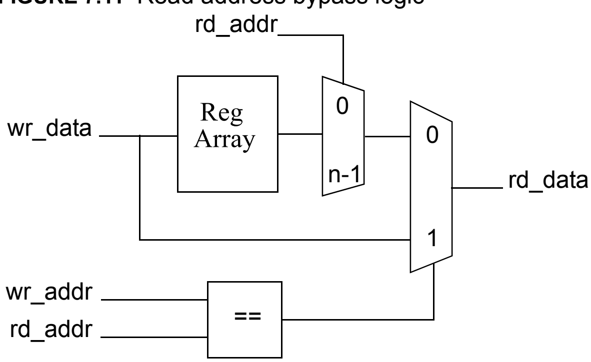
bp_sel_d0	v	wa_eq_ra_and_rd_en_and_wr_en	bp_sel
0	0	0	0
x	0	1	1
0	1	0	0
0	1	1	0

The contents of a shadow register file can be written in any order. The shadow register file is shifted into the current register file in one cycle. This one cycle update mechanism guarentees that the bits in the crf are updated at the same time. This allows software to write the shadow register file registers in any order.

7.3.9 Read address bypass logic

Writes to the register file during cycle n and reads from the same address during cycle n+1

FIGURE 7.11 Read address bypass logic



7.3.10 Register files for processor architectures

Register files for processor architectures have special requirements.

- valid bits
- operand bypassing

7.3.11 Output valid bits

The register files have an output valid which is the delayed version of the enable read bit. The delay is equal to the delay of the read request of the output. If the read is a bypass then the valid bit needs to be bypassed. Else if the read is the output of the register file then the delay is equal to the normal delay through the register file.
A valid bit may be optionally added to the register file. The register file will generate a valid bit whenever there is a read operation (i.e. rd_en is asserted) . The valid bit may be qualified if necesary with the OR of the read address decoder outputs to check the address range.
A valid bit is used in hw pipelines to indicate that the contents of the current pipeline is valid.
The valid bit (v) is the pipelined rd_en which corresponds to the data which will be read of the register file due to the read enable. If the rd_en is only associated with the register file then the read enable does not need to be qualified with the OR of the read address decoder's outputs.

7.3.12 Register file dataflow architecture

Valid transactions have a valid bit associated with them in the same pipestage. In effect, "data announces its arrival to the next pipestage".
Read transactions can have a valid bit associated with the output data .

FIGURE 7.12 Register file with unqualified valid bit

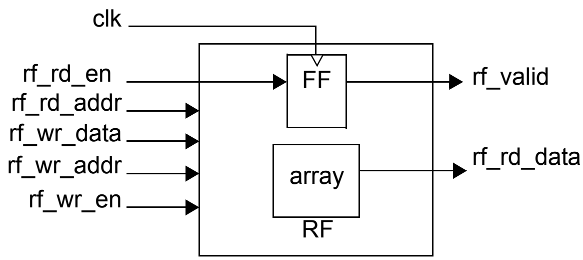
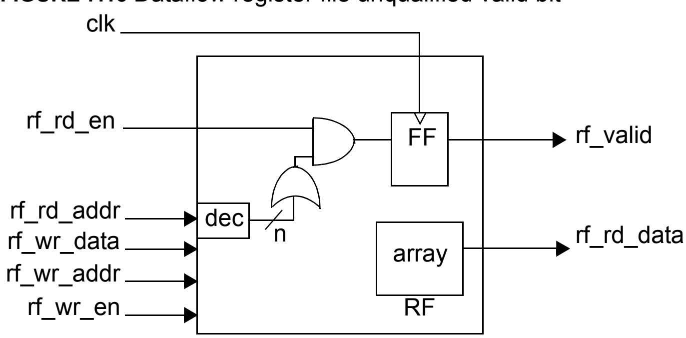
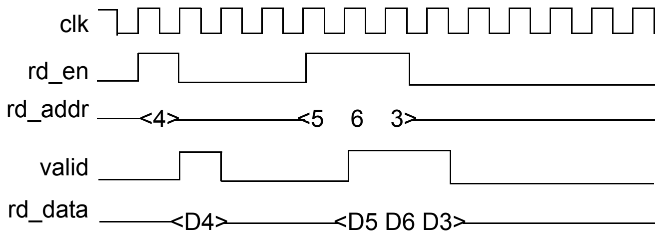


FIGURE 7.13 Dataflow register file unqualified valid bit



7.3.13 Dataflow register file read logic

FIGURE 7.14 Register file timing diagram example



There is an option for a bad address checker which sets an error bit, captures the bad address in a register and generates an interrupt.

7.3.14 Register File flags specifying registers/fields

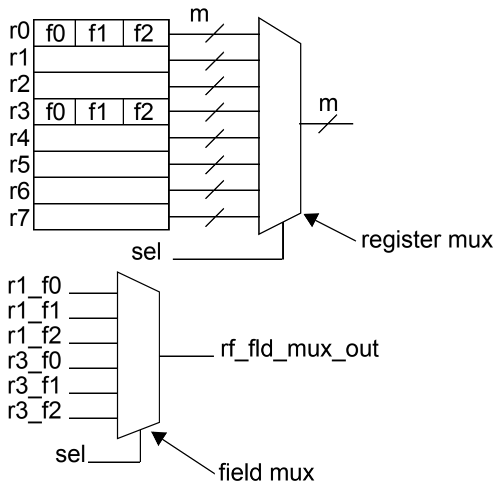
7.3.15 Register files are constructed hierarchically using elements

TABLE 7.7 Register file abbreviations used for registers/fields in diagrams

Abreviation	Description
v	valid - generate a valid bit at the register field
e	event - generate an event when the register field is written
x	external - the register field data is stored external to the block
n	normal - the register field is inside of the RF

Individual fields in registers in a register file may be accessed with a mux just as the individual registers in a register file are accessed with a mux.
In figure 1.14 registers rf[0] and rf[3] have individual fields defined with the csl_field method. Each individual field in rf[0] and rf[3] are connected to the rf_fld multiplexer which is connected to the rfoutput.

FIGURE 7.15 The rf_fld_mux_out signal



7.3.15.1 Register files are constructed hierarchically using elements

The register is declared with the name of the register, the range (this may be discontiguous if there are unused bits-a concatenation of ranges is allowed {[6:4],[1]}), the read/write/shadow bits, either an increment of the previous address or an explicit address in decimal or hexadecimal. Note that where numbers are allowed constants can be used instead.

7.3.15.2

The width of the fields cannot exceed the width of register. If the fields are defined by the **rws** (*read*, *write* and *shadow*) bits then the field uses the register's rws bits.

7.3.16 Register file Inputs

A register in a Register File can be written using either the data_in and the wr_addr, wr_en or a special input which is tied to a specific register along with a special wr_en.

TABLE 7.8 Table for signals

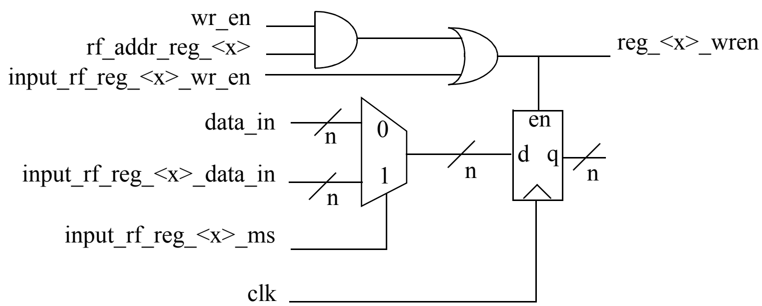
Signal	Description
wr_en	
rf_addr_dk_<x>	
input_rf_reg_<x>_wr_en	
data_in	

TABLE 7.8 Table for signals

Signal	Description
input_rf_reg_<x>_data_in	
input_rf_reg_<x>_ms	

7.3.16.1 RF Outputs

FIGURE 7.16 Register file inputs



7.3.17 Connecting register file inputs and outputs to registers and fields

Specifying that a register or field is connected to an individual input will not disconnect that register from data_in. The register can still be written using the global register file address, write enable, and data_in signals. The default write action uses the global write signals. There are multiplexers connected to each of the register/fields which override the data_in and write enable signals for each register when the signal *register_field_name*_wr_en is asserted. *register_field_name*_wr_en is the mux select for the inputs to the register/field.

7.3.18 Register Files event detectors

- n is the width of the bus
- m is the number of unused addresses in the addresses spaces

Note: the verilog implementation of the register file address decoder is 1 << addr
If a event detector is added to ... a unit to a rf then the event detection is added outside of the rf and the rf's write addr decoder outputs are connected to the event detector.

FIGURE 7.17

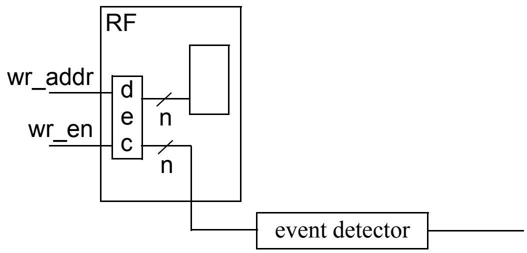
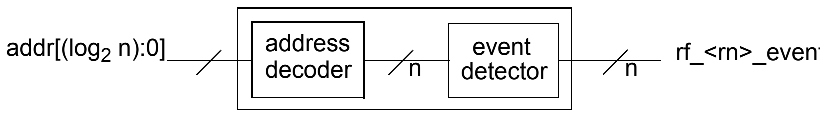


FIGURE 7.18 Register File with an event detector

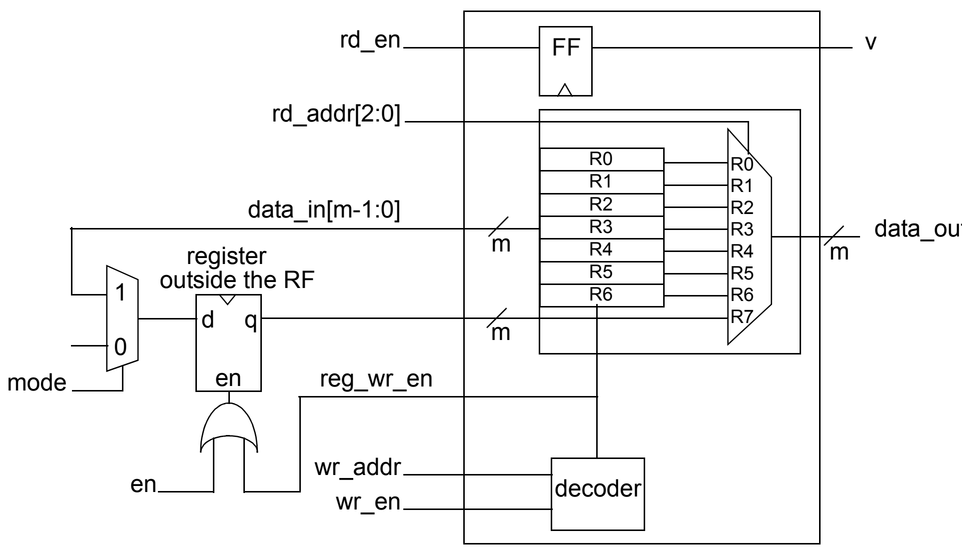


7.3.19 Register File with external register

The external register is an input to the output mux.

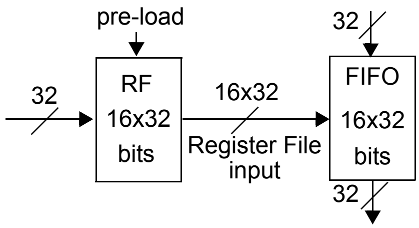
The external write enable is generated by the register file's address decoder.

FIGURE 7.19 Register outside of Register File



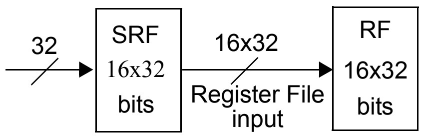
7.3.20 Preloading register files

FIGURE 7.20 Pre-loading a register file and shifting its contents into a FIFO in one cycle



The contents of a register file may be written and the entire register file can be shifted into a FIFO in one cycle.
This can be used to readout (pop) the words in the FIFO while at the same time loading the next sequence of commands.

FIGURE 7.21 Pre-loading a register file and shifting its contents into a SRF in one cycle

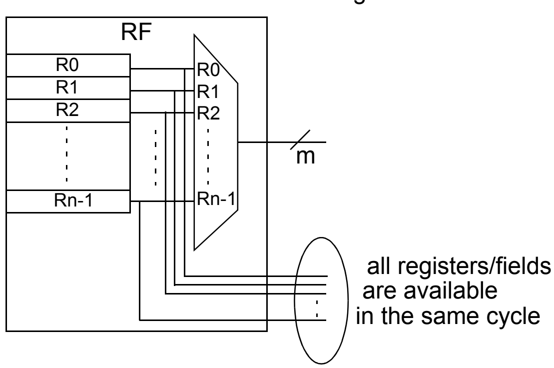


The entire contents of the Shadow Register File is moved in one cycle into the register file.
A simultaneous write from the Shadow Register File to the register file in one cycle of all configuration registers in an address space guarantees that the bits in the current register file will not conflict with each other if they were programmed correctly by the software layer. Each register in the Shadow Register File can be updated in any order.

7.3.20.1 Make example with individual inputs connected to the registers

7.3.20.2 Make example with individual outputs connected to the output signals

FIGURE 7.22 Connect individual registers or fields to outputs



7.3.21 Register file ports and logic

When creating a register file, the compiler automatically creates default ports and logic for the respective unit. Custom **add_logic()** commands add functionality and/or ports to the unit. The ports, their functionality and naming is detailed below:

TABLE 7.9 Register File's ports

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	clock
wr_addr	i	ud	Automatically	write address
rd_addr	i	ud	Automatically	read address
wr_data	i	ud	Automatically	write data
rd_data	o	ud	Automatically	read data
wr_en	i	1	Automatically	write enable
rd_en	i	1	Automatically	read enable
reset	i	1	Automatically	asyncroneus reset
clear	i	1	Automatically	clear / init
valid_<register_file>	o	1	add_logic(read_valid);	valid read

NOTE: Dir = port direction, i = input, o = outpt, w = port width, ud = user defined

NOTE: THE ADD_LOGIC(RD|WR_CHANNEL, [PREFIX]) HAS BEEN MOVED TO CSL_LANGUAGE BECAUSE IS USED ALSO FOR FIFO

Ports automatically generated by cslc for register file:

`port: input - clock`

DESCRIPTION :

The clock port called clock is created automatically for the regiter file regiter_file_name. If the clock is not specified then the module clock will be used. This default only works when there is one module clock.

`port: input - wr_addr`

DESCRIPTION :

The write address port called wr_addr is created automatically for the regiter file regiter_file_name. The write address signal that is used to select one register from register file to be writing.

`port: input - rd_addr`

DESCRIPTION :

The read address port called rd_addrs is created automatically for the regiter file regiter_file_name. Through this port will be connected the read address signal that is used to select one register from register file to be read.

`port: input - wr_data`

DESCRIPTION :

The write data port called wr_data is created automatically for the regiter file regiter_file_name. Through this port will be connected the write data signal.

`port: output - rd_data`

DESCRIPTION :

The read data port called rd_data is created automatically for the regiter file regiter_file_name. Through this port will be connected the read data signal.

`port: input - wr_en`

DESCRIPTION :

The write enable port is automatically created for the register_file_name. This port enables the writes to register file.

`port: input - rd_en`

DESCRIPTION :

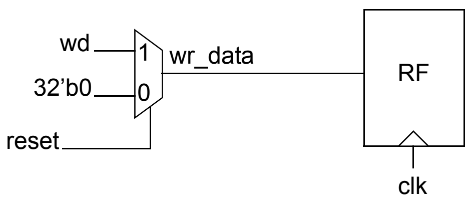
The read enable port is automatically created for the register_file_name. This port enables the reads from register file.

`port: input -  reset`

DESCRIPTION :

The asynchronous reset port called reset_ is created automatically for the regiter file regiter_file_name. On the negative level of the reset signal in all registers from the register file will be stored logic 0.

FIGURE 7.23 Register File with reset signal



`port: input - clear`

DESCRIPTION :

The clear port called clear is created automatically for the regiter file regiter_file_name. On the positive level of the clear signal the register file will be charged with the clear value.

`port: output v_reg_name`

FIGURE 7.24 Generate the valid signal

Notes: all default/non-default ports will exist in the register file scope. To access them the user uses hierarchical identifer; example *register_file_name.port_name*

7.4 CSL Register File Examples

7.4.1 Register file with no special options

Figure 7.25 shows the inputs to a black box labeled RF. The black box is a register file (RF). The implementation of the black box can be inferred from the CSL code.

7.4.1.1 CSL code

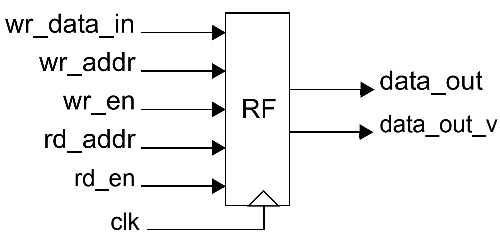
7.4.1.2 C++ code

7.4.2 Register file with read valid bit

7.4.2.1 CSL code

```
csl_register_file rf;
rf.width(D_WIDTH);
rf.depth(A_WIDTH);
rf.clock(clk) ;
rf.valid();
rf.rd_en();
```

FIGURE 7.25 Register file with read valid bit



7.4.2.2 Verilog code

```
module register_file(clk, reset, rf_wr_data_in, rf_wr_addr, rf_wr_en,
rf_status_wr_data_in, rf_status_wr_addr, rf_status_wr_en, rf_data_out,
rf_rd_addr, rf_rd_en);
   parameter A_WIDTH =8;
   parameter D_WIDTH =8;
   input [D_WIDTH - 1: 0] rf_wr_data_in;
```

```
 input [A_WIDTH - 1: 0] rf_wr_addr;
 input  rf_wr_en, reset, clk,  rf_status_wr_en;
 input [D_WIDTH - 1: 0] rf_status_wr_data_in;
 input [A_WIDTH - 1: 0] rf_status_wr_addr;
 output [D_WIDTH - 1: 0] rf_data_out;
 input [A_WIDTH - 1: 0] rf_rd_addr;
 input rf_rd_en;
 reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0], rf_data_out;
always @ (posedge clk) begin
    if(rf_wr_en) begin
       rf[rf_wr_addr] <= rf_wr_data_in;
    end
    if(rf_rd_en) begin
     rf_data_out <= rf[rf_rd_addr];
    end
end
 always @ (posedge clk) begin
    if(rf_status_wr_en) begin
       rf[rf_status_wr_addr] <= rf_status_wr_data_in;
    end
 end
 endmodule
```

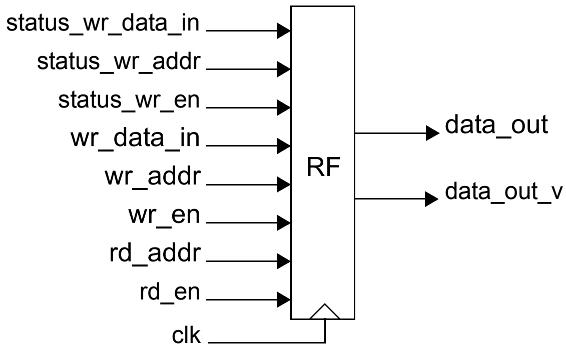
7.4.2.3 C++ code

7.4.3 Register file with write to an individual register named status

7.4.3.1 CSL code

```
csl_register_file rf;
rf.width(D_WIDTH);
rf.depth(A_WIDTH);
rf.clock(clk) ;
rf.valid();
rf.named_register(status, 12) ; // address 12 has the name status asso-
ciated with it
rf.connect_input_to_registers_fields(status) ;
```

FIGURE 7.26 Register file with write to an individual register named status



7.4.3.2 Verilog code

```
module register_file(clk, reset, rf_wr_data_in, rf_wr_addr, rf_wr_en,
rf_status_wr_data_in, rf_status_wr_addr, rf_status_wr_en, rf_data_out,
rf_rd_addr, rf_rd_en);
   parameter A_WIDTH =8;
   parameter D_WIDTH =8;

   input [D_WIDTH - 1: 0] rf_wr_data_in;
   input [A_WIDTH - 1: 0] rf_wr_addr;
   input clk, reset, rf_wr_en;
   input [D_WIDTH - 1: 0] rf_status_wr_data_in;
   input [A_WIDTH - 1: 0] rf_status_wr_addr;
   input                  rf_status_wr_en;
   output [D_WIDTH - 1: 0] rf_data_out;
   input [A_WIDTH - 1: 0] rf_rd_addr;
   input  rf_rd_en;
   reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0], rf_data_out;
   always @ (posedge clk) begin
      if(rf_wr_en) begin
         rf[rf_wr_addr] <= rf_wr_data_in;
      end
      if(rf_rd_en) begin
       rf_data_out <= rf[rf_rd_addr];
      end
  end
   always @ (posedge clk) begin
```

```
        if(rf_status_wr_en) begin
          rf[rf_status_wr_addr] <= rf_status_wr_data_in;
        end
      end
    endmodule
```

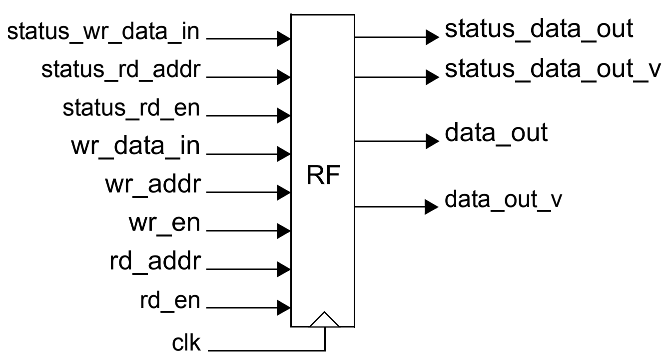
7.4.3.3 C++ code

7.4.4 Register file with a read from an individual register named status

7.4.4.1 CSL code

```
csl_register_file rf;
rf.width( D_WIDTH );
rf.depth( A_WIDTH );
rf.clock( clk );
rf.valid();
rf.named_register( status, 12 ) ; // address 12 has the name status associated with it
rf.connect_output_to_registers_fields( status ) ;
```

FIGURE 7.27 Register file with a read from an individual register named status// FIX



7.4.4.2 Verilog code

```
module register_file(clk, reset, rf_wr_data_in, rf_wr_addr, rf_wr_en,
rf_data_out, rf_rd_addr, rf_rd_en);
```

```
  parameter A_WIDTH =8;
  parameter D_WIDTH =8;
  input [D_WIDTH - 1: 0] rf_wr_data_in;
  input [A_WIDTH - 1: 0] rf_wr_addr;
  input  clk, reset, rf_wr_en;
  reg  rf_status_rd_data_in,rf_status_rd_addr,rf_status_rd_en;
  output [D_WIDTH - 1: 0] rf_data_out;
  input [A_WIDTH - 1: 0] rf_rd_addr;
  input  rf_rd_en;
  reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0];
  reg [D_WIDTH - 1: 0] rf_data_out;
  always @ (posedge clk) begin
     if(rf_wr_en) begin
        rf[rf_wr_addr] <= rf_wr_data_in;
     end
     if(rf_rd_en) begin
      rf_data_out <= rf[rf_rd_addr];
     end
 end
  always @ (posedge clk) begin
     if(rf_status_rd_en) begin
        rf[rf_status_rd_addr] <= rf_status_rd_data_in;
     end
  end
endmodule
```

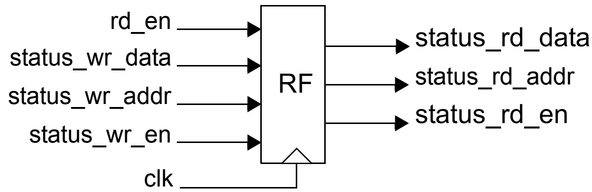
7.4.4.3 C++ code

7.4.5 Register file with read enable

7.4.5.1 CSL code

FIX

FIGURE 7.28



7.4.5.2 Verilog code

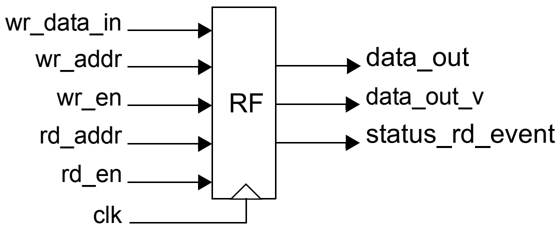
```
module register_file(clk, reset, rf_rd_en, rf_status_wr_addr,
rf_status_wr_en, rf_status_rd_data,
rf_status_rd_addr,rf_status_wr_data, rf_status_rd_en);
   parameter A_WIDTH =8;
   parameter D_WIDTH =8;
   input [D_WIDTH - 1: 0] rf_status_wr_data;
   input [A_WIDTH - 1: 0] rf_status_wr_addr;
   input  clk, reset, rf_rd_en, rf_status_wr_en;
   output [D_WIDTH - 1: 0] rf_status_rd_data;
   output [A_WIDTH - 1: 0] rf_status_rd_addr;
   output rf_status_rd_en;
   reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0];
   reg [D_WIDTH - 1: 0] rf_status_rd_data;
   always @ (posedge clk) begin
      if(rf_status_wr_en) begin
         rf[rf_status_wr_addr] <= rf_status_wr_data;
      end
      if(rf_rd_en) begin
       rf_status_rd_data <= rf[rf_status_wr_data];
      end
  end
endmodule
```

7.4.5.3 C++ code

7.4.6 Register file with an event generated from a read to an individual register named status

7.4.6.1 CSL code

FIGURE 7.29 Register file with an event generated from a read to an individual register named status



7.4.6.2 Verilog code

```
 module register_file(clk, reset, rf_wr_data_in, rf_wr_addr, rf_wr_en,
rf_data_out, rf_rd_addr, rf_rd_en, rf_status_rd_event);
   parameter A_WIDTH =8;
   parameter D_WIDTH =8;
   input [D_WIDTH - 1: 0] rf_wr_data_in;
   input [A_WIDTH - 1: 0] rf_rd_addr;
   input [A_WIDTH - 1: 0] rf_wr_addr;
   input  clk, reset, rf_wr_en,rf_rd_en;
   reg  rf_status_rd_data_in, rf_status_rd_addr, rf_status_rd_en;
   output [D_WIDTH - 1: 0] rf_data_out;
   output rf_status_rd_event;
   reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0];
   reg [D_WIDTH - 1: 0] rf_data_out;
   always @ (posedge clk) begin
      if(rf_wr_en) begin
         rf[rf_wr_addr] <= rf_wr_data_in;
      end
      if(rf_rd_en) begin
       rf_data_out <= rf[rf_rd_addr];
      end
  end
```

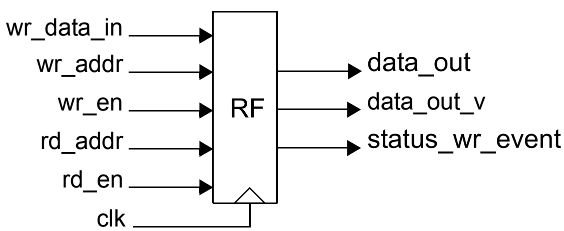
```
    always @ (posedge clk) begin
       if(rf_rd_en) begin
           rf_status_rd_event <= rf[rf_rd_addr] ;
       end
    end
 endmodule
```

7.4.6.3 C++ code

7.4.7 Register file with an event generated from a write to an individual register named status

7.4.7.1 CSL code

FIGURE 7.30 Register file with an event generated from a write to an individual register named status



7.4.7.2 Verilog code

```
  module register_file(clk, reset, rf_wr_data_in, rf_wr_addr, rf_wr_en,
 rf_data_out, rf_rd_addr, rf_rd_en, rf_status_wr_event);
    parameter A_WIDTH =8;
    parameter D_WIDTH =8;
    input [D_WIDTH - 1: 0] rf_wr_data_in;
    input [A_WIDTH - 1: 0] rf_rd_addr;
    input [A_WIDTH - 1: 0] rf_wr_addr;
    input  clk, reset, rf_wr_en,rf_rd_en;
    reg  rf_status_rd_data_in, rf_status_rd_addr, rf_status_rd_en;
    output [D_WIDTH - 1: 0] rf_data_out;
```

```
    output rf_status_wr_event;
    reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0];
    reg [D_WIDTH - 1: 0] rf_data_out;
    always @ (posedge clk) begin
       if(rf_wr_en) begin
          rf[rf_wr_addr] <= rf_wr_data_in;
       end
       if(rf_rd_en) begin
        rf_data_out <= rf[rf_rd_addr];
       end
   end
    always @ (posedge clk) begin
       if(rf_rd_en) begin
         rf_status_wr_event <= rf[rf_rd_addr] ;
       end
    end
  endmodule
```

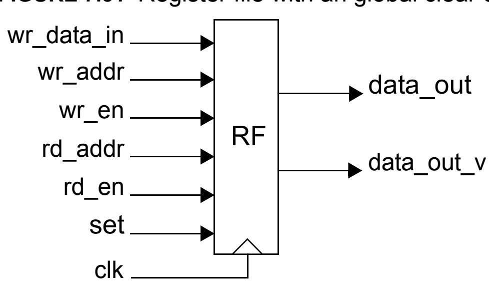
7.4.7.3 C++ code

7.4.8 Register file with an global set operation which sets all registers to a known value

7.4.8.1 CSL code

Register file with a global set operation which sets all registers to a known value.

FIGURE 7.31 Register file with an global clear operation which clears all registers to zero



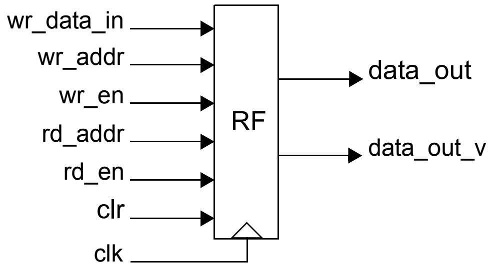
7.4.8.2 Verilog code

```
module register_file(clk, init, rf_rd_en, rf_wr_addr, rf_wr_data_in,
rf_wr_en, rf_rd_addr, rf_data_out,rf_data_out_v);
   parameter A_WIDTH =8;
   parameter D_WIDTH =8;
   integer i;
   input [D_WIDTH - 1: 0] rf_wr_data_in;
   input [A_WIDTH - 1: 0] rf_wr_addr;
   input [A_WIDTH - 1: 0] rf_rd_addr;
   input  clk,init rf_rd_en, rf_wr_en;
   output [D_WIDTH - 1: 0] rf_data_out;
   output [D_WIDTH - 1: 0] rf_data_out_v;
    reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0];
   reg [D_WIDTH - 1: 0] rf_data_out;
   always @ (posedge clk) begin
      if(rf_wr_en) begin
         rf[rf_wr_addr] <= rf_wr_data_in;
      end
      if(rf_rd_en) begin
      rf_data_out <= rf_wr_data_in;
      end
  end
  always @ (posedge clk) begin
   if (init) begin
    for (i = 0; i < A_WIDTH-1; i = i + 1) begin
  rf[i] <= D_WIDTH-1'b0;
  end
  end
 end
endmodule
```

7.4.9 Register file with an global clear operation which clears all registers to zero

7.4.9.1 CSL code

FIGURE 7.32 Register file with an global clear operation which clears all registers to zero



7.4.9.2 Verilog code

```
module register_file(clk, clr, rf_rd_en, rf_wr_addr, rf_wr_data_in,
rf_wr_en, rf_rd_addr, rf_data_out,rf_data_out_v);
   parameter A_WIDTH =8;
   parameter D_WIDTH =8;
   integer i;
   input [D_WIDTH - 1: 0] rf_wr_data_in;
   input [A_WIDTH - 1: 0] rf_wr_addr;
   input [A_WIDTH - 1: 0] rf_rd_addr;
   input  clk,clr, rf_rd_en, rf_wr_en;
   output [D_WIDTH - 1: 0] rf_data_out;
   output [D_WIDTH - 1: 0] rf_data_out_v;
    reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0];
   reg [D_WIDTH - 1: 0] rf_data_out;
   always @ (posedge clk) begin
      if(rf_wr_en) begin
         rf[rf_wr_addr] <= rf_wr_data_in;
      end
      if(rf_rd_en) begin
      rf_data_out <= rf_wr_data_in;
      end
  end
  always @ (posedge clk) begin
   if (clr) begin
    for (i = 0; i < A_WIDTH-1; i = i + 1) begin
  rf[i] <= D_WIDTH-1'b0;
  end
  end
 end
```

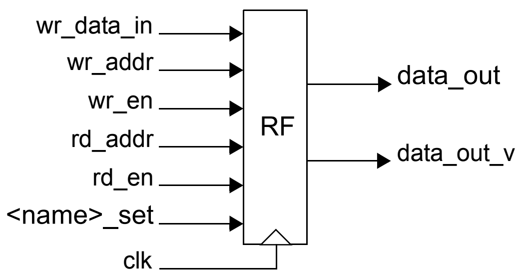
```
endmodule
```

7.4.9.3 C++ code

7.4.10 Register file with a set operation on a specific register/field or register/field group which sets the registers/fields to a known value

7.4.10.1 CSL code

FIGURE 7.33 Register file with a set operation on a specific register/field or register/field group which sets the registers/fields to a known value

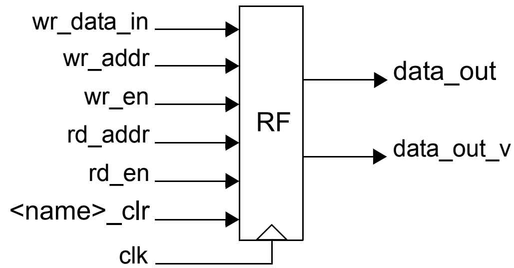


7.4.10.2 Verilog code

7.4.11 Register file with a clear operation on a specific register/field or register/field group which clears the registers/fields to zero

7.4.11.1 CSL code

FIGURE 7.34 Register file with a clear operation on a specific register/field or register/field group which clears the registers/fields to zero



7.4.11.2 Verilog code

7.4.12 nothing here ?

7.4.12.0.1 CSL code

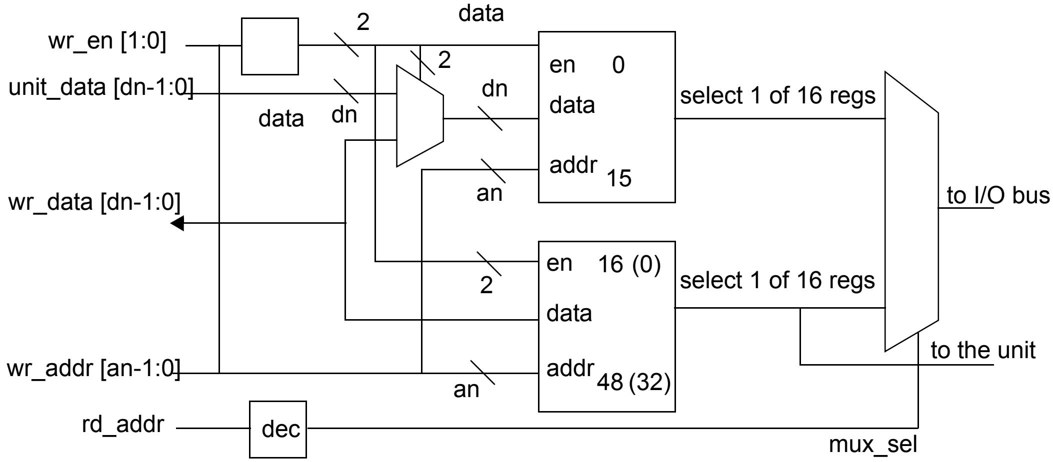
7.4.12.0.2 Verilog code

7.4.13 Building trees of Register Files

Multiple Register Files may be used to create a single address space within a unit.

7.4.13.0.1 CSL code

FIGURE 7.35 Combining register Files

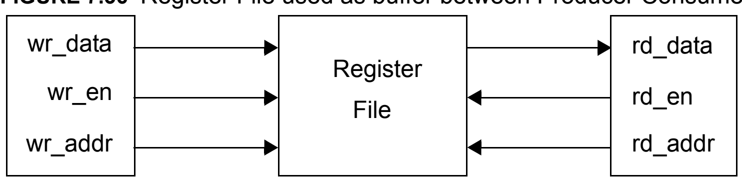


7.4.13.0.2 Verilog code

7.4.14 Producer/consumer register file buffer

7.4.14.1 CSL code

FIGURE 7.36 Register File used as buffer between Producer Consumer modules



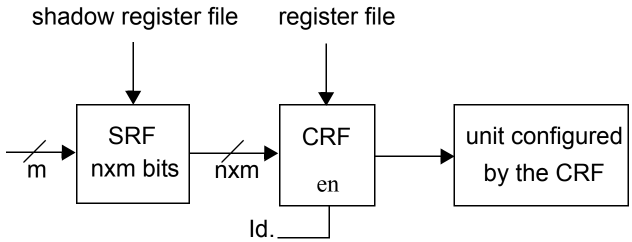
7.4.14.2 Verilog code

7.4.14.3 Shadow register within R.F.'s

One register can shadow another register which is why a register can be an element and an element can be a register (register linking).

7.4.14.3.1 CSL code

FIGURE 7.37 Shadow register file configuration



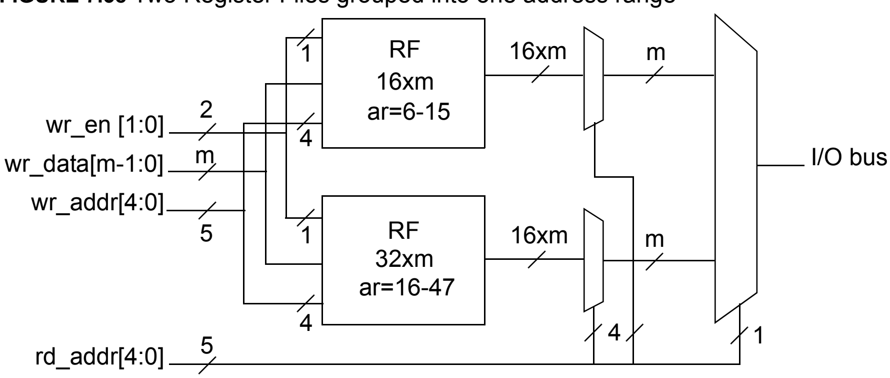
7.4.14.3.2 Verilog code

7.4.15 Grouping Register Files into one address space

Multiple Register Files can be gromped into one address space using wrapper logic.

7.4.15.1 CSL code

FIGURE 7.38 Two Register Files grouped into one address range



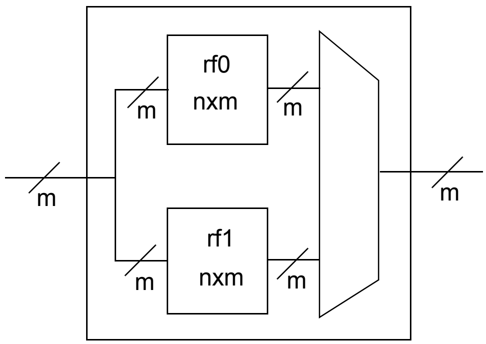
7.4.15.1.1 Verilog code

7.4.16 “Ping Pong” register file

A “Ping Pong” register file architecture can be used to switch between two identical register files at any clock edge.

7.4.16.1 CSL code

FIGURE 7.39 “Ping Pong”



7.4.16.1.1 Verilog code

7.4.17 no title here

7.4.17.0.1 CSL code

```
csl_register_file rf;
rf.width(D_WIDTH);
rf.depth(A_WIDTH);
rf.awc();//bitrange defaults to width
rf.arc();//bitrange defaults to log2(depth)
```

FIGURE 7.40

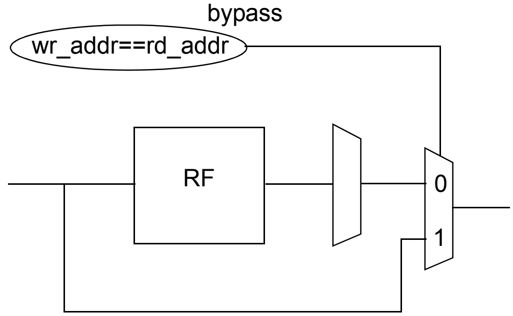
7.4.17.1 Verilog code

```
module register_file(clk, reset, wr_data_in, wr_addr, wr_en, rd_addr,
data_out);
   input [D_WIDTH - 1: 0] wr_data_in;
   input [A_WIDTH - 1: 0] wr_addr;
   input [A_WIDTH - 1: 0] rd_addr;
   input [A_WIDTH - 1: 0] ?!?
   output [D_WIDTH - 1: 0] data_out;
   reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0];
   always @ (posedge clk) begin
      if(wr_en) begin
         rf[wr_addr] = wr_data_in;
      end
      data_out = rf[rd_addr];
   end
end module
```

7.4.18 Register file with bypass

7.4.18.1 CSL code

FIGURE 7.41 Register File with bypass



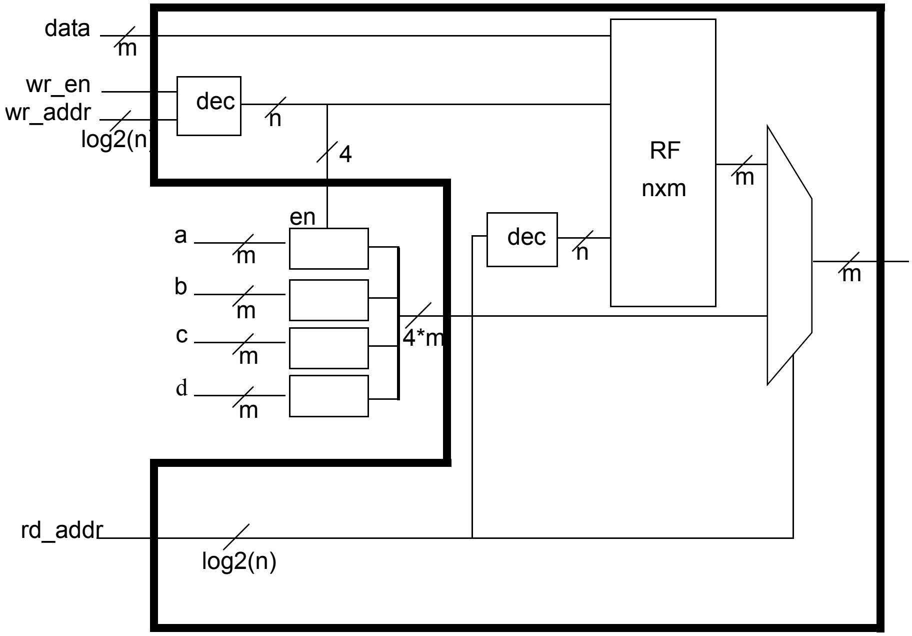
7.4.18.2 Verilog code

7.4.18.3 C++ Code

7.4.19 Register file with external registers

7.4.19.1 CSL code

FIGURE 7.42 External registers connected to register file



7.4.19.1.1 Verilog code

7.4.19.1.2 C++ Code

7.5 CSL Register File Checker

7.5.1 CSL Register File Reports

<BEGIN OF MOVED HERE SECTION>

<move this to register files>

7.5.2 constant look up tables

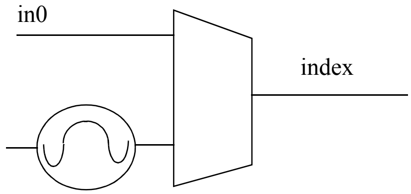
ROMS or memories which contain numeric constants can be used as LUT. !!create CSL type for this and expand it. Filters use look up tables (LUT) to determine the values rather than calculating the values.

<add figure of a memory with constant values in it here use figure 1.42 and add memory>

Typical filter LUT operations include determining the distance (radius) between the center of a filter and a point in a cartesian plane.

```
for (i=
             .a (table[i])
creating filter arrays
```

FIGURE 7.43 see Paper 5 note1(optional)

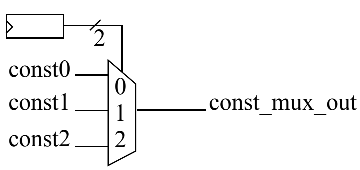



```
for (x=0  x<3){
 for (y=0   y=3){
  if (x=0) set in y
  else
  {( in0(
  (         )
out (              )
```

</move>
<move>
Transform the picture into an algorithmic class

constant mux/RAM

FIGURE 7.44



where const is a number
CSL constant mux
we should supply constants in binary
PI 3.1416..
e 2.78

</move>

<move to register file>

7.5.2.1 Register file status and interrupts

A hardware checker will identify bad addresses. A bad address is a n address which is not defined in the memory map.1 The hardware checker will set an error bit, capture the bad address in a register and generates an interrupt.

7.5.2.2 Inputs to aggregate (RF) memory mapped structures

Signals can drive the mux outputs in the generated memory mapped structure. This is allowed so that registers do not have to be present in the memory structure and the output decoder and multiplexer logic can be used to select the register which is not in the generated code.